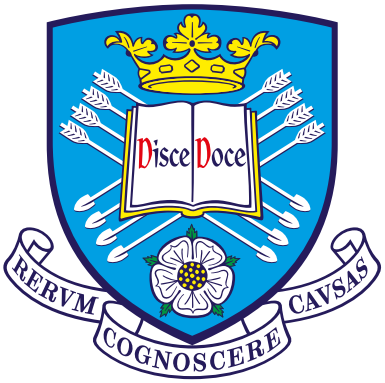
Coverage & cooperation:

Completing complex tasks as quickly as possible using teams of robots



Isaac Vandermeulen

Automatic Control and Systems Engineering

University of Sheffield

A thesis submitted for the degree of

Doctor of Philosophy

September 2019

Abstract

As the robotics industry grows and robots enter our homes and public spaces, they are increasingly expected to work in cooperation with each other. My thesis focuses on multirobot planning, specifically in the context of *coverage* robots, such as robotic lawnmowers and vacuum cleaners.

Two problems unique to multirobot teams are task allocation and search. I present a task allocation algorithm which balances the workload amongst all robots in the team with the objective of minimizing the overall mission time. I also present a search algorithm which robots can use to find lost teammates. It uses a probabilistic belief of a target robot's position to create a planning tree and then searches by following the best path in the tree.

For robust multirobot coverage, I use both the task allocation and search algorithms. First the coverage region is divided into a set of small coverage tasks which minimize the number of turns the robots will need to take. These tasks are then allocated to individual robots. During the mission, robots replan with nearby robots to rebalance the workload and, once a robot has finished its tasks, it searches for teammates to help them finish their tasks faster.

Preface

It was January 2016. I was living in Kingston, tired of another Canadian winter that only seemed to get worse with climate change and frustrated that I was still in Chemical Engineering. I was nearing the end of my masters—my third degree at this school—and was very ready for a new program in a new department in a new country.

This change had been a long time coming. I was a third year student majoring in engineering chemistry—a hybrid of chemistry and chemical engineering—excited to finally take the infamous quantum chemistry class. Despite its challenging reputation, I longed to get deep into the mathematics of atoms and molecules. It turned out that this course would shatter my entire understanding of electrons, of atoms, of chemistry itself! Electrons aren't simple point-like particles, but rather this kind of spread-out cloud called electron density. Without being able to concretely describe individual electrons as being located in a single location, the concepts of chemical bonds and molecules make little sense. The entire field of chemistry was built on wrong assumptions! No wonder every "rule" in organic chemistry seems to have more exceptions than examples that actually follow the rule. Although nobody else seemed to mind, at this moment I lost all respect for the subject.

Over the next year and a half, I struggled to be engaged in any chemistry courses. I picked up sloppy lab techniques, eyeballing things instead of measuring and sometimes being so careless as to throw the wrong solution down the sink.

Ideally, after that quantum course, I would have switched my major from engineering chemistry to chemical engineering, essentially replacing all of my chemistry courses with electives. If only I had been allowed to switch. Two years earlier I had enrolled in a dual degree program and was concurrently taking additional math courses towards a separate math degree. Academic regulations prevented me from switching my engineering major while continuing with the math degree, so I decided to grit my teeth and finish the engineering chemistry degree.

Meanwhile, I loved the math courses. They had the level of rigour that I craved and couldn't get from chemistry. An afternoon of group theory, partial differential equations, or cryptography would completely make up for a morning stuck in a chemical lab. I knew that whatever work I did in the future would have to be more mathematical.

At this point it was 2014. The price of oil was high. All of my friends were moving out west—mostly to Alberta—to go work in Canada's booming oil industry. I was tempted by the lucrative salaries and a two-weeks-on-two-weeks-off schedule that would afford plenty of opportunity for travel and adventure but ultimately I was still drawn to the idea of grad school.

Robotics had long been in the back of my mind as a field I'd like to switch to. I kept hearing more and more about it in the media and it seemed full of interesting yet practical math problems. But I didn't have a background in mechanical engineering or computer science and what good would a chemical engineer be in robotics?

Without the confidence to move directly to robotics, I decided to use my masters a stepping stone in that direction. Control theory—the only real branch of mathematics used heavily by chemical engineers—was the obvious choice, and luckily for me, I already knew two chemical engineering professors working on control theory. After two years working with Drs. Jim McLellan and Martin Guay on extremum seeking control, writing lots of MATLAB code, and taking a few

additional math and robotics courses, I hoped I had a strong enough background that someone would let me start a PhD in robotics.

Robotics was a largely foreign domain to me. My friends were mostly classmates from chemical engineering, people I played sports with, and other writers, cartoonists, and graphic designers from the satire newspaper I volunteered at. None of them did robotics. I knew lots of professors, but most of them were chemical engineers, mathematicians, or chemists. The vast majority of their contacts were also involved in the chemical industry that I was trying to get away from. Without any real contacts to the robotics community, I had to find a school, advisor, and topic on my own.

It was intimidating! I didn't really know what I wanted to do beyond just "robotics" and that's way too broad of a topic for a PhD. First, I narrowed down where I wanted to go. It would have been easy to stay in Canada, but I wanted the experience of living in another country with a different culture, where I would hopefully meet lots of interesting people different from myself and expand my perspective on life.

My girlfriend, now wife, had started her PhD in Manchester, UK while I was finishing the last year of my masters so naturally I began thinking of doing my PhD in the UK as well. However, after visiting the UK for the first, I realized the country is much grayer, wetter, and less friendly than Canada, and I had serious second thoughts about moving there. For a few months, I searched for PhDs elsewhere, and found a potential advisor in Australia who looked like a good fit for me. Australia is in the southern hemisphere though and like its seasons, its academic calendar is offset by 6 months. Not wanting to wait 6 months between degrees, I decided that I would apply in Europe after all. And if I was going to be in Europe, I might as well be near my girlfriend.

So I searched online for robotics programs in the UK. I quickly discovered that the University of Sheffield has an entire department of *Automatic Control*

& Systems Engineering and Sheffield is only an hour by train from Manchester where my girlfriend was! It seemed like the perfect place to go. I still didn't really know what exactly I wanted to do within robotics, but it was easy enough to read the biographies of the professors in Sheffield and see whose research sounded interesting.

Of all the professors, one in particular stood out: Dr. Andreas Kolling. Just like me he had done his undergrad in mathematics and switched to robotics during grad school. He was also working on multirobot systems, which I was interested in because I had worked on network control systems in during masters and already knew a few things about graph theory and consensus algorithms. So I sent him an email explaining my background and research interests and attached my transcripts and CV. He responded quickly and wanted to set up a video call.

I still remember that first call I had with Andreas. We discussed a shortest path planning algorithm on a graph. Having never encountered this problem before, I reasoned that this problem's difficulty probably scaled exponentially with the size of the graph. After all, the number of paths scales exponentially, so it must require a similar effort to find the shortest one, right? Any roboticist would say "Of course not!" as Dijkstra's algorithm and A* can both solve it in at most quadratic time. Andreas explained this fact, which at the time was quite incredible to me! As much as I was embarrassed at not figuring out the solution on my own, Andreas didn't seem to mind and somehow I had made a good enough impression on him that he agreed to advise me and help with my scholarship application.

Part of the scholarship application was a research proposal. I had written some research proposals before as a masters student, but they were all based on previous proposals that my advisors had written. This time, I had no starting point; Andreas wanted me to write it on my own. I took the easy way out and based the first draft heavily on a few paragraphs from his webpage about planning in unstructured environments. I think I only spent two hours writing it and the

main citation was to a video on Kiva systems—a system of autonomous robots which reconfigures shelves in Amazon's warehouses navigating using a system of lines on the floor. I sent Andreas the draft, and he gave me the harsh criticism that I ultimately needed. My proposal was crap and I needed to put a lot more effort in if I wanted to get a scholarship. I spent the next week researching and writing a proper proposal with a variety of citations. A proposal that I could actually be proud of. That proposal ended up earning me the scholarship I needed to be able to go to Sheffield.

After accepting the scholarship and starting my UK visa application, I got some news from Andreas. He was leaving the University. He had accepted a full time position at iRobot in Pasadena, California and would be leaving the UK a month before I was to arrive. I was pissed. How could this happen to me? I had my PhD all sorted out, but now it wasn't going to work out. Not without an advisor. Maybe I should turn down the scholarship, wait 6 months, and go to Australia after all.

A day later, I had calmed down. Further down in Andreas' email telling me about his new job, he said "I will still advise you over Skype, or in person when I'm in Europe or you're in California". But I had no plans to be in California. Was he suggesting that I could go work with him in California? Maybe this could actually be a good thing for me. Then I looked up up iRobot, the company he was leaving Sheffield for. I read things like "world's largest robotics company", "makers of the Roomba robotic vacuum", and "over $600 million in revenue". This was definitely starting to look like a good thing for me. I decided to the PhD anyways, and I was right about going to work with Andreas in California. I ended up going there as intern 3 times during my PhD and spent about 1/3 of my PhD at iRobot. These internships complemented my work at the University really well, exposing me to the robotics industry and a real robotic platform, while still allowing me to return to the UK and academia to focus on research.

Acknowledgments

I would first like to thank my advisor, Andreas Kolling. When I first met him, I had no experience in robotics, yet he was eager to work with me and help me make the transition into the field. He left academia shortly before I started my PhD, yet he honoured his commitment to advising me, and was able to get me three internships at iRobot which have proved invaluable for me, both academically and professionally. Over the years, Andreas and I have had many interesting and productive conversations about robotics, mathematics, and life in general.

Secondly, I'd like to thank my other advisor, Roderich Groß who stepped up when Andreas left and became my advisor at the university. He has welcomed me into his research group and met me with me regularly to work out the details of my research. His skills have complemented Andreas' very well and he has provided me lots of valuable perspectives on my work.

In addition to my advisors, I would also like to thank Dr. Paul Trodden and Dr. Francesco Amigoni for taking the time to thoroughly read this thesis and provide me with valuable feedback for improving it.

In the UK, my colleagues at the University of Sheffield have made the PhD experience more enjoyable, with many interesting discussions in the office, at group meetings, at lunch, and outside of the university in the evenings. In particular, I would like to thank Buket Sonbas and Leonardo Stella for their friendship and the many good times I've shared with them over the years.

On the other side of the Atlantic, I am grateful the opportunity to intern

at iRobot where I learned to work in a large code base, and use Ubuntu, vim, git, C++, and python which have made me much more efficient as a software developer and robotics researcher. In addition to Andreas, I'd like to thank Vazgen Karapetyan, Dhiraj Goel, Darren Earl, Manju Narayana, and Martin Llofriu for helping me learn to develop on real robots, navigate iRobot's code base, and use their software tools. Their help and patience helped me go from a clueless intern to someone who is confident and can contribute. Lastly I'd like to thank the many other interns for all the times we've enjoyed together, trying new restaurants in Pasadena, playing soccer, drinking maté, rock climbing, and learning about each other's culture.

Prior to my PhD, many people have encouraged and challenged me, ultimately giving me the skills and confidence needed to approach a robotics PhD. From a young age, my parents have always praised my curiosity, encouraged my interests in math and science, and given me the resources I needed to succeed. In high school, Jim Chapman and Rick Guetter devoted many hours to enrichment math classes which helped me prepare for math contests and be excited about the subject. At Queen's University, many professors across the Mathematics, Chemical Engineering, and Chemistry departments—Bahman Gharesifard, Andrew Lewis, Mike Roth, Noriko Yui, Xiang Li, Michael Cunningham, Scott Parent, Aris Docoslis, Dave Mody, Bill Newstead, and Nick Mosey—taught me the beauty of mathematics and the natural world and inspired me to continue on in academia. My two masters advisors, Martin Guay and Jim McLellan, in particular, spent countless hours teaching me to be an effective researcher and be excited about applied mathematics.

Finally, I would like to thank my wife, Hattie Xu, for her unwavering love and support despite the many months of long distance. She has helped me stay focused as we both write our PhDs and is a daily source of joy.

Contents

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

Autonomous **A**gents and **M**ulti-**A**gent **S**ystems . . . . . AAMAS
Average **P**artition **A**lgorithm . . . . . APA
Average **P**artition **P**roblem . . . . . APP
EXPonential time . . . . . EXP
Global **P**ositioning **S**ystem . . . . . GPS
Hidden **M**arkov **M**odel . . . . . HMM
Hierarchical **M**arket-based **S**olution . . . . . HMS
Interntational **C**onference on **R**obotics and **A**utomation . . . . . ICRA
Integer **L**inear **P**rogramming . . . . . ILP
Internation conference on **I**ntelligent **RO**bots and **S**ystems . . . . . IROS
Invasive **W**eed **O**ptimization . . . . . IWO
Lin–**K**ernighan (heuristic for the TSP) . . . . . LK
Lin–**K**ernighan-**H**elsgaun (TSP solver) . . . . . LKH
Memetic **A**lgorithm . . . . . MA
Minimum **P**erfect **M**atching . . . . . MPM
Minimum **P**artition **A**lgorithm . . . . . MPA
Minimum **P**artition **P**roblem . . . . . MPP
Minimum **S**panning **T**ree . . . . . MST
multiple **T**ravelling **S**alesperson **P**roblem . . . . . m-TSP
Non-deterministic **P**olynomial time . . . . . NP
Number **P**artition **P**roblem . . . . . NPP
Printed **C**ircuit**B**oard with **1173** vertices (TSP problem) . . . . . `pcb1173`
Polynomial time . . . . . P
Probabilistic **R**oad **M**aps . . . . . PRM
Rapidly-exploring **R**andom **T**ree . . . . . RRT
Rapidly-exploring **R**andom **T**ree (asymptotically optimal variant) . . . RRT*
Simultaneous **L**ocalization **A**nd **M**apping . . . . . SLAM
Travelling **S**alesperson **P**roblem . . . . . TSP
Travelling **S**alesperson **P**roblem **LIB**rary . . . . . TSPLIB
Unmanned **A**erial **V**ehicle . . . . . UAV
3-SATisfiability problem 3-SAT

List of Symbols

Probability . . . . . $\mathbb{P}[\cdot]$
Real numbers . . . . . $\mathbb{R}$
Circle . . . . . $\mathbb{S}^1$

Set of cycles or paths . . . . . $\mathcal{C}$
Edge set . . . . . $\mathcal{E}$
Graph (weighted and undirected) . . . . . $\mathcal{G}$
Checkerboard partition (set of rectangles) . . . . . $\mathcal{H}$
Information known by robot i . . . . . $\mathcal{I}_i$
Set of start/endpoint pairs . . . . . $\mathcal{L}$
Complexity Class . . . . . $\mathcal{O}$
Partition of a graph into subgraphs . . . . . $\mathcal{P}$
Environment . . . . . $\mathcal{Q}$
Set of coverage ranks . . . . . $\mathcal{R}$
Search path planning tree . . . . . $\mathcal{T}$
Set of outlier vertices . . . . . $\mathcal{U}$
Vertex set . . . . . $\mathcal{V}$
Set of hidden states (in an HMM) . . . . . $\mathcal{X}$
Cellular decomposition of environment . . . . . $\mathcal{Y}$

Markov transition matrix . . . . . $\boldsymbol{A}$
Communication matrix . . . . . $\boldsymbol{C}$
Identity matrix . . . . . $\boldsymbol{I}$
Projection matrix from state- to cell-belief . . . . . $\boldsymbol{P}$
Transformation matrix between cellular decompositions 0 and 1 . . . . . $\boldsymbol{T}_0^1$

Covector of transition probabilities to cell y_j . . . . . $\boldsymbol{a}_j$
Belief vector . . . . . $\boldsymbol{b}$
Communication covector associated with cell y_j . . . . . $\boldsymbol{c}_j$
Projection covector associate with cell y_j . . . . . $\boldsymbol{p}_j$
Hidden state belief vector . . . . . $\boldsymbol{w}$
Hidden state belief vector (not normalized) . . . . . $\widehat{\boldsymbol{w}}$
Observation vector . . . . . $\boldsymbol{z}$

Vector of ones . . . . . $\mathbf{1}$

Area . . . . . A

Confidence limit at significance level α . . . . . B_α^-, B_α^+
Communication strength between two locations . . . . . $C(\cdot,\cdot)$
Cost of a partition or m-TSP solution C
Decile . . . . . D
Search reward function . . . . . J
Size of a subgraph . . . . . S
Path length . . . . . T
Sum of edge weights of a graph . . . . . $W(\cdot)$
Sum of edge weights between a graph and vertex . . . . . $\Delta W(\cdot,\cdot)$

Transition probability from cell y_i to cell y_j . . . . . a_j^i
Belief that robot is in cell y_i . . . . . b_i
Cycle (closed path) c
Distance . . . . . d
Edge of a graph or polygon . . . . . e
Monotonically increasing function . . . . . f
Rectangle in a checkerboard partition . . . . . h
Index i
Alternate index . . . . . j
Constant . . . . . k
Length . . . . . ℓ
Number of robots . . . . . m
Number of vertices, elements in set, etc... . . . . n
Path . . . . . p
Probability that robot in state x_i is in cell y_j . . . . . p_j^i
Location in the robot's environment . . . . . q
Rank . . . . . r
Velocity . . . . . s
Similarity between cells i and j of different decompositions . . . . . t_i^j
Time . . . . . t
Time step . . . . . Δt
Vertex of a graph or polygon v
Edge weight function for a graph . . . . . w
Belief that target is in hidden state x_i . . . . . w_i
Hidden state of HMM . . . . . x_i
Cell of decomposition . . . . . y_i
Probability of observation if target is in cell y_j . . . . . z_j

Significance level (typically 2.5%) . . . . . α
Discount factor for search reward . . . . . β
Convex coefficient . . . . . γ
Small strictly positive number . . . . . δ
Small angle . . . . . ϵ
Normalization factor . . . . . η
Angle, direction, or orientation . . . . . θ
Probability measure . . . . . μ

Zero-mean noise . . . . . ν
Discrete time index . . . . . τ
Marginal search probability . . . . . $\Delta\phi$
Total search probability . . . . . ϕ
Boolean check variable for transfers or swaps . . . . . $\chi_{i,j}$
Threshold for transferring outliers . . . . . ω

Difference . . . . . Δ
Set of possible orientations . . . . . Θ

Boundary . . . . . ∂
Elementwise product . . . . . $\odot$
Vector of ones . . . . . $\mathbf{1}$

Chapter 1

Introduction

Robots are becoming ubiquitous in society. Many industries—manufacturing, agriculture, mining, and logistics—depend heavily on the automation of various tasks performed by specialized robots. Consumer robotics is taking off too as robotic vacuums, mops, and lawnmowers are quickly being integrated into the smart home. Autonomous vehicles will be the next wave of consumer robots and when these highly anticipated robots arrive in the next few years, they will revolutionize transportation industries. Through these technologies, billions of people will be interacting with robots on a daily basis all over the world.

As the world is rapidly roboticized, many robots will be operating in the same spaces and will often need to work together towards some common objective. They will need to form cooperative teams and efficiently divide and complete tasks. In the dynamic, unpredictable environment of the real world, robots also must be able to communicate with their teammates to coordinate behavior and share new information. The needs to communicate and divide tasks amongst the team are unique challenges for multirobot systems which do not exist for a single robot working alone.

1.1 The value of planning

Initial forms of automation—large machines used in manufacturing—perform the exact same task thousands of times. For such repetitive tasks, the machine's

behavior only needs to be planned once. This single behavior can then be hard-coded and the machine will behave in an identical way every single time. This approach has been wildly successful and manufacturing plants around the world are now full of machines important for everything from chopping vegetables, to sewing clothing, to making automobiles. Despite its success, using a fixed plan is inflexible. Machines used in industrial automation are unable to adapt to changes in their surroundings and they can only be used in one specific environment which was designed for that machine and typically does not contain humans.

As robots become more ubiquitous, they are no longer constrained to specialized factories and humans are increasingly welcoming them onto our roads and into our houses. In these environments, dynamic planning is essential. Two identical robots sold to different consumers can operate in two very different environments as everyone's home is unique. Even from day-to-day, a single robot's environment can change drastically, such as when people rearrange furniture, close and lock a door, or simply make a mess. As the same robot is expected to work in different environments, it cannot simply use one plan and must instead plan based on its current environment.

Planning in a dynamic or unknown environment cannot be done in advance. Robots must instead do their own planning as they are operating. Fortunately, hardware costs have steadily decreased as robots have become more common. Many consumer robots now have enough computing power to do some online planning using various planning algorithms. For simple tasks, such as navigating between two points, the robotics community has already developed adequate algorithms. For more complex tasks, where robots must go to many locations and respond to external stimuli, planning problems remain unsolved.

1.2 Planning for a team

Planning is also closely related to coordination of a team. As robots become cheaper and more readily available, consumers will expect the robots to be able to coordinate with each other in the same way that we expect any new electronic device to be compatible with our old devices. The main aspect of planning—planning how a single robot will complete a specific task—is identical whether there is only one robot or multiple robots completing separate tasks simultaneously. However, there are additionally two aspects of planning related to the fact that there are multiple robots:

1. Planning which tasks each robot is responsible for; and

2. Planning how the robots will communicate when they need to replan or share information.

These problems of task allocation and communication are unique to multirobot systems.

Communication is particularly important for teams of robots, as robots can only coordinate their behavior if they can communicate. Many consumer robots use inexpensive parts and cannot communicate reliably over large distance or through walls. These communication constraints force them to spend time searching for each other or travelling to a planned rendezvous, creating a fundamental trade-off for teams of robots. Communication helps them share information and make plans together, but it can also slow the team down if they spend too much time communicating instead of working on their actual objective. How much the robots end up communicating depends heavily on how much effort it takes for them to communicate, what other objectives they have, and how much reality differs from the model they used when planning. By communicating just the right amount, the team can complete complex tasks much faster than a single robot,

despite poor knowledge of their environment and limitations in their ability to communicate.

1.3 Robotic coverage

Consumer robotics has become a large market in recent years and the most successful category of consumer robots are *coverage robots*, such as vacuum cleaners and lawnmowers (Figure 1.1). These robots perform cleaning or mowing tasks where the robot has to cover a large environment, such as a room or a yard, by passing its cleaning or cutting tool over every square inch of the environment. Despite differences in their tools and environments, all of these robots have essentially the same behavior and the same planning algorithm could easily be used for many different kinds of coverage robots. Due to their reasonable price points and effectiveness at performing repetitive chores, the market for coverage robots has grown steadily and now accounts for over $6.3 billion dollars in revenue, approximately 60% of the entire consumer robotics industry [164].

Planning helps make coverage robots more efficient, however, it often comes as an afterthought and in some cases, a robot doesn't plan at all! The first robotic vacuum cleaners had very limited processing power and could not plan coverage paths. Instead, they would follow simple preprogrammed behaviors such as spiraling or random bouncing [91] (Figure 1.2). These fixed behaviors are not very robust and the vacuum cleaner can take a long time to fully clean a simple, square room and may never reach certain parts of a more complex environment. Robotic lawnmowers, on the other hand, typically require a boundary wire or a sequence of boundary posts, and the robot's behavior is based on the location of these markers [152] (Figure 1.3). This method can work well for simple environments like lawns, but it requires a technician to physically modify the robot's environment by installing the boundary markers. The newest models of coverage robots have

Figure 1.1: Examples of five commercially available coverage robots: iRobot Roomba s9 vacuum cleaner [89] (top left); Robomow RS635 lawn mower [160] (top right); Maytronics Dolphin S300i pool cleaner [132] (bottom left); Ecovacs winbot X window cleaner [56] (bottom center); and iRobot Braava m6 mop [88] (bottom right).

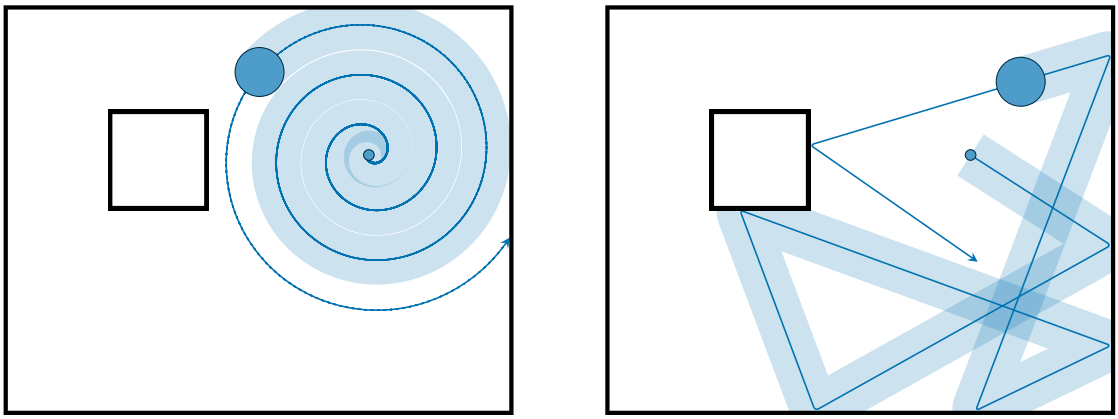


Figure 1.2: Two basic coverage behaviors for a robotic vacuum cleaner are spiraling (left) and randomly bouncing (right).

the processing power and mapping capability needed to plan efficient coverage plans without needing to modify the environment to suit the robot, yet many still use inefficient coverage methods based on ad-hoc short-term planning [70].

The behavior of today's most popular coverage robots can be drastically improved with more intelligent coverage planning. Improved planning includes the

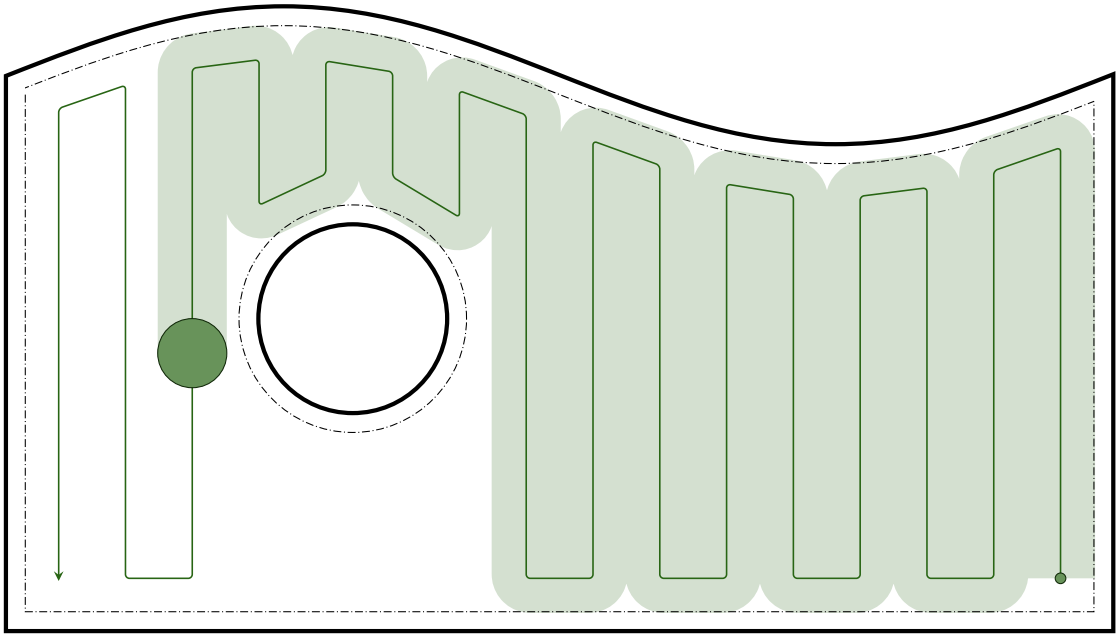


Figure 1.3: Coverage strategy for a robotic lawnmower using a back-and-forth motion based on a guide wire which is installed in advance.

obvious—making the path as short as possible with minimal repeat coverage—but must also result in a robot which is reliable and adaptable to all the possible environments it may be required to cover. Minimizing the number of turns has numerous benefits in coverage even if it requires the robot to follow a slightly longer coverage path. Turns take time so a longer path with fewer turns is faster than a shorter path with more turns. Robots are also much more likely to get stuck or damaged when turning—turns usually occur near walls or obstacles—so minimizing turns minimizes these risks, making the robot more reliable. In many applications, turns also result in worse performance. A painting robot deposits paint in a more uniform layer when travelling straight than when turning; distance data recorded by a mapping robotic boat or UAV's lidar sensor is not useful if the robot tilts during a turn; a robotic lawnmower may occasionally damage flowers in a garden near the edge of the lawn when it turns. Qualitatively better coverage, due to straighter paths with fewer turns, is also achieved by more intelligent coverage planning.

Reliability and adaptability also both depend on the quality of the map used by the coverage planner. An intelligent planner should be aware of the capabilities of the robot—how close it can get to the wall, what kinds of objects it can easily navigate around, how much space it needs to turn—and use its knowledge of these behaviors to filter the map based on the coverage behavior. Effective replanning to adapt to changes in the environment without repeating previous coverage is also essential to the kind of reliable coverage tomorrow's consumers will expect from their robots.

1.4 Coverage for humans

In early 2015, before I considered working in robotics, I decided I wanted to become a runner. But when I started running, I couldn't stand it. It was boring and exhausting. For a few months, I was very inconsistent, running approximately once a week and making excuses every day I didn't go for a run. I needed a better way to motivate myself. My big idea: I wanted to run on every street in Kingston, the city I was living in at the time. This challenge would give me a real goal when running, and would also help me get to know the city that I had been living in for several years better.

As a first step, I went to a local gas station and bought a map. A paper map. When I went home, I marked off the route that I ran the last time I went running. Then, every day after I went running, I would mark off where I ran. I quickly discovered that I was only filling up certain parts of the map. Within a few weeks I had been on every major street, all the streets at the university, and a few parks near my house, but hadn't visited most residential streets, industrial areas, the nearby military base, or any places more than 3 km away. I realized that if I wanted to finish this project, I would have to plan where I would run.

I now had an interesting challenge which gave me something to think about

when I was running: How do I maximize the number of new streets I run on while minimizing the time I spend getting to those streets?

This project was my first introduction to coverage planning. I had a coverage region—the streets of Kingston—and I needed to plan several paths that visited every location. This problem also had two main constraints:

1. Every path must start and end at my house; and

2. The paths must be at least 5 km long, so I would get enough exercise for the day, but shouldn't be so long that I would get too exhausted to run all the way back home.

Although I had no experience in path planning, I quickly developed a system. Before my run, I would choose a *target region* on the map where I wanted to run. Usually there would be lots of unvisited streets in this region and a few on the way to it. Initially, I would take a route to my target region along many unvisited streets, even if it was longer than the direct route. However, I soon realized that the direct route is better. Unvisited streets on the way to the target region were always closer to my house than unvisited streets in the target region. Since I can only run so far before getting tired and far away streets take longer to get to, the further away a street is, the more *valuable* it is. Although this rule holds in general, there is one place where it doesn't: *main streets*. Regardless of where I was going, there were a few streets that were usually part of the shortest route to my target region. I had already run on the main streets near my house dozens of times, but there were plenty of streets further away that I hadn't been to yet, but I knew I would eventually run on when going somewhere even further away. Since I'd be running on them many times eventually, they provide *less* value than the other nearby streets. My resulting mental heuristic (Figure 1.4) provided me a useful way to evaluate various running routes based on their value in helping me achieve my goal of running on every street. Before going out for a run, I could

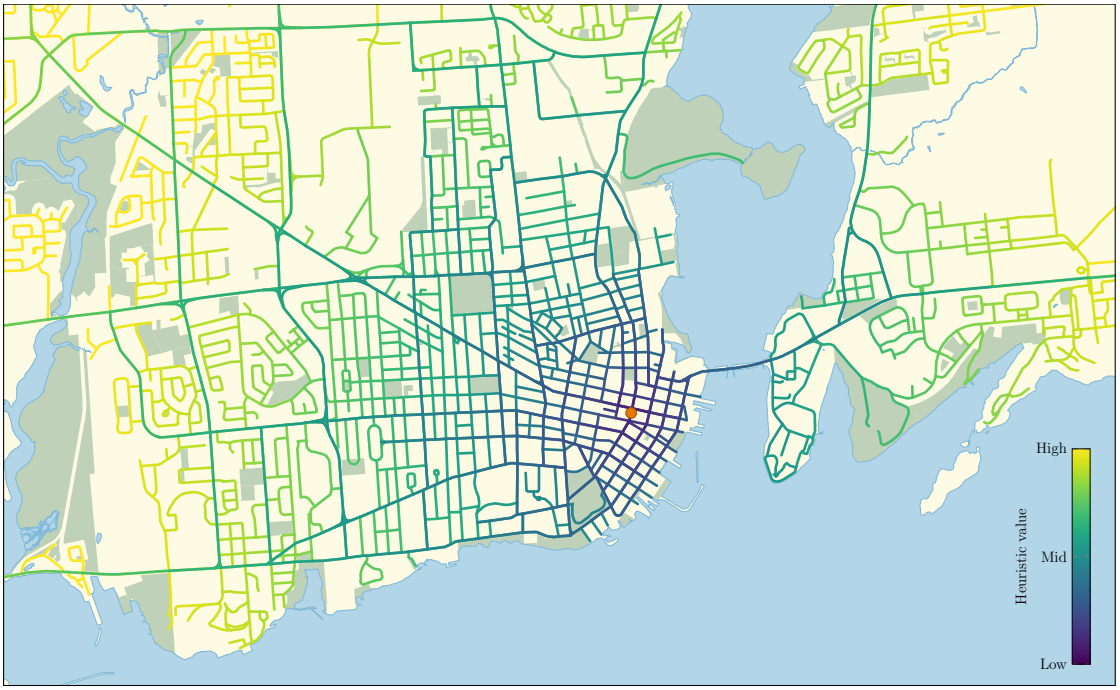


Figure 1.4: Heuristic values of streets in Kingston, Ontario used when planning running routes. The heuristic value increases with distance from my house (orange dot), but main streets have lower value than other streets nearby.

quickly glance at my map of where I'd already been and find a set of unvisited streets which maximize the heuristic value based on the length of run I wanted that day.

When I arrived at my target region, I had to plan the best coverage path for that region, often a small neighborhood in the suburbs. Canadian suburbs are often designed using a large grid for major roads, with lots of curved roads and cul-de-sacs making up the residential area in between. Finding the shortest path that goes on all of the roads of one of these subdivisions is not trivial! Fortunately, I had lots of time to think while running, so I could mentally plan the best path to lots of unvisited streets (Figure 1.5). The two criteria I would use to decide between possible paths were:

1. Only running on the same street twice when absolutely necessary; and

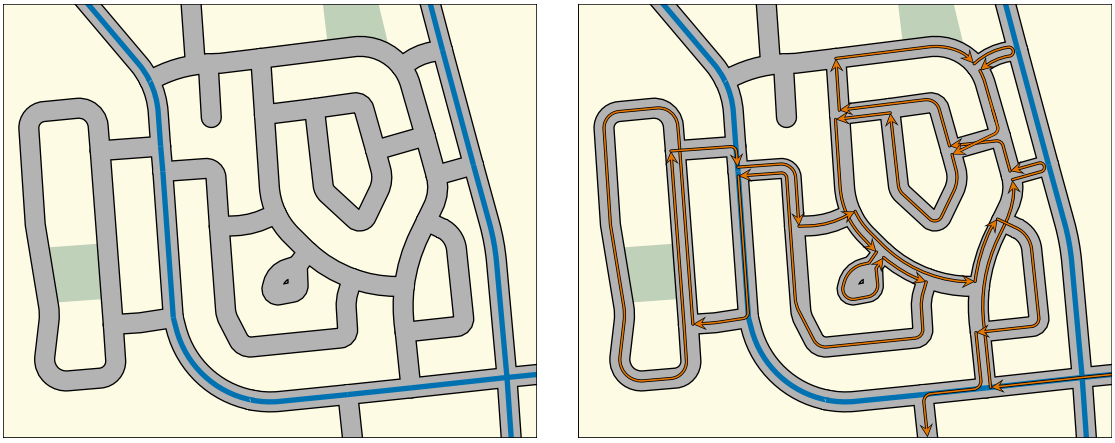


Figure 1.5: Efficient route to run through a neighborhood. As I had run on a few streets (left), this route covers most of the remaining streets while trying to avoid running on the same street twice or turning 180°.

2. Preferring 90° turns and avoiding 180° turns which require me to stop and lose all my momentum.

Usually by the time I arrived at my target region, I had mentally planned a good path with minimal repeat coverage and with few sharp turns. The more I ran and planned efficient routes, the more my intuition improved and I could easily plan a near-optimal route even though I did not use a real algorithm.

Little did I know, but this project ended up being really similar to my PhD work. I was using heuristics to plan good paths. Constraints of running a similar distance each day is remarkably similar to multirobot planning where each robot does the same amount of work. I even cared about minimizing turns long before I thought about doing that for a robot!

1.5 Objectives & contributions

The main objective of my thesis is:

> *To develop a theoretical understanding of planning problems faced for multirobot teams in realistic environments—both in general and while performing coverage—and design practical algorithms to solve these multirobot problems.*

The main problems considered in this thesis are task allocation and search, which are general problems unique to multirobot teams, as well as coverage planning, a specific problem faced by today's largest category of consumer robots. Although the work that I present is largely theoretical, the problems are based on the challenges faced by real robots in real environments and thus depends on the robot's hardware and the many uncertainties related to working in different environments shared by humans.

The contributions towards this main objective include: a novel relationship between two different cost functions which is exploited to develop an efficient multirobot task allocation algorithms (Chapter 3); a new coverage planning approach based on a one-dimensional rank decomposition (Chapter 4) which is computationally efficient and minimizes the number of turns made by the robot, in contrast with existing approximate (zero-dimensional) and exact (two-dimensional) decompositions; an extension of the successful rapidly-exploring random trees (RRT) algorithm to a multirobot search problem based on maximizing an infinite-horizon reward function (Chapter 5) instead of minimizing path length; and a description of many of the practical problems I encountered and solved while working to implement multirobot coverage on real coverage robots, the iRobot Roomba (Chapter 6). These contributions have been published in the following publications:

- [186] I. Vandermeulen, R. Groß, and A. Kolling, "Re-establishing communication in teams of mobile robots," in *International Conference on Intelligent Robots and Systems (IROS)*, pp. 7947–7954, 2018.

- [187] I. Vandermeulen, R. Groß, and A. Kolling, "Balanced task allocation by partitioning the multiple traveling salesperson problem," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 2019, pp. 1479–1487.

- [188] I. Vandermeulen, R. Groß, and A. Kolling, "Turn-minimizing multi-robot coverage," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 1014–1020.

- [108] A. Kolling and I. Vandermeulen, "Turn-minimizing or turn-reducing robot coverage," Mar. 19 2020, US Patent App. 16/565,721.

- [189] I. Vandermeulen, R. Groß, and A. Kolling, "Sampling based search for a semi-cooperative target," *under review for publication in International Conference on Intelligent Robots and Systems (IROS)*, 2020.

1.6 Overview of thesis

The remainder of my thesis consists of six additional chapters:

- **Chapter 2** contains the relevant background information, including the state-of-the-art related work. I begin with a brief discussion of computational complexity (Section 2.1) which is essential to the analysis of the numerous algorithms presented in this thesis. Next, I discuss basic planning problems (Section 2.2) which my algorithms often rely on. Existing algorithms for solving navigation-to-a-point problems are presented in Appendix B and rely on visibility graph algorithms from Appendix A. Existing travelling

salesperson algorithms are shown in Appendix D and are based on minimum spanning tree algorithms in Appendix C. The background chapter continues with a discussion of communication for multirobot teams (Section 2.3). This discussion includes realistic models of wireless communication and explanations of various communication strategies that can be used by multirobot teams. I conclude the chapter with a description of existing robotic coverage algorithms (Section 2.4).

- **Chapter 3** focuses on the problem of dividing tasks within a team of robots which is central to cooperation within a team of robots. Although different robots can fill many different roles in society, and their behaviour is equally varied, their work can often be broken down into a set of smaller tasks. A team of robots must divide these tasks amongst individual robots, based on the locations of the tasks, the time needed to complete each task, and the robot's abilities. The goal of task allocation should be *balancing* the workload so that the team finishes as quickly as possible and no robot sits idly while other robots still have several tasks left to complete. This problem (Problem 3.1) is equivalent to the minmax multiple travelling salesperson problem (Problem 3.2), which combines task allocation and routing. Although this problem is NP-hard, its cost function has an approximately monotonic relationship with another cost function that is easier to evaluate (Subsection 3.3.1). By exploiting this relationship, I developed a heuristic algorithm (Algorithm 3.3) which partitions the set of tasks, approximately solving the task allocation problem. The solution to the combined routing and allocation problem (Algorithm 3.4) is based on solving the travelling salesperson problem on each set of tasks produced by the allocation algorithm. This heuristic runs quickly, can be decentralized (Section 3.7), is

compatible with constraints on where the robots must start or end (Section 3.8), and has produced better quality solutions than other state-of-the-art approaches (Section 3.9).

- **Chapter 4** contains my turn-minimizing coverage algorithm. Coverage is an example of a complex robotics problem which can be divided into smaller tasks. It is typically solved by dividing a coverage region into small grid cells, equal in size to the robot's footprint, or large regions, akin to the rooms of a house. My coverage strategy, on the other hand, first divides the coverage region into *ranks* which are long thin rectangles as wide as the robot but much longer than it (Section 4.2). These ranks are constructed to minimize the number of turns the robot will need to make. Ranks are suitable component tasks for dividing the coverage mission amongst a team of robots—small enough to divide evenly amongst many robots, yet large enough that assigning them takes relatively little computational effort. These tasks are then assigned to robots and converted into coverage paths using my algorithm from Chapter 3 with a few modifications to account for the fact that the robot will start and end each task in different locations (Section 4.3). I validated this strategy by computing coverage plans for 25 test environments that had been mapped experimentally by an iRobot Roomba$^{\text{TM}}$ robotic vacuum (Section 4.4). Based on these plans, I found that my approach reduced the average number of turns required by approximately 7% and resulted in coverage time decreasing by a factor of approximately $1/m$ when m robots are used instead of 1 robot.

- **Chapter 5** presents a method for tracking and searching for teammates that a robot cannot communicate with. Reducing the cost of robots often involves using inexpensive communication devices and so robots cannot necessarily communicate over long distances or through walls. As teams are

often most productive when the robots spread out to complete different tasks simultaneously, it is then impossible for robots to be constantly connected. Search is a flexible way to re-establish communication when a team gets separated because it does not require prior planning. In cooperative search, robots often have lots of information on how their target—another robot in the same team—is likely to behave. My search strategy uses information, such as historic data of target behavior, to maintain a probabilistic belief of the locations of all the robots in the team that it cannot communicate with (Section 5.3). The belief also incorporates both positive and negative observations of the target robot (Section 5.4) and can be combined with another searcher's belief of the same target (Section 5.5). Using the belief, a target can evaluate various potential search paths using a discounted reward function which rewards paths which find the target quickly (Section 5.6). I designed a planner which constructs a tree of possible search paths by adding new vertices based on random samples and uses reward bounds to remove old vertices that are guaranteed to not be part of the best path (Section 5.7). This planner proved to be quite effective, finding the target slightly quicker on average than two baseline strategies, while drastically decreasing the time needed to find the target in the most difficult 30% of searches (Section 5.8).

- **Chapter 6** covers some of the practical details of implementing single and multirobot coverage on real robots. Much of this work is based on my experience interning at iRobot, the makers of the commercially successful Roomba robotic vacuum cleaner. As many things can go wrong during a coverage mission—human interference, damage to a robot's tool, needing to recharge, or differences between the robot's map and the real environment—real coverage requires robots to adapt to these circumstances in real time. Three main ways of making coverage more robust are through semantic commands,

replanning, and search. Semantic commands (Section 6.3) are used to ensure robots actually achieves a meaningful outcome rather than simply trying to follow a precise description of how to achieve that outcome. Replanning (Section 6.5) enables the robot to plan partial coverage paths when a robot's location changes suddenly (a human moved it) or its map changes (a door was opened) while not repeating coverage of places the robot already covered. Search (Section 6.8) is vital for coordinating robots after they get separated, which is especially useful near the end of a mission to help rebalance the workload. I tested my ideas on robust coverage using a combination of simulation and real world experiments using the iRobot Roomba (Subsections 6.6.1, 6.7.1, and 6.9.1). The results of these experiments showed that robust strategies using on the planned coverage paths from Chapter 4 and search strategy from Chapter 5 consistently perform better than basic strategies which use less planning.

- **Chapter 7** concludes my thesis with some highlights of the main ideas and results of the previous chapters.

Chapter 2

Background

Robots are machines which can think and move. Their mobility sets them apart from other computers and lets them change the very environment they inhabit. Their intelligence sets them apart from many other machines which rely on human operators: robots can operate autonomously, planning their actions and responding to external stimuli. Although robots perform computations in much the same way that all computers do, there is a fundamental difference in the *types* of problems solved by robots. Robotic problems involve interactions between the robot's hardware and its environment, and in some way are related to the robot's motion.

In this chapter, I will briefly introduce some basic problems in robotics. The background material presented in this chapter are directly related to the main objectives of this thesis (Section 1.5) which include theoretical descriptions of and algorithms to solve planning and coverage problems. Computational complexity theory (Section 2.1) is a tool used to analyze problems and algorithms and provide a theoretical understanding an analysis of them. Path planning (Section 2.2) is the main kind of problem considered in my thesis and basic planning problems, such as navigating efficiently between two points, are the building blocks of more complex planning algorithms presented in my thesis. Although not all robotic planning problems involve *path* planning, I have focused on it as my research involves mobile robots most planning problems can be cast as path planning problems. Indeed, the seemingly unrelated problem of task allocation (Chapter 3) is equivalent to the multiple traveling salesperson problem (Subsection 2.2.3) which is a path planning

problem. In addition to path planning, communication (Section 2.3) is essential for multirobot cooperation but is often limited due to the robot's hardware or environment and search algorithms (Chapter 5) must understand these limitations in order to make up for them. Finally, coverage (Section 2.4) is a common task for consumer robots which involves some unique planning problems that are covered in this thesis (Chapter 4). The three main objectives of this chapter are thus:

1. Introduce several important planning problems in robotics;

2. Present some foundational algorithms which solve basic problems and are building blocks of solutions to larger problems; and

3. Survey of the literature on more complex problems—primarily coverage and search—which do not necessarily have a single solution that works for any scenario.

The material in this chapter closely reflects my own learning throughout my PhD. As someone coming from a different academic background, all of this material was necessary background information that I had to learn as a PhD student before I could really start doing novel work in robotics.

2.1 Computational complexity

As robots are increasingly being integrated into society, two important changes are happening:

1. Robots are getting cheaper, often relying on inexpensive processors and sensors without a lot of computing power; and

2. Robots are operating in larger, more complex, less predictable, and increasingly dynamic environments.

These two trends mean that robots are doing more with less. How are they getting out of this apparent paradox? Better, more *scalable* algorithms.

Computational complexity theory is a basic tool of computer science used to quantify the effort required by an algorithm. It classifies algorithms by how the computational effort scales with the size of the problem, making it extremely useful for comparing different algorithms that solve the same problem. If two algorithms solve the same problem, the better algorithm is the less complex one. In many cases, an algorithm that gives an approximate solution might even be more useful than a more complex algorithm that solves the problem exactly!

2.1.1 Big-$\mathcal{O}$ notation

At their most basic level, algorithms are ways to perform large computations by combining many small operations. A simple example is the long division algorithm taught to elementary school children. When performing this algorithm, the child repeatedly performs multiplications and subtractions to determine each digit of the result. Multiplications and subtractions are considered *easy* because students have typically already memorized addition and multiplication tables and so they can perform these computations instantaneously by recalling the product or sum from the table. As students do not usually memorize "division tables", division is considered more difficult. The long division algorithm enables students to compute a quotient, which would otherwise be difficult, by performing a series of easy computations.

Just as addition and multiplication are easy for children, there are certain operations which are easy for computers to compute. Alan Turing's computing machine [185] had five basic operations: read a symbol, write a symbol, erase a symbol, move left, and move right. Modern silicon-based computers contain several hard-coded electronic circuits for addition, multiplication, subtraction, and bitwise operations. These basic circuits are themselves created from a fixed number of logic

gates, made of transistors and diodes, which carry out the basic operations of a computer. Every more complex task performed by a computer can be reduced to a number of basic operations performed by these individual logic gates.

The complexity of an operation is defined by the number of basic operations needed to perform it. For example, computing the inner product of two n-dimensional vectors requires n multiplications and $n-1$ additions. Similarly, computing the product of two $n \times n$ matrices requires the computation of n^2 inner products—one for each element of the product matrix—and thus requires $n^2(2n-1)$ basic operations.

Big-$\mathcal{O}$ notation is a way to describe the complexity of performing an operation. Suppose an operation with input size n requires at most $kf(n)$ operations for some constant $k \in \mathbb{R}_{>0}$ and function $f(n)$. Then we can say that the operation is $\mathcal{O}(f(n))$. For example, computing inner products is $\mathcal{O}(n)$ as it requires $2n-1 \leq 2n$ operations. Similarly, matrix multiplication is $\mathcal{O}(n^3)$ as it requires $n^2(2n-1) \leq 2n^3$ operations. In both examples, we used simple functions for $f(n)$. Some common orders of complexity ranked from simplest to most complex are:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n\log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(\exp(n)) \subseteq \mathcal{O}(n!)$$

Algorithms with complexity closer to the left side of this hierarchy are more useful for solving large problems on inexpensive hardware, as is commonly required in robotics.

2.1.2 NP-hardness

We're often interested in finding the least complex algorithm which solves a given problem. Most problems can only be solved by algorithms with at least a certain order of complexity. This idea leads us to classify problems based on what classes of algorithms are required to solve them.

Three common complexity classes are [100]:

1. **P**, the set of all problems that can be solved in polynomial time;
2. **NP**, the set of all problems whose solutions can be checked in polynomial time; and
3. **EXP**, the set of all problems that can be solved in exponential time.

Determining which class a given problem belongs in is not always easy. If we know an algorithm that solves a problem, then we might be inclined to classify the problem based on the complexity of that algorithm, but what if there is a better algorithm? And what if we don't have an algorithm for solving the problem?

A clever way to analyze problems that we don't have an algorithm for is using *polynomial reductions* (Figure 2.1). A polynomial reduction is any algorithm which can, in polynomial time, convert any instance of one problem into another problem. We can then solve the original problem by reducing it to the second problem and then solving the second problem. Using this process, we have effectively found an algorithm for solving the original problem by way of the second problem. It leads us to two useful observations:

1. If the transformed problem is in P, then the original problem is also in P; and
2. If the original problem is not in P, then the transformed problem also must not be in P. Otherwise we could solve the original problem in polynomial time by reducing it to the transformed problem.

The second observation is the contrapositive of the first. In informal terms, these observations tell us that the original problem is no more difficult than the transformed problem.

The class NP can be defined in two equivalent ways. Previously, I defined it as the set of all problems whose solutions can be checked in polynomial time.

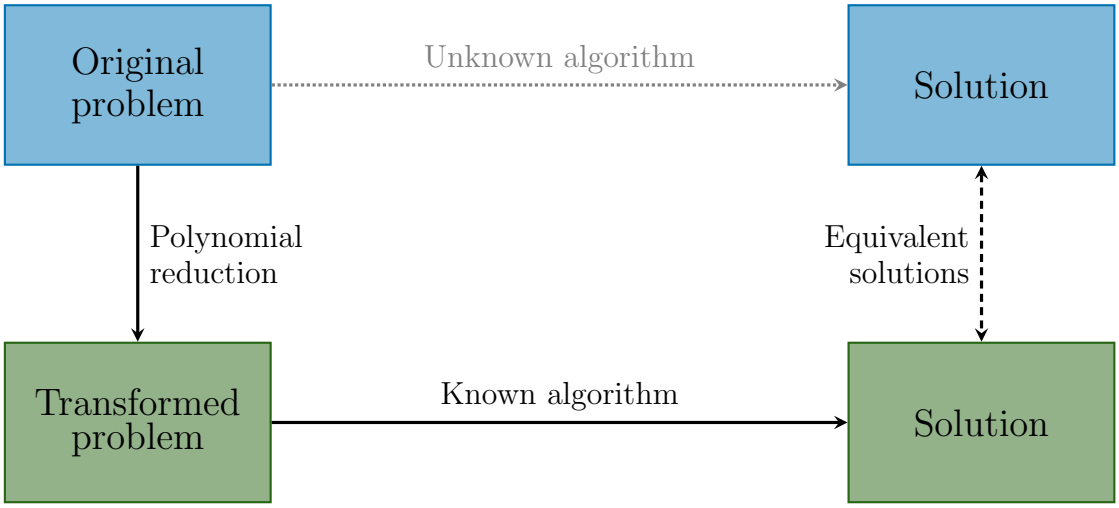


Figure 2.1: A polynomial reduction transforms one problem into another problem. If we can solve the transformed problem, then we can solve the original problem by transforming it and solving the transformed problem.

Alternatively, it is the set of all problems that can be solved by a *non-deterministic* Turing machine in polynomial time. The non-deterministic Turing machine is similar to an ordinary Turing machine, but it can choose between several possible behaviors in any situation and is assumed to choose the best one (i.e. the one that leads to a solution as quickly as possible). These definitions of NP are equivalent because the non-deterministic Turing machine could use a known solution as a "cheat sheet" when deciding which of its possible behaviors to choose and therefore solve the problem as quickly as a deterministic Turing machine checks the solution.

Another important class of problems is the NP-hard problems. A problem is NP-hard if every problem in NP can be reduced to it [107]. This definition seems like an impossible task! How could someone possibly find a reduction from *every* NP-hard problem? We don't even have a list of all NP-hard problems.

It turns out, that we only need to check a single problem: 3-SAT. An instance of 3-SAT consists of several logical clauses of the form $(x \vee y \vee z)$—dependent on boolean variables x, y, and z—which may share variables or their negations with other logical clauses of 3 variables. 3-SAT then asks: is there a value for each

variable which makes all clauses simultaneously true? In 1971, Stephen Cook [41] proved that the very behavior of any non-deterministic Turing machine can be encoded as an instance of 3-SAT. Any problem in NP can be solved by the operation of a non-deterministic Turing machine which can be reduced to an instance of 3-SAT, and so his result proves that every NP problem can be reduced to 3-SAT, which is therefore NP-hard. After Cook proved this now-famous result, many other problems have been proven to be NP-hard [66], usually by reducing them to 3-SAT. Lots of these problems have practical value to robotics.

Since not all NP-hard problems are themselves in NP, we also define the class of NP-complete problems which are both NP and NP-hard. NP-complete problems are interesting because they are in some sense the "hardest" NP problems. If a single NP-complete problem could be solved in polynomial time, we could use it to solve all the NP problems in polynomial time. Such a result would imply that P = NP, a proposition which has been asked for decades [16], and although widely believed to be false, has never been proven. Assuming that P $\neq$ NP, then any NP-complete problem is not in P and so it is a futile effort to search for a polynomial time algorithm to solve it.

Most problems encountered in robotics, and indeed in my thesis, are either in P or are NP-hard. Problems in P are usually solved using a known polynomial time algorithm that produces the problem's exact solutions. For NP-hard problems, on the other hand, any known algorithm requires more than polynomial time. These exact algorithms are rarely practical, and so NP-hard problems are typically solved using heuristics which produce approximate solutions in polynomial time.

2.2 Path planning

Motion is critical to everything that robots do. Mobile ground robots drive or walk in cluttered two dimensional environments. Underwater and aerial robots propel

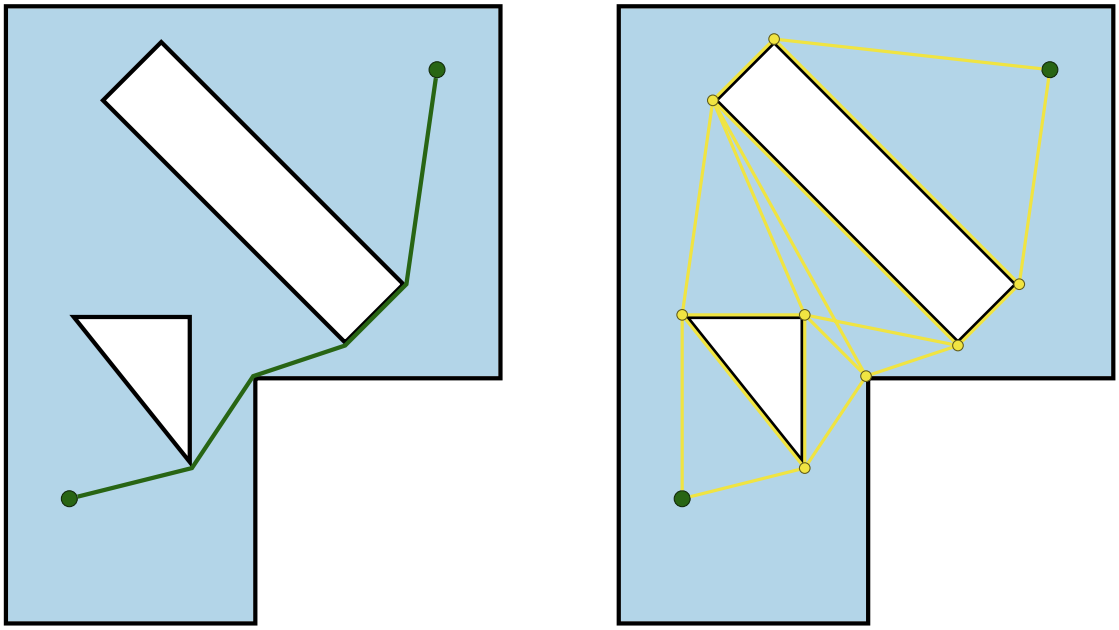


Figure 2.2: The shortest path between two points in a 2D environment (left). All of its edges are part of the visibility graph with the two points and all the concave corners as vertices (right).

themselves in three dimensional space. Often these motions also involve two or three dimensional rotations. Robotic arms, like human arms, typically have 7 degrees of freedom. Then if you add in the robot's dynamic constraints, and any obstacles in its environment, path planning can become quite a difficult task!

2.2.1 Navigating to one location

The simplest planning problem is navigating from A to B. Ignoring rotations, and dynamic constraints, this problem is quite easy. In an open environment, the robot can move directly from its current location to its target destination. When there are obstacles in the environment, the shortest path is via *concave corners* of the environment (Figure 2.2).

This planning problem is in P. It can be solved by computing a visibility graph (see Appendix A) and then finding the shortest path on that graph (see Appendix B). The visibility graph can be computed using Welzl's algorithm in

$\mathcal{O}(n^2)$ [197] and the shortest path on this graph can be computed in $\mathcal{O}(n^2)$ using Dijkstra's algorithm [48] or the slightly more efficient A* algorithm [73]. If a robot is spends a long time in the same environment and will need to compute many shortest paths, it may be more efficient to compute all the paths ahead of time using the Floyd-Warshall algorithm which is $\mathcal{O}(n^3)$ but computes $\mathcal{O}(n^2)$ paths [60]. For computing a single path in a large environment, Missura's minimum construct algorithm can drastically improve performance by only computing a small portion of the visibility graph [137]. Although the shortest path is often used for planning, real robots may have kinodynamic constraints preventing them from following arbitrary paths. As detailed kinodynamic planning is computationally expensive, a robot can estimate transit times along feasible paths using simplified dynamics when deciding between many target locations, and only plan a detailed trajectory to the one selected location [26].

Visibility-graph-based planning algorithms are quick and deterministic. They are primarily useful for high-level planning for planar robots. If the robot is not able to follow an arbitrary path due to constraints that couple its translation and rotation, these simple visibility-graph-based algorithms do not necessarily result in plans that the robot can actually follow. For more complex robots, such as 7 degree-of-freedom arms, the robot's configuration space looks nothing like the plane and visibility-graph-based approaches don't even make sense! These more complex planning problems are typically solved using sampling-based algorithms.

Rapidly-exploring random trees (RRTs) are a powerful sampling-based tool for path planning [116]. These trees are constructed by randomly sampling the robot's configuration space and connecting the sampled point to an existing vertex in the tree (Figure 2.3). Starting with a single vertex representing the robot's initial configuration, additional vertices are added to the tree as follows:

1. Select a random point q in the configuration space

2. For each existing vertex in the tree, compute a path from the vertex to q
3. Select the vertex v which minimizes the length of the path to q
4. Follow the path from v to q a distance of 1 to find the point v'
5. Add v' to the tree with edge (v, v')

This approach is quite powerful as the random sampling results in the tree quickly expanding to fill the configuration space. The construction of the path from v to q ensures that v' is a valid configuration with a feasible transition from v to v'. Paths are planned by growing the tree from the start configuration until a vertex is located in a desired set of end configurations. While the paths produced are not necessarily optimal, they are always feasible and are usually fairly short. An asymptotically optimal variant, RRT* also exists [96] which rewires the tree when a new vertex is added so that previously-added vertices can use the new vertex if it results in a shorter path. A similar technique, probabilistic road maps [101] creates a general graph—not a tree—which is typically precomputed and can then be reused when planning many paths in the same environment.

2.2.2 The travelling salesperson problem

Complex tasks usually require a robot to visit multiple different locations. Delivery robots have to deliver multiple packages to different homes around a city. A packing robot in a warehouse has to collect all the items a customer has ordered from different shelves around the warehouse and put them all in the same box for shipping. Autonomous taxis have to drive to a passenger and then take them to their destination. A robotic plow must plow every location in a field. In all of these examples, the robot must go to multiple locations to complete some larger task.

When planning paths to multiple locations, the order that the locations are visited is very important. The shortest path to a destination depends on where

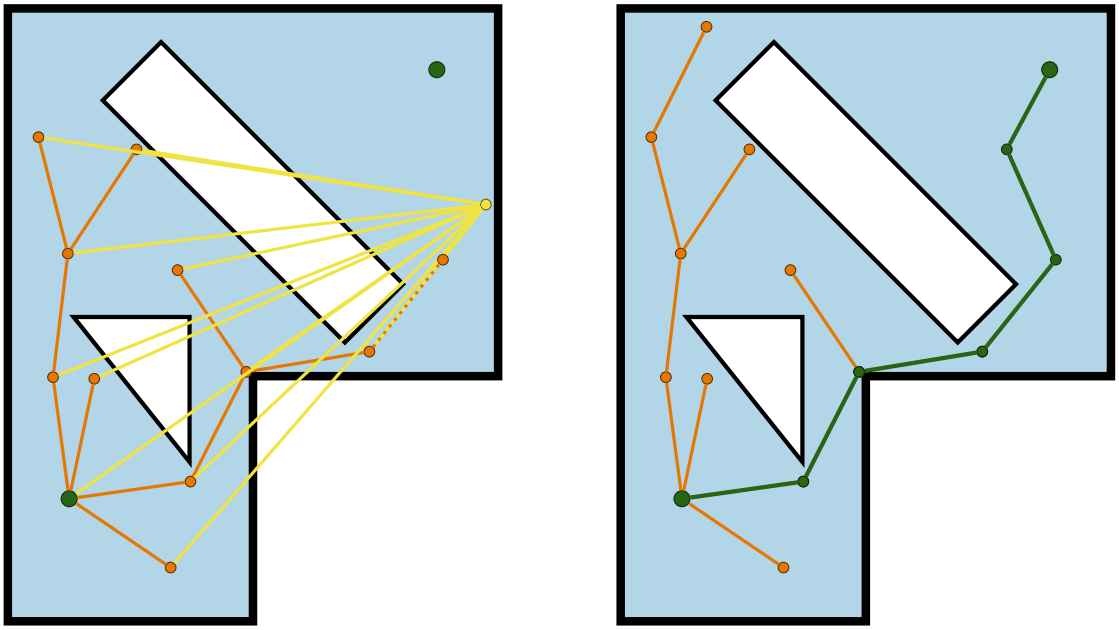


Figure 2.3: An RRT is incrementally grown by sampling a random point, choosing the nearest vertex in the tree, and then adding a new vertex connected to this nearest vertex in the direction of the sampled point (left). Once the vertex reaches the target destination, the algorithm terminates, resulting in a path from the initial location to the target destination which is guaranteed to be feasible (right).

the robot starts, and so a robot will typically take different paths to the same destination depending on which location it went to directly before. Multiple-destination tasks can be classified in one of two ways:

1. Tasks where the locations have to be visited in a specific order. For example, the autonomous taxi has to drop off its current passenger before picking up the next passenger.

2. Tasks where the robot must also choose the order of the locations. For example, the packing robot can typically collect the items for a single customer in any order.

Both scenarios rely on point-to-point planning algorithms. If the order is fixed, it is trivial to compute a plan that visits all the locations in the correct order by just finding the shortest path between each pair of consecutive locations. When

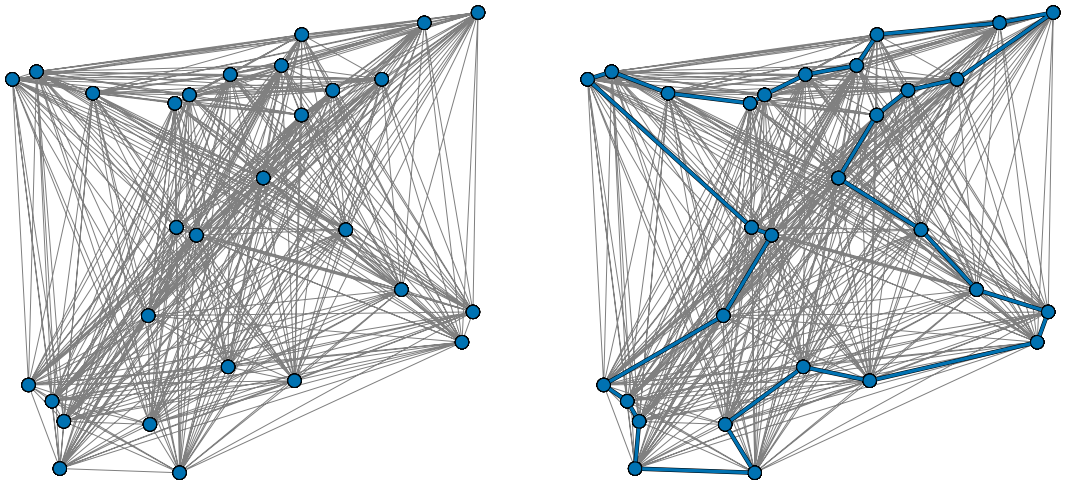


Figure 2.4: Complete graph with vertices representing tasks and edges representing distance between tasks (left). The shortest cycle on this graph (right) solves the travelling salesperson problem.

the order is not fixed, the robot will have to compute the best order based on the lengths of the shortest paths between each pair of locations. The best order can then be found by solving the travelling salesperson problem.

The travelling salesperson problem (TSP) is perhaps the most famous example of an NP-hard problem. The problem asks "given a list of cities and the times needed to travel between them, what is the fastest route for a salesperson to visit every single city?" (Figure 2.4). It has two main forms: in the tour-TSP the salesperson starts and ends in the same city; in the path-TSP, the salesperson starts and ends in different cities. In 1977, Papadimitriou proved that these two variants are polynomial reducible to each other and that each problem is NP-hard [150]. While the brute force approach to solving the TSP is $\mathcal{O}(n!)$, many polynomial time heuristics have been developed for this classic problem. I will focus on two variants. In the single travelling salesperson problem (1-TSP), there is one salesperson; in the multiple travelling salesperson problem (m-TSP), there are multiple salespeople.

An instance of the 1-TSP can be formulated in terms of a complete weighted

graph $\mathcal{G}$ with vertices $\mathcal{V}$, edges $\mathcal{E}$, and weights, w. For every pair of vertices in the graph there is an edge with a weight which indicates the distance between those cities. If the graph is not complete (i.e. some edges are missing), the missing edges can be filled in either with infinite weight, or with the length of the shortest path between the two vertices. The lengths of the shortest paths can be found in polynomial time using algorithms such as Dijkstra's algorithm and the Floyd-Warshall algorithm (Appendix B). Once we have a complete graph, the problem of the 1-TSP is to find the shortest path or shortest cycle which visits every vertex. Such a cycle is called an *spanning cycle* or *Hamiltonian cycle*.

The simplest heuristics for the 1-TSP are the nearest neighbor, greedy, and insertion heuristics [131]. The nearest neighbor heuristic is based on growing a path by choosing the shortest edge possible in each iteration. A random start vertex is chosen and then vertices are added to the path by choosing the closest vertex to an endpoint of the path which is not already part of the path. The greedy heuristic is based on adding the shortest edges possible without necessarily having a continuous path at all times. The edges are ordered based on length and the shortest edges are iteratively added. Once two edges have been added that both connect to the same vertex, all other edges connected to that vertex can no longer be chosen. Insertion is based on iteratively growing a cycle. First, the shortest cycle of three vertices is chosen. Then vertices are iteratively added to minimize the increase in cycle length when increasing the cycle size by one. All of these heuristics are simple to implement, fast to compute, and are often used as initial cycles that will then be improved to obtain a superior solution.

The TSP is closely related to the problem of finding a minimum spanning tree (MST) which is the smallest tree which contains every vertex of a graph. The MST is guaranteed to be shorter than the TSP solution and can be computed in $\mathcal{O}(|\mathcal{E}|\log(|\mathcal{V}|))$ by Kruskal's algorithm [112] (Section C.1) or in $\mathcal{O}(|\mathcal{E}|+|\mathcal{V}|\log(|\mathcal{V}|))$ by Prim's algorithm [154] (Section C.2). Christofides' algorithm is an elegant

heuristic which constructs a TSP using an MST and guarantees a solution which is at most one-and-a-half times the optimal cycle's length [39] (Section D.1). The MST is also used to define the Held-Karp lower bound, which is equal to the length of the minimum one-tree (an MST with one additional shortest edge added) of a transformed graph [75]. This transformed graph is also used to generate candidate edge sets in the Lin–Kernighan–Helsgaun heuristic [77].

As the TSP is a combinatorial optimization problem, potential TSP solutions can be improved using one of several heuristics which make incremental modifications. The simplest of these heuristics is 2-opt [44] where cycles are repeatedly improved by replacing two edges with two shorter edges (Section D.2). This idea can be generalized to k-opt where up to k edges are replaced simultaneously. Larger values of k are better at finding good solutions but the number of possible exchanges is $\mathcal{O}(|\mathcal{V}|^k)$ so it quickly becomes infeasible for large k. Lin and Kernighan's heuristic [122] is a variable k-opt—the value of k is determined by the algorithm. It limits the number of k-opt moves it needs to check by sequentially choosing pairs of edges to swap and only continuing to add pairs of edges to a potential move if the partial move would have a positive effect (Section D.3). This heuristic has been expanded by numerous authors, most notably in the Lin–Kernighan–Helsgaun algorithm [77] and chained Lin–Kernighan [12]. These variants are the bases of the two very effective open source solvers—LKH [76] and Concorde [42]—which have both solved the 110 standard test problems on TSPLIB [155] to optimality. Other generic combinatorial optimization techniques, such as simulated annealing and tabu search [130], have been applied to the TSP with good results.

2.2.3 The multiple TSP

The multiple travelling salesperson problem is similar to the 1-TSP except there are multiple salespeople. There are two common objectives for the m-TSP (Figure 2.5). In the minsum m-TSP, the objective is to minimize the sum of the

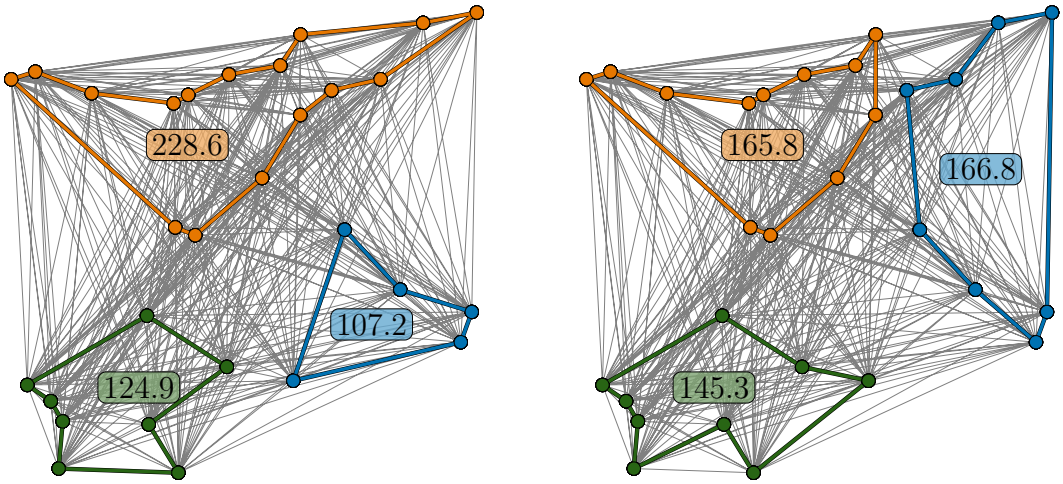


Figure 2.5: The multiple travelling salesperson problem has two main forms. In the minsum m-TSP (left), the sum of the cycle lengths is minimized; whereas in the minmax m-TSP (right), the length of the longest cycle is minimized.

distance travelled by individual salespeople. In the minmax m-TSP, the objective is to minimize the time taken by the slowest salesperson. These objectives are usually conflicting even for small problems [158]. The minsum objective tends to result in some salespeople visiting many cities while others visit few. The minmax objective balances the workload so that all the salespeople take a similar amount of time and therefore usually visit similar numbers of cities. However, a minsum objective with a constraint on agents' path lengths can have a similar result [190, 193].

Unlike the 1-TSP, the locations where the salespeople start is important in the m-TSP. If they all start in distinct locations, each of these locations must be included in exactly one tour in the solution; if they start in the same location, that location will have to be part of all m tours. Additionally, each salesperson's path could be a path or a cycle, resulting in even more variants of the m-TSP!

Most m-TSP algorithms are based on a modification of the 1-TSP [25]. There

are several different ways to convert the m-TSP graph into a slightly different 1-TSP graph, however they are all intended for the minsum m-TSP. If all salespeople must start and end in the same place, that vertex can be repeated once for each salesperson and then, by chopping up the 1-TSP solution whenever it reaches one of those vertices, we obtain an m-TSP solution [72, 181]. As all the repeated vertices are equivalent, this approach results in many equivalent solutions which can make the 1-TSP difficult to solve. These equivalent solutions can be avoided by making each duplicated vertex only accessible from a few other vertices by giving some edges infinite weight [92]. Rather than duplicate the shared vertex, it can be removed, and then after chopping up the solution to the smaller 1-TSP, each partial path can be connected back to the home vertex to get an m-TSP solution [24, 138]. When performing k-opt on a 1-TSP solution that will eventually get split into multiple pieces, the requirement that the improved solution must be a single cycle can be relaxed [153]. These approaches allow any 1-TSP heuristic to be used for the m-TSP but they can only be used for the minsum and not for the minmax.

The minmax m-TSP is generally much more difficult as the cost function only depends on one salesperson's path at a time and is non-linear. Some changes to the solution may result in a change in which salesperson has the longest path while others may have no effect on the minmax cost even though the change reduces the length of a salesperson's paths. Most m-TSP heuristics use a common approach of cycle generation, improvement, and recombination. Initial sets of cycles can be generated by a modified version of Christofides' algorithm [30, 62], k-means clustering [103, 144], k-centers clustering [143], nearest neighbor, greedy, or random heuristics. These cycles can be improved by tabu search [149], simulated annealing [172], compressed annealing [125], or general variable neighborhood search [176]. If many solutions are generated, they can be recombined using

evolutionary methods [6], ant colony optimization [123], invasive weed optimization [191], or a memetic algorithm [194]. These algorithms are all based on generic combinatorial optimization techniques and do not use any intuition about the actual structure of the minmax m-TSP. In Chapter 3, I present a combinatorial algorithm developed specifically for the minmax m-TSP which partitions a graph based on a minmax criterion, then solves the 1-TSP on each subgraph of the partition, and then improves these paths, again using a minmax criterion. In addition to being more intuitive, this algorithm has outperformed the state-of-the-art m-TSP algorithms [103, 194] in terms of solution quality and computation time on several large scale test problems.

2.3 Robot-to-robot communication

As robots become more common in society, there will be more opportunities for robots to interact and cooperate with each other. When humans cooperate, we can achieve things that wouldn't be possible for a single person, and we can work much faster than if we worked alone. Similarly, robot teams can cooperate to work faster and accomplish more complex tasks. How effectively they can cooperate depends on two things:

1. How effectively they can communicate to share information and make plans for the team; and

2. Whether or not they have algorithms that can take advantage of their ability to communicate.

How well they can communicate depends on the robots' hardware, whereas their ability to exploit communication depends on their software. Although my thesis focuses on how robots can use communication to their advantage, any algorithm for coordinating robots is only effective if the robots can actually communicate as well as the algorithm expects.

2.3.1 Realistic communication models

Most robots communicate through some form of wireless communication, such as light, sound, or radio waves. These wireless signals do not have infinite range. They are often blocked, attenuated, or reflected by various solid obstacles commonly encountered in typical indoor or outdoor environments. Wireless communication depends on signal strength which is affected by three phenomena [140]:

1. **Path loss** is a linear decrease in strength due to distance from the signal source. The decrease in signal strength depends on the environment and type of signal, however, it has experimentally been measured to approximately follow an inverse square or inverse cube law [141].

2. **Shadowing** is loss in signal strength due to obstacles between the source and receiver. Although some obstacles do not fully block certain types of signal, we will treat it as a binary effect where signals can only be transmitted if there is a direct line-of-sight between the source and receiver.

3. **Multipath fading** is caused by destructive interference when a signal reflects off of different surfaces and multiple signals are all received with different phase shifts. Its effect tends to be small and stochastic.

Based on these phenomena, the two main factors that determine how well two robots can communicate are distance and line-of-sight.

The effects of distance and line-of-sight can be combined to create several different communication models (Figure 2.6). Depending on the type of signal and properties of obstacles, line-of-sight may have little to no effect (e.g. using visible light to communicate in an environment with glass walls) or may completely block signals. Distance always decreases signal strength, however this decrease in signal strength may have different effects on whether or not communication is possible. Above a certain signal strength, communication is always possible. If the signals

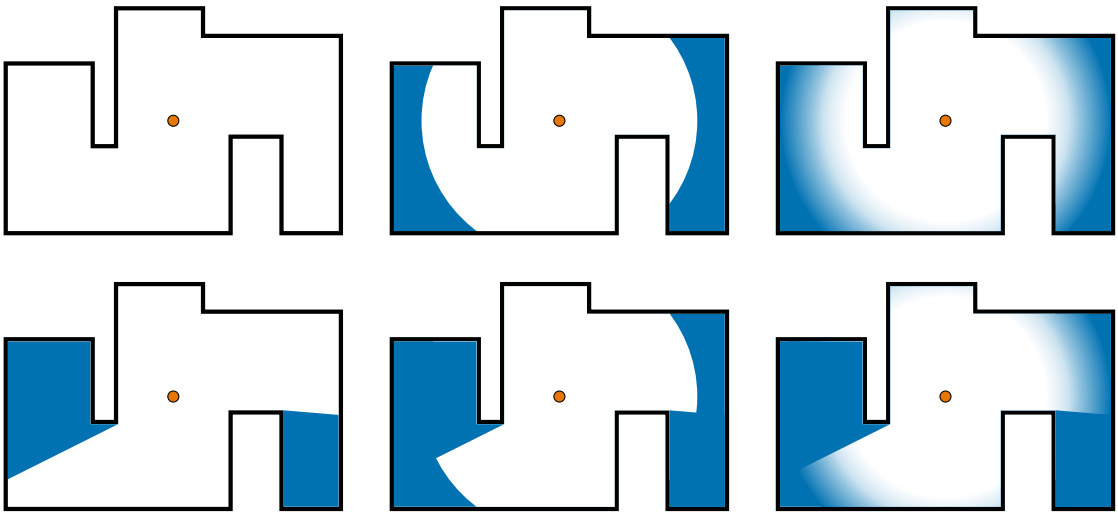


Figure 2.6: Communication between robots can depend on distance, line-of-sight, or both. Distance can have no effect (left), a binary effect (center), or a gradual effect (right). If communication signals can pass through obstacles (top), only distance has an effect; if obstacles block signals (right), line-of-sight and distance both effect communication.

are strong and the environment is small, the strength will always be above this threshold and distance has no effect. For weaker signals or larger environments, if the distance is large enough, the strength will drop below this threshold and either reduce the probability of communication or prevent communication entirely. Rather than use a predetermined communication model, it is also possible to construct a communication map online using data obtained by a team of robots as they move through their environment [119].

Sometimes robots communication via some intermediate device (e.g. using a Wi-Fi network). In these cases, distance and line-of-sight still effect communication. However the strength is not based on the robots' positions relative to each other, but instead to their position relative to the intermediate devices on the network. Often, a communication network provides strong enough signals everywhere that the robots can always communicate.

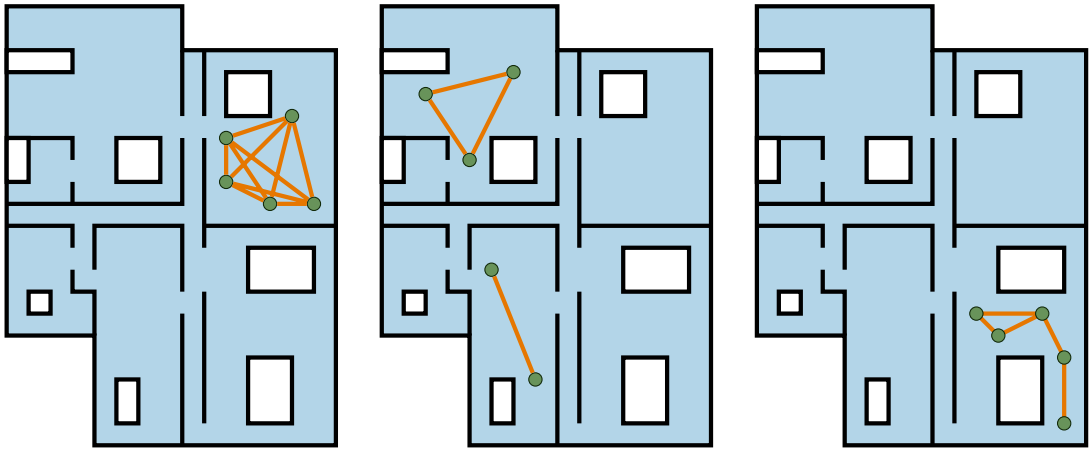


Figure 2.7: For a team of robots to be connected, we don't necessarily need all pairs of robots to be directly connected (left). Similarly, it is not sufficient to simply check that each robot is connected to another robot (center). Instead, we need to check that there is at least one, potentially indirect path between any two robots (right).

2.3.2 Maintaining connectivity

An easy way to enable coordination in a team of robots is to constrain their motion so that the entire team is always connected. What does connectivity mean for a team? For two robots, the team is connected if the two robots can communicate directly. For teams of $m > 2$ robots, we don't need all the robots to be able to communicate directly. Instead, we simply require each robot to be connected to every other robot by some chain of connected robots (Figure 2.7).

In mathematical terms, we require the communication graph to be connected. At a first glance, it is not obvious how to check that a graph is connected. A sufficient, but not necessary, condition is that every pair of robots is directly connected. A necessary, but not sufficient condition is that every robot is connected to another robot. We need some in between criterion condition which is both necessary and sufficient. It turns out that we can determine a graph's connectivity by examining its Fiedler eigenvalue, the second smallest eigenvalue of the graph's Laplacian matrix [59]. If this eigenvalue is zero, the graph is disconnected; if it is strictly positive, the graph is connected. The Laplacian matrix can be computed directly

from the adjacency matrix and eigenvalues can be computed in cubic time, so we can test whether or not a team of robots is connected in $\mathcal{O}(m^3)$. Larger values of the Fiedler eigenvalue indicate a larger average number of communication routes between pairs of robots (or better average strength if the Laplacian is weighted by signal strength).

Algorithms that maintain connectivity often use the Fiedler eigenvalue. Weighting the Laplacian by signal strength results in a Laplacian that changes continuously as robots move and so a gradient ascent can be used to find directions which maintain or improve connectivity of the network [179]. A direction of motion which causes the Fiedler eigenvalue to increase can be combined with other directions related to the robot's task to enable the team to accomplish their task without losing connectivity [163]. The relationship between a robot's position and the team's Fiedler eigenvalue can also be used to construct potential fields that can be added to potential fields for collision avoidance to maintain connectivity and prevent collisions, both between robots and with obstacles [46, 205]. These approaches to maintaining connectivity always assume that the network topology is constant as changing topology results in discontinuities in the Fiedler eigenvalue.

An alternative way to enforce connectivity is to use it as an explicit constraint when robots plan their paths. This constraint can significantly reduce where robots can move, especially when limited by line-of-sight. As a result, it is often only useful when there are large numbers of robots and several of the robots just behave as routers forming chains of communication between far apart robots that are performing other tasks [151]. Alternatively, the team can split into two smaller teams provided they plan a time and place to rendezvous and each robot is constrained to stay connected to the other robots in its smaller team [156].

2.3.3 Occasional connectivity

Communication constraints can be restrictive especially for robots and environments with poor communication channels. Connectivity generally requires the robots to be near each other and coordinate their motion, whereas many other tasks, such as exploration [7], benefit from robots spreading out without necessarily maintaining connectivity. Since these objectives are competing, in many cases it may be more efficient for the team to only communicate occasionally.

There are several ways to coordinate robotic behavior without always having a connected network. If the connectivity of the graph varies over time, the network can be described by a time-varying graph. There are many different notions of connectivity for time-varying graphs [32], but as long as the communication graph is *recurrently connected*, information can flow throughout the graph even if it is never fully connected at any instance. Periodic connectivity is a more restrictive form of time-vary connectivity where robots' communication network must be connected at periodic intervals but doesn't need to be connected in between. This form of connectivity is relatively easy to implement and fairly flexible, as individual robots only have their paths constrained at certain times and can plan independently between these meetings [80, 94]. This approach allows the robots to meet in different locations at different times and does not require them to be constantly connected. Recurrent connectivity is a less restrictive requirement because the entire network doesn't have to ever be connected simultaneously. If there are sufficiently many robots in an environment, it is likely that the network will be recurrently connected even if the robots do not consider connectivity when planning their paths and just opportunistically communicate with robots who happen to be nearby [199]. In Chapter 5, I present a form of recurrent connectivity where robots normally do not consider communication when planning and only plan based on communication requirements when they have a need to share information or coordinate with

a disconnected robot.

2.3.4 Robotic search

Sometimes, a team of robots gets separated. The two subteams (possibly a single robot team) cannot communicate when separated so they cannot plan where to meet and will instead need to search for each other. In robotics, search problems can be broadly classified based on the behavior of the target robot [4, 40, 159].

- **Rendezvous** is when the target is cooperative. This situation is symmetric as both robots are both simultaneously trying to find each other.
- **Pursuit-evasion** is when the target is actively avoiding the searcher. Often, the target is assumed to move infinitely fast, and the searcher uses a path that traps the target in a corner.
- **Search** is when the target is indifferent to the searcher. This target could be stationary or mobile. If it is mobile, its motion does not depend at all on what the searcher is doing.

In a cooperative team, the target is another robot in the team and is therefore never adversarial. The target may be actively searching for the searcher, resulting in a rendezvous problem, or it may be performing some other task, resulting in a search problem. This *rendezvous-search* problem—central to my coordination approach (Chapter 5)—can be formalized similarly to the *rendezvous-evasion* problem [5] where the searcher uses a mixed strategy dependent on the probability that its target is friendly.

Searching for a stationary target is as simple as finding a path that can see the entire environment, which can be achieved by solving a version of the TSP [114] or the related Chinese postperson problem [93]. A moving target is more difficult to find because the searcher may have to visit the same location multiple times

before finding the target. Typically, moving targets are modelled using a Markov model, or similar probabilistic model, and this model is used to maintain a belief about the target's position. Then the searcher can plan a path by solving an NP-hard optimization problem using a technique such as branch-and-bound [115] or by solving a partially observed Markov decision process [81].

When two robots are attempting to rendezvous, they both use the same strategy to search for each other. A simple strategy is to have one robot remain in one place while the other robot searches for it [8]. Since the robots cannot communicate, they have no way to choose *which* robot should search so each robot will randomly decide whether to search or wait for a period of time and hope that its teammate chooses the other behavior. To avoid the one quarter chance that both robots choose to wait, they can use a different approach where both robots always search by travelling between the most distinctive several locations in the environment [52]. As long as both robots have equivalent ways of measuring distinctiveness, they will both have a common set of locations despite not communicating and will be more likely to rendezvous faster than if both robots searched every part of the environment [33].

2.4 Coverage

Robotic coverage is task where a robot must travel over every point in an environment. Lawn mowing, painting, milling, vacuuming, plowing, and surveillance are all coverage problems. In each application, the robot's tool (e.g. rotating blade, paint brush, or camera) traces out a two dimensional region as it moves (Figure 2.8). Coverage planning refers to the problem of finding a path for the robot which results in its tool covering the required area. Ideally, the planned coverage path minimizes some criterion such as the time needed to follow that path.

Coverage is closely related to the TSP (Subsection 2.2.2) and is also NP-hard

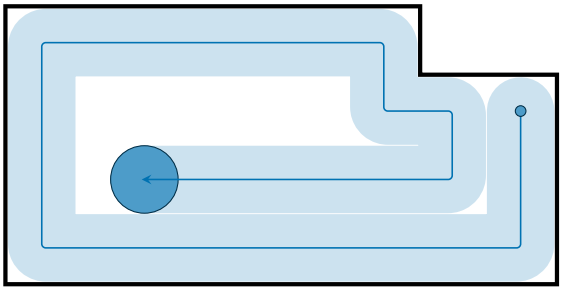


Figure 2.8: When a robot moves through its environment, its tool sweeps out a 2-dimensional region along its path. The task of coverage is to find a short path so that this region covers the whole space.

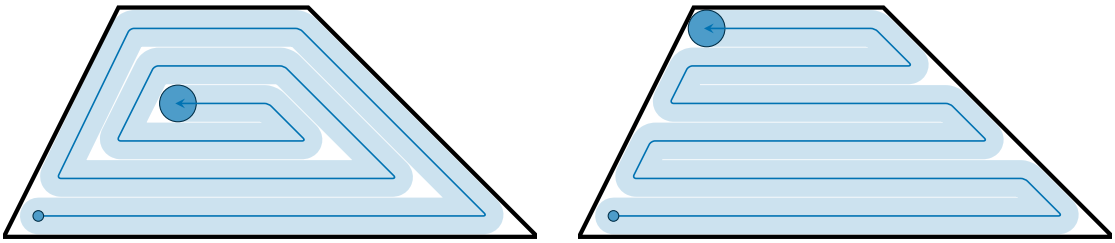


Figure 2.9: Coverage of a trapezoid using two strategies. Contour-parallel paths (left) follow the region's perimeter; direction-parallel paths (right) move back in forth in straight lines.

[14]. Both problems involve travelling to many different locations in an order that must be chosen to complete the task as efficiently as possible. In the TSP, the salesperson must travel to cities which are predefined locations. In coverage, the robot does not have predefined locations. Instead, it can choose any set of regions—each of which can be covered by simple motion—so that when the robot visits and covers all of them, it will cover the whole environment. This set of regions is a partition of the entire space called a *decomposition*. Typically, the regions are small and have simple geometry so they can be covered by simple strategies such as contour-parallel or direction-parallel paths [74] (Figure 2.9). The order that cells are covered is determined by solving the TSP to get a full coverage plan.

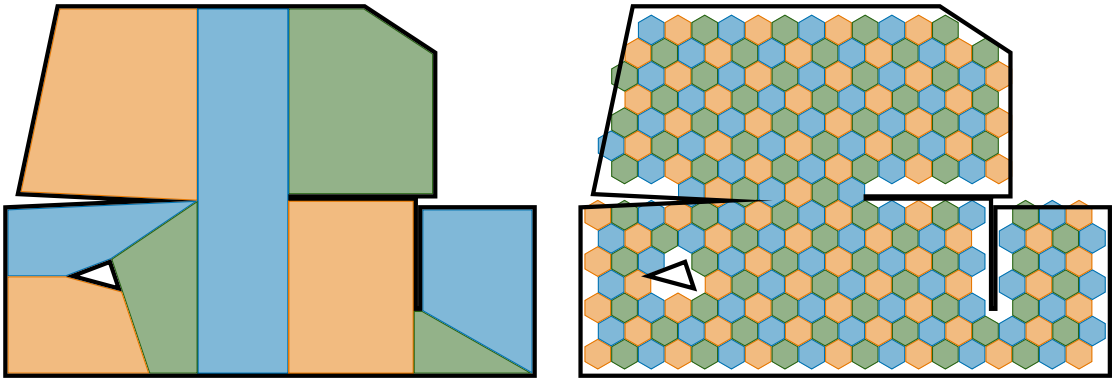


Figure 2.10: Geometric decompositions used in coverage are typically classified as either exact or approximate. Exact decompositions (left) consist of large cells of varying shapes and sizes. Approximate decompositions (right) consist of many small identically shaped cells. Here, we only show the portion of cells near the boundary which are inside the environment.

2.4.1 Decompositions

Many coverage algorithms are based on a geometric decomposition of the environment [37, 65]. Basic coverage strategies like the contour- and direction-parallel strategies only work well in simple convex environments. For non-convex environments, a decomposition divides the environment into several small convex regions called *cells* so that these strategies can still be applied. As there are many ways to divide a polygon into several smaller pieces, there are many different kinds of decompositions used in coverage. They are broadly classified as exact or approximate (Figure 2.10).

Exact decompositions divide the environment into several cells that vary in size but whose union is exactly equal to the original environment. The cells are typically large and have simple geometries but can be generated by several different methods. The boustrophedon decomposition slices the environment using parallel vertical lines which are drawn at any x value where the topology of the environment changes [38]. A Morse decomposition slices the environment based

on the topology of level sets of a special function called a Morse function [1] (Figure 2.11). This approach can be particularly beneficial on uneven terrain, where the elevation is used as a Morse function, and the resulting coverage paths limit the amount the robot must drive up or downhill [64]. In the special case where the Morse function is linear, the resulting decomposition is a boustrophedon decomposition [109]! These kinds of decompositions are particularly useful as a robot can cover each cell by moving along level set curves, which for the boustrophedon decomposition results in direction-parallel paths. The name boustrophedon means "the way of the ox" describing this back-and-forth method of covering a cell. Since the boustrophedon decomposition is based on changes in topology along straight lines, it can be generated in an unknown environment by two robots which travel along parallel paths and can detect obstacles between them via occlusion [156].

Approximate decompositions use small cells that are all the same shape and size—usually smaller than the footprint of the robot's tool. The idea is that if the robot's tool passes over every cell of such a decomposition, then it has effectively covered the entire space (Figure 2.12). Although these decompositions are typically assumed to be square grids, hexagonal or triangular grids could work equally well. Once the decomposition has been computed, the shortest path on it can be found by solving the TSP on the set of grid cells, or using a fast heuristic [206]. Alternatively, if a larger grid—twice the size of the robot's tool—is used, then the robot can simply follow around both sides of the MST and avoid having to solve the TSP [2]. These approximate approaches often do not work well in practice as the decomposition results in a large number of cells, which is computationally expensive, and the restriction to a grid can cause the robot to miss spots near the walls.

My coverage algorithm (Chapter 4) uses a decomposition which is neither grid-based or exact. Instead, each cell of the decomposition is a unit width rectangle called a *rank*. These ranks can be any length and have any orientation. They

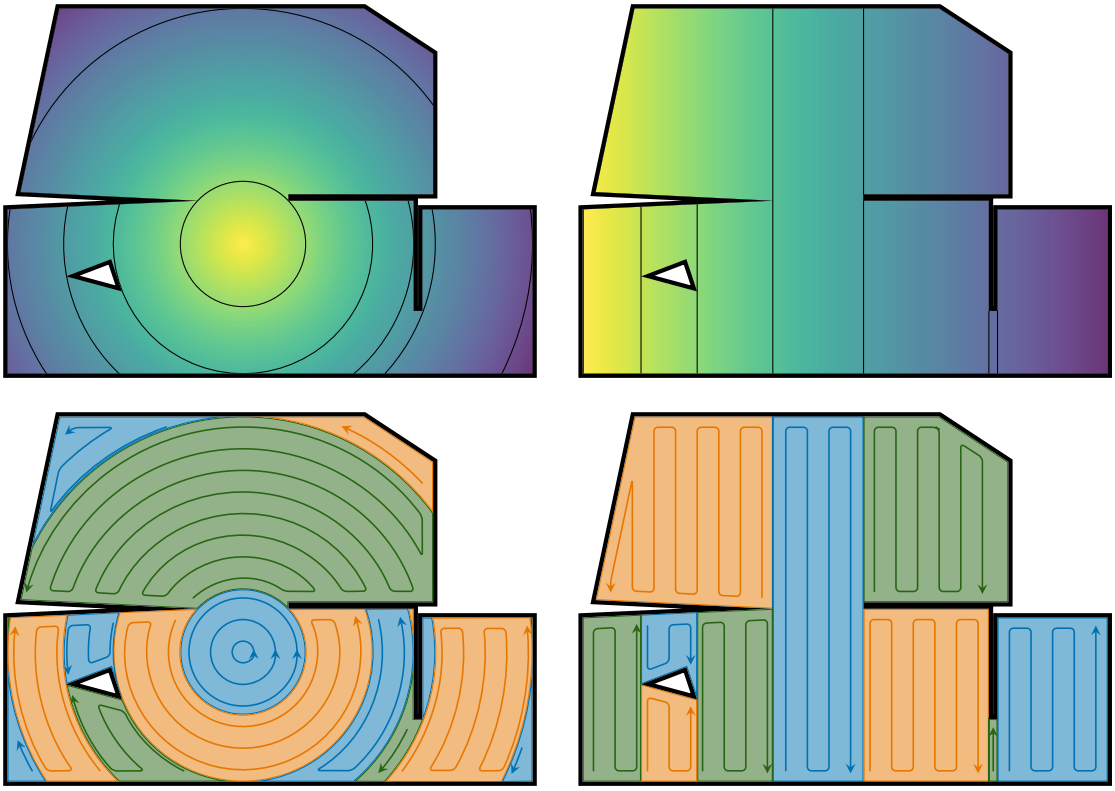


Figure 2.11: The Morse decomposition (left) is defined using the level sets of a *Morse function* (here, the distance from a central point) as boundaries between cells are defined at level sets where the topology of the environment changes. The boustrophedon decomposition (right) is a special case where the Morse function is linear. Coverage paths for each decomposition can be obtained by connecting the simple paths for each cell (bottom).

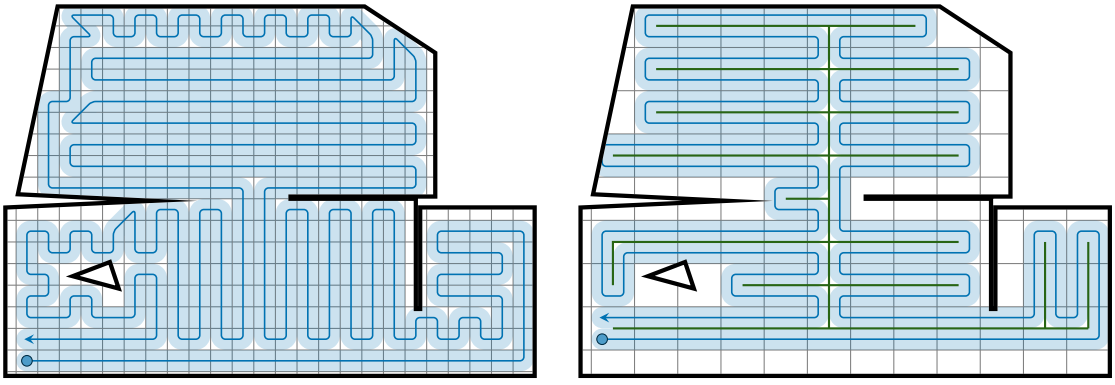


Figure 2.12: Approximate decompositions can be used by solving the TSP on a fine grid (left) or by following the perimeter of the MST on a coarser grid (right).

completely cover the environment (except possibly some small parts in corners that are too narrow for the robot) but there is usually some overlap between ranks to guarantee full coverage as robots can miss small pieces when turning.

2.4.2 Turn-minimization

A lot of coverage algorithms only care about finding the shortest coverage path. They don't care about the *quality* of this path. In general, paths with many turns are bad. Most robots are able to move efficiently in straight lines, but have difficulties making precise turns. Furthermore, robots performing specific tasks might be unable to do that task effectively while turning. When a painting robot turns, it will leave more paint on the inside part of the turn than the outside, resulting in an inconsistent thickness of paint [11]. When a UAV with a fixed camera turns, its camera does not point at the ground so it is less effective at scanning or surveilling during a turn [15]. Turns also take time—the robot likely needs to come to a stop or at least decelerate—so the time it takes a robot to follow a path is not directly proportional to its length.

All of these factors make turn-minimization an important criterion in coverage planning in addition to distance minimization. For a convex polygon, turns are minimized by using a direction-parallel strategy whose ranks are perpendicular to the direction that minimizes the height of the polygon [85] (Figure 2.13). For non-convex polygons, simply minimizing the number of turns on each convex cell of a decomposition does not mean that we have minimized turns for the entire polygon. The total number of turns will depend on which decomposition is used. The ranks generated by my coverage algorithm (Chapter 4) are designed to be as long as possible to minimize the total number of ranks which is equivalent to minimizing the number of turns.

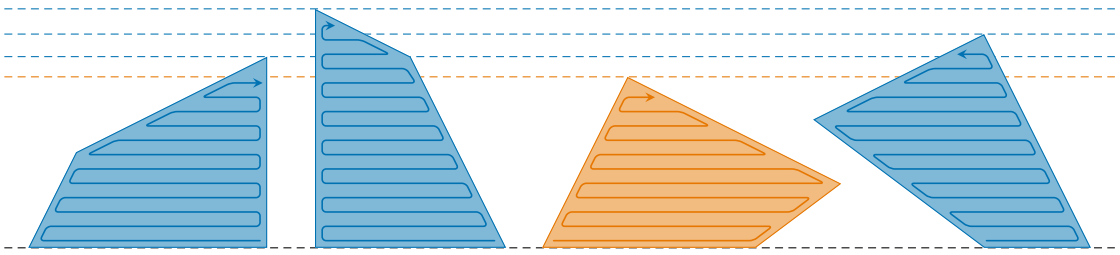


Figure 2.13: A convex cell can be covered using a minimal number of turns using a direction-parallel strategy with the ranks parallel to one of the cell's edges. The minimizing direction (second from right) is perpendicular to the direction that minimizes the height of the polygon.

Chapter 3

Balanced task allocation in multirobot teams

Collaboration between robots involves sharing work. The work can usually be broken down into a set of small tasks, which each must be performed by a single robot. A team of delivery robots has to complete many individual deliveries that are each a single task. For monitoring robots, individual tasks could be taking a photograph or measuring some data at a specific location. When robotic snowplows clear snow from city streets, their basic tasks are clearing the snow from a single road or short section of a road. In all of these examples, all the tasks only require one robot and in theory, the entire job could be performed by a single robot. However, it is often much more efficient to use multiple robots as n robots can complete all the necessary tasks in approximately $1/n^{\text{th}}$ of the time it would take a single robot.

Teams of robots can share work by assigning different tasks to each robot. The entire point of using multiple robots is to have the team finish the mission as quickly as possible. Therefore, the tasks must be assigned to minimize

$$\text{Time taken by team} = \max_{\text{robot} \in \text{team}} \{\text{Time taken by robot}\}.$$

When a robot is assigned a task associated with a physical location, it must travel to that location before completing the task. The total time taken to complete its

assigned tasks is the time spent performing each task plus the time spent travelling between tasks. As the time needed to travel to any task depends on where the robot is coming from, the travel time depends on which task was previous. Therefore the order of tasks affects how quickly each robot completes its assignment and must be considered when fairly assigning tasks among robots.

This task allocation problem is equivalent to the minmax multiple travelling salesperson problem (m-TSP). In this chapter, I present a new combinatorial approach to solving the minmax m-TSP. My approach is based on the perspective of the m-TSP as a partition problem instead of a routing problem (Figure 3.1) and exploits a near-monotonic relationship between average and minimum spanning cycle lengths (Subsection 3.3.1). This near-monotonicity results in a novel transformation between two optimization problems whose cost functions are not proportional. My task allocation solves the transformed optimization problem, which uses the average spanning cycle length, which is easier to compute, as a cost function. The algorithm consists of partitioning (Section 3.4) and routing (Section 3.5) phases, can be decentralized (Section 3.7), and can incorporate constraints on the robots' start and end locations (Section 3.8). Some comparisons with existing minmax m-TSP algorithms are presented in Section 3.9. This chapter is an expanded version of my paper "Balanced task allocation by partitioning the minmax multiple travelling salesperson problem" [187].

3.1 Related work

One way to formulate task allocation problems is with the central objective of maximizing the *utility* of the tasks allocated to each robot [68, 111]. For heterogeneous teams, different robots have different abilities so the utility of each task depends on which robot it is assigned to [67]; for homogeneous teams, all robots are identical so the utility depends only on the task [134]. Utility can be defined as a sum over

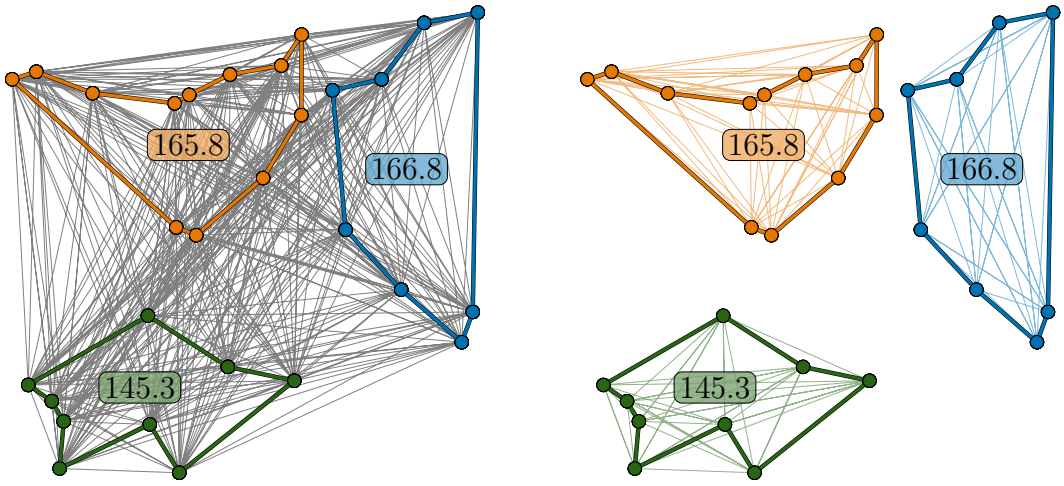


Figure 3.1: The minmax m-TSP is often defined as a routing problem (left) where the objective is to find several cycles on the same graph. Alternatively, it can be viewed as a partition problem (right) where the objective is to divide the graph into several subgraphs. By measuring a subgraph's size as the length of its shortest size, the two perspectives are equivalent.

subsets of tasks and is not necessarily the linear sum of the utilities of individual tasks [146, 168]. The objective of task allocation is to maximize either the sum of utilities, or the utility of the robot with the smallest utility [207]. Maximizing the smallest utility—equivalent to minimizing the maximum time required—results in a balanced allocation of tasks, which I believe is much more useful for most robotic applications.

Economic methods, such as auctions [36, 67, 145], markets [47, 207], or token exchange [57] are often used to assign tasks. These methods are distributed: tasks are sequentially assigned to individual robots but can be transferred between robots when necessary. Alternatively, the same task can be assigned to multiple robots who then compete to complete tasks and achieve a reward—called a bounty—after completing a task [198]. Task assignment and path finding are often combined into a single problem to find optimal assignments while planning collision free paths [84, 127, 134]. For robots in constrained environments where

kindodynamic constraints are important, task assignment and kinodynamic planning can be decoupled by estimating transit times using simplified dynamics to compute feasible trajectories between every pair of tasks, with a single detailed trajectory planned for each robot after task allocation [26].

For mobile robots, the combined task allocation and routing problem is closely related to the m-TSP [13, 102, 166, 184, 202]. The m-TSP asks: "What is the quickest way for m salespeople to visit a set of n cities?" and is a generalization of the classic 1-TSP, which is well-known NP-hard routing problem [150]. While task assignment is primarily a *partition* problem, the m-TSP is primarily a *routing* problem.

The m-TSP (Subsection 2.2.3) has two variants with different objectives:

- The objective of the *minsum m-TSP* is to minimize the total distance travelled by the team,

$$d_{\text{total}} = \sum_{\text{robot} \in \text{team}} d_{\text{robot}},$$

  without any requirement that each robot does a similar amount of work. One robot may perform the majority of the tasks—especially if a lot of them are located near each other—travelling much further and finishing much later than its teammates who only do a few tasks each.

- The objective of the *minmax m-TSP* is to minimize the mission time,

$$t_{\text{mission}} = \max_{\text{robot} \in \text{team}} \left\{ \frac{d_{\text{robot}}}{s_{\text{robot}}} \right\},$$

  even if it results in the team travelling a longer total distance. Under this objective, if some robot were to finish while another robot still has several tasks left, a better assignment would have some of the slowest robot's tasks transferred to a faster robot. The resulting assignment would have all robots finishing at approximately the same, balancing the workload.

Of these two conflicting objectives, the minmax objective results in a team of n robots completing a mission in approximately $1/n^{\text{th}}$ of the time needed by a single robot, whereas the minsum objective does not drastically improve the total distance travelled by the team as more robots are added. My task allocation algorithm uses a minmax objective as it takes advantage of coordination to complete the mission faster.

Both variants of the m-TSP are NP-hard as they are generalization of the NP-hard 1-TSP [150] and so these problems are usually solved by heuristics which run in polynomial time but are not guaranteed to find the optimal solution. As the minmax m-TSP is non-linear in the individual robots' path lengths, heuristics for the minsum m-TSP [24, 25, 72, 92, 138, 153, 181]—which is linear in the individual robots' path lengths—are not easily generalizable to the minmax m-TSP. Existing successful heuristics for the minmax m-TSP have used many different techniques such as:

- Tabu search [149]
- Simulated annealing [143, 172]
- Compressed annealing [125]
- Markets [103]
- Ant colony optimization [123, 125]
- Invasive weed optimization [191]
- Variable neighborhood search [176]
- Evolutionary algorithms [6, 30, 194]

Many of these techniques use solutions to the 1-TSP. There are several efficient techniques for solving the 1-TSP, most notably the open-source LKH [76] and Concorde [42] solvers which are both based on the Lin–Kernighan heuristic [122] (Section D.3).

3.2 Task allocation and the m-TSP

The combined task allocation and routing problem can be defined on a complete graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V}$ representing tasks and edge set $\mathcal{E} = \mathcal{V} \times \mathcal{V}$ representing transit between two tasks. Task completion times are represented by a function, $w_{\text{vertex}} : \mathcal{V} \to \mathbb{R}_{\geq 0}$, and transit times are represented by $w_{\text{edge}} : \mathcal{E} \to \mathbb{R}_{\geq 0}$. I will denote the number of tasks by n and the number of robots by m.

Suppose robot i is assigned a set of tasks, $\mathcal{V}_i \subset \mathcal{V}$ with $n_i = |\mathcal{V}_i|$. The time needed to complete these tasks depends on the order they are completed. This order can be represented as a cycle $c = (e_0, \ldots, e_{n_i})$ which visits each vertex of $\mathcal{V}_i$ once. For a set of tasks, $\mathcal{V}_i$, and route, c, the completion time is

$$t_{\text{total}}(\mathcal{V}_i, c) = \sum_{v \in \mathcal{V}_i} w_{\text{vertex}}(v) + \sum_{e \in c} w_{\text{edge}}(e). \tag{3.1}$$

Each vertex is incident to exactly two edges of the cycle, so we can define an overall weight function $w : \mathcal{E} \to \mathbb{R}_{\geq 0}$ by

$$w(e) = \frac{1}{2}\Big(w_{\text{vertex}}(v_0) + w_{\text{vertex}}(v_1) \Big) + w_{\text{edge}}(e)$$

where v_0 and v_1 are the two edges of e. Rather than use the vertex weights, w_{vertex}, and edge weights w_{edge}, we can use this overall weight and simply rewrite (3.1) as

$$t_{\text{total}}(\mathcal{V}_i, c) = \sum_{e \in c} w(e).$$

The resulting task allocation problem which depends only on w is equivalent to the original problem depending on w_{vertex} and w_{edge}. For the remainder of this chapter, I will simplify notation by using the overall weight, w, instead of w_{vertex} and w_{edge}.

A subset of vertices, $\mathcal{V}_i \subset \mathcal{V}$, induces a subgraph, $\mathcal{G}_i$ of $\mathcal{G}$. This subgraph

contains all edges of $\mathcal{G}$ between two vertices of $\mathcal{V}_i$ and is therefore precisely $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{V}_i \times \mathcal{V}_i)$. A *spanning* cycle on $\mathcal{G}_i$ is any cycle which visits each vertex of $\mathcal{V}_i$ exactly once. Let $c^*(\mathcal{G}_i)$ be the shortest spanning cycle on $\mathcal{G}_i$. Robot i can complete its assigned tasks as quickly as possible by following $c^*(\mathcal{G}_i)$ and so we define the *size* of the subgraph, $\mathcal{G}_i$, by

$$S_{\min}(\mathcal{G}_i) = \sum_{e \in c^*(\mathcal{G}_i)} w(e). \tag{3.2}$$

Computing $S_{\min}$ is difficult because finding the shortest spanning cycle is equivalent to solving the 1-TSP which is NP-hard [150]. In Subsection 3.3.1 we define an alternate size of subgraphs which can be computed in polynomial time and provide relationship between it and $S_{\min}$. As far as I am aware, this proxy has not previously been studied in the literature.

A partition, $\mathcal{P} = \{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$, of $\mathcal{G}$ is a set of subgraphs with each vertex of $\mathcal{G}$ contained in at least one (usually exactly one) subgraph, $\mathcal{G}_i$. I will define the *cost* of a partition as the size of that partition's largest subgraph,

$$C_{\min}(\mathcal{P}) = \max_{\mathcal{G}_i \in \mathcal{P}} \{S_{\min}(\mathcal{G}_i)\}. \tag{3.3}$$

A subgraph's size equals the minimum amount of time needed for a robot to complete all of the tasks in that subgraph, so the cost of a partition equals the team's total mission time if tasks are assigned according to $\mathcal{P}$. Using this cost function, the task assignment problem is equivalent to finding the partition, $\mathcal{P}^*$, of $\mathcal{G}$ with the smallest cost, $C_{\min}$. I will call this partition problem the Minimum Partition Problem (MPP).

Problem 3.1 (MPP)**.** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$ be a complete weighted graph. For a given $m \geq 2$, find a partition, $\mathcal{P} = \{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$, of $\mathcal{G}$ which minimizes $C_{\min}$ as defined in* (3.3)*.*

The MPP is closely related to the minmax multiple travelling salesperson problem (m-TSP). A solution to the m-TSP is a set of m disjoint cycles, $\mathcal{C} = \{c_1, \ldots, c_m\}$, such that each $v \in \mathcal{V}$ is in at least one (usually exactly one) cycle of $\mathcal{C}$. The cost of a candidate solution is the length of its longest cycle:

$$C(\mathcal{C}) = \max_{c \in \mathcal{C}} \left\{ \sum_{e \in c} w(e) \right\}. \tag{3.4}$$

The objective of the m-TSP is to find the set of cycles, $\mathcal{C}^*$, on $\mathcal{G}$ with the smallest cost, C.

Problem 3.2 (m-TSP)**.** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$ be a complete weighted graph. For a given $m \geq 2$, find a set of cycles, $\mathcal{C} = \{c_1, \ldots, c_m\}$, on $\mathcal{G}$ which minimizes C as defined in* (3.4)*.*

The MPP and m-TSP are equivalent problems. A solution to the MPP can be converted to a solution to the m-TSP by solving the 1-TSP on each subgraph of the partition. A solution to the m-TSP can be converted to a solution to the MPP by defining each subgraph by the vertices of a single cycle of the m-TSP solution. For the remainder of the chapter, all cycles are spanning (on their subgraph) and I will therefore refer to spanning cycles simply as cycles.

Implicit in the definitions I've given for both problems is that the robots' paths are all cycles. Having a cyclic path means that the robot starts and ends at the same task (although it only performs the task once). This assumption is not very realistic! In most situations robots have to start and end in the specific locations—usually starting at its current location and ending at some sort of depot or charging station. In many cases, all robots start and end at the same depot, and so a single "task" has to be part of each robot's path, which is the only case where a single task should assigned to more than one robot. Although the difference between a cycle and a path with fixed endpoints may seem significant, it is actually very easy to modify the solver based on these kinds of constraints. To simplify explanations,

I will initially explain my approach using cyclic paths with each task assigned to exactly one robot. Then in Section 3.8, I will describe the few modifications necessary to extend the approach to these more realistic depot situations.

3.3 A proxy for minimum cycle length

My plan is to solve the task allocation problem by solving the MPP. This problem is NP-hard, so I will have to use a heuristic to solve it. A typical heuristic would look something like:

1. Choose a random partition, $\mathcal{P}$.
2. Apply a local transformation (e.g. moving one task from one subgraph to another subgraph) to $\mathcal{P}$ to get a similar partition, $\mathcal{P}'$.
3. If $C_{\min}(\mathcal{P}') < C_{\min}(\mathcal{P})$, set $\mathcal{P} = \mathcal{P}'$.
4. Repeat steps 2–3 until the transformation can no longer improve $\mathcal{P}$.

Although this type of heuristic gets used for all kinds of combinatorial problems, there is a fatal flaw here. In step 3, we need to compare $C_{\min}(\mathcal{P})$ with $C_{\min}(\mathcal{P}')$ but computing $C_{\min}$ involves solving the 1-TSP which is NP-hard!

Right away, this whole idea of solving the task allocation problem as a partition problem (the MPP) instead of a routing problem (the m-TSP) seems like a bad idea. Originally, we had an NP-hard problem, but now my idea for a heuristic involves solving multiple NP-hard problems in every single round of the heuristic! The way out of this problem is to solve a slightly different problem instead of Problem 3.1. Instead of using the NP-hard $C_{\min}$ as the cost function, we'll use a different cost function which is easier to compute.

Problem 3.3 (Average Partition Problem (APP))**.** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a complete weighted graph. For a given $m \geq 2$, find a partition, $\mathcal{P} = \{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$, of $\mathcal{G}$ which*

minimizes

$$C_{\text{avg}}(\mathcal{P}) = \max_{\mathcal{G}_i \in \mathcal{P}} \{S_{\text{avg}}(\mathcal{G}_i)\} \tag{3.5}$$

where $S_{\text{avg}}(\mathcal{G}_i)$ *is the average length of a cycle on the subgraph,* $\mathcal{G}_i$*.*

The only difference between the APP and MPP is that the APP uses the average cycle length in its cost function instead of the minimum cycle length (Figure 3.2). The average cycle length can be computed in quadratic time. Since $\mathcal{G}_i$ is a complete subgraph, each of its edges is equally likely to appear in a cycle. There are $|\mathcal{E}_i| = \frac{n_i(n_i-1)}{2}$ edges in $\mathcal{G}_i$ and n_i edges in a cycle so

$$S_{\text{avg}}(\mathcal{G}_i) = n_i \sum_{e \in \mathcal{E}_i} \frac{2}{n_i(n_i-1)} w(e) = \frac{2}{n_i - 1} \sum_{e \in \mathcal{E}_i} w(e). \tag{3.6}$$

Using this formula, $S_{\text{avg}}(\mathcal{G}_i)$ can be computed in $\mathcal{O}(n_i^2)$. Computing $C_{\text{avg}}(\mathcal{P})$ requires computation of $S_{\text{avg}}(\mathcal{G}_i)$ for all m subgraphs of $\mathcal{P}$. As $\sum_{i=1}^{m} n_i^2 < n^2$, it can be computed in $\mathcal{O}(n^2)$.

The average size of a subgraph as defined in (3.6) is quite similar to the utility functions defined as sums of utilities of subsets of tasks which are used in other task allocation approaches [146]. The cost of a set of tasks, $\mathcal{V}_i$, is simply the sum of costs over pairs of tasks with the cost of a pair of tasks equal to $\frac{2}{n_i-1} w((v_0, v_1))$. In contrast, the minimum size in (3.2) cannot be expressed as the sum of costs of subsets of tasks as it only depends on the cost of some pairs of tasks and not on all pairs of tasks.

3.3.1 Is C_{avg} a good proxy for C_{min}?

The idea is to use the solution to Problem 3.3 as if it is a solution to Problem 3.1. This idea uses the average cycle length as a proxy for the minimum cycle length. At first, it may seem that the average length is a terrible proxy for the minimum length. The average cycle is much longer than the shortest cycle, and these lengths

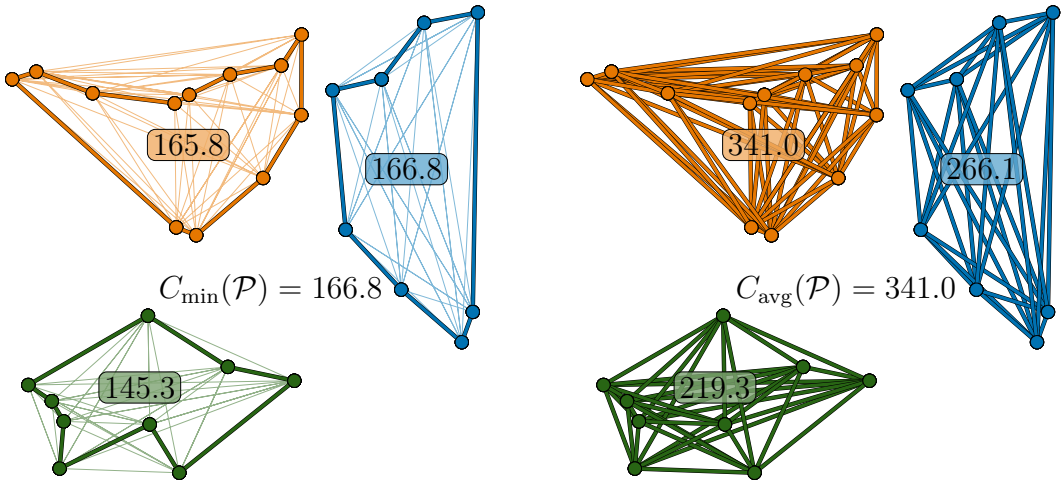


Figure 3.2: The MPP and APP use different cost functions for the same partition. Both problems define the cost of the partition as the size of the largest subgraph, but the size is defined differently. The MPP's size function uses the length of the shortest cycle on the subgraph which only depends on some edges of the graph. The APP's size function uses the average length of all cycles on the subgraph which gives equal weight to every edge of the graph.

are definitely not proportional. For the three subgraphs in Figure 3.2, the ratios of $\frac{S_{\text{avg}}}{S_{\min}}$ are 1.51, 1.60, and 2.06.

Ultimately, it doesn't matter if the two lengths are equal or proportional at all, as long as they can both be used to get similar solutions to the partition problems. In step 3 of my original idea for a heuristic for the MPP, we have to check whether $C(\mathcal{P}') < C(\mathcal{P})$ but the actual values of these costs don't matter. Therefore any cost function which would order the same partitions the same way is equally good. This property is just monotonicity! If C_{avg} is related to $C_{\min}$ by some monotonically increasing function, then the heuristic would behave identically with either cost so it is a good proxy.

The reason why we only need to worry about monotonicity, is because monotonically increasing functions distribute over the $\min\{\cdot\}$ operator. If $f : \mathbb{R} \to \mathbb{R}$ is

monotonically increasing then

$$f\left(\min\{x_0, \ldots, x_n\}\right) = \min\{f(x_0), \ldots, f(x_n)\}.$$

Similarly, monotonically increasing functions distribute over the max operator, which relates the cost of a partition to the sizes of its subgraphs. Therefore, if the relationship between S_{avg} and $S_{\min}$ is monotonic, this property will transfer to the relationship between C_{avg} and $C_{\min}$. Therefore we need to check that if $\mathcal{G}_i$ and $\mathcal{G}_j$ are two subgraphs of $\mathcal{G}$ then $S_{\text{avg}}(\mathcal{G}_i) < S_{\text{avg}}(\mathcal{G}_j)$ if and only if $S_{\min}(\mathcal{G}_i) < S_{\min}(\mathcal{G}_j)$.

In general, the average and minimum sizes are not related by a *strictly* monotonically increasing function. However, for many different types of graphs (Figure 3.3) the two measurements are approximately monotonic as their relationship can be expressed by

$$S_{\min}(\mathcal{G}_i) = f(S_{\text{avg}}(\mathcal{G}_i)) + \nu \tag{3.7}$$

where $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is a monotonically increasing function and ν is zero-mean noise. For any $\alpha \in (0, 0.5)$, let $B_\alpha^-, B_\alpha^+ \in \mathbb{R}_{>0}$ be defined such that

$$\mathbb{P}\left[\nu \geq -B_\alpha^-\right] = \mathbb{P}\left[\nu \leq +B_\alpha^+\right] = 1 - \alpha.$$

For $\alpha = 0.025$, $[-B_\alpha^-, B_\alpha^+]$ is the 95% confidence interval for ν. I will use these quantiles to establish that solutions to the APP are good solutions for the MPP. Lemma 3.1 establishes a relationship between $C_{\min}$ and C_{avg} for partitions which is equivalent to the relationship (3.7) for subgraphs. Finally, I use this relationship between the costs of partitions to prove that the partition that minimizes $C_{\min}$, and hence solves the APP, tends to have a low C_{avg} and is thus a good solution for the MPP (Theorem 3.1 and Figure 3.5). Although these probabilistic results are complex, the analogous result when $\nu = 0$ (Corollary 3.2) is very simple.

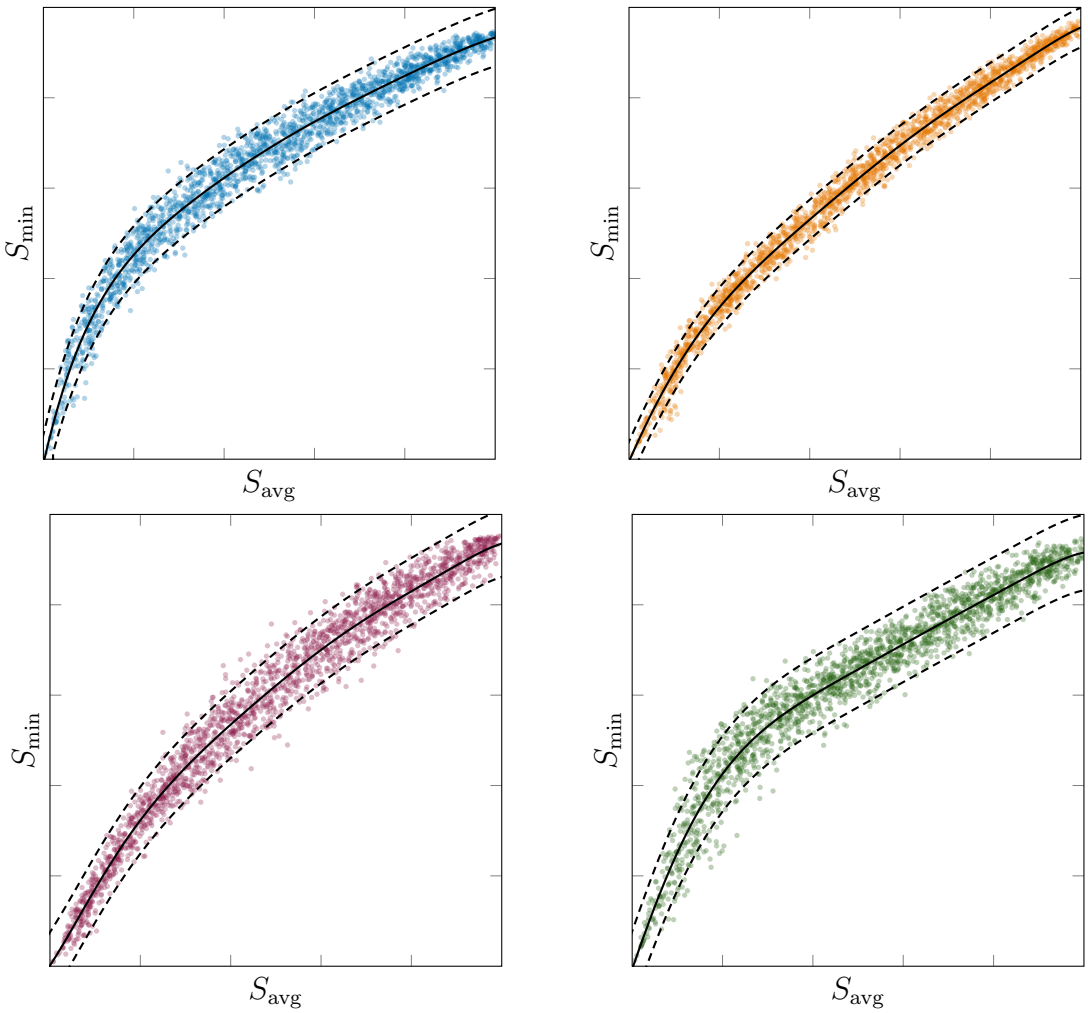


Figure 3.3: Minimum vs average cycle lengths of randomly sampled subgraphs of a 2D Euclidean graph (top left), a 3D Euclidean graph (top right), an Erdős-Rényi random graph with weights from an exponential distribution with parameter 1 (bottom left), and the graph of the lower 48 US capitals (bottom right). Dots represents the average and estimated minimum cycle lengths (approximated using Concorde [42]) of a randomly sampled subgraph, the solid line is the mean relationship between S_{avg} and $S_{\min}$, and the dashed lines are the 2.5% and 97.5% quantiles.

Lemma 3.1. *Suppose that $\mathcal{G}$ is such that (3.7) holds, then*

$$\mathbb{P}\left[C_{\min}(\mathcal{P}) \geq f(C_{\text{avg}}(\mathcal{P})) - B_\alpha^-\right] \geq 1-\alpha$$
$$\mathbb{P}\left[C_{\min}(\mathcal{P}) \leq f(C_{\text{avg}}(\mathcal{P})) + B_\alpha^+\right] \geq 1-\alpha$$

for any partition $\mathcal{P}$ of $\mathcal{G}$.

Proof. Some of the quantities used in this proof are colored according to Figure 3.4 to aid understanding. Let $\mathcal{G}^*_{\min}, \mathcal{G}^*_{\text{avg}} \in \mathcal{P}$ be the subgraphs which maximize $S_{\min}$ and S_{avg}, respectively. By the definitions of $C_{\min}$ and C_{avg}, we have $C_{\min}(\mathcal{P}) = S_{\min}(\mathcal{G}^*_{\min})$ and $C_{\text{avg}}(\mathcal{P}) = S_{\text{avg}}(\mathcal{G}^*_{\text{avg}})$ so

$$\mathbb{P}\left[C_{\min}(\mathcal{P}) \leq f(C_{\text{avg}}(\mathcal{P})) + B_\alpha^+\right] = \mathbb{P}\left[S_{\min}(\mathcal{G}^*_{\min}) \leq f(S_{\text{avg}}(\mathcal{G}^*_{\text{avg}})) + B_\alpha^+\right].$$

Since $\mathcal{G}^*_{\text{avg}}$ maximizes S_{avg}, it is always true that $S_{\text{avg}}(\mathcal{G}^*_{\min}) \leq S_{\text{avg}}(\mathcal{G}^*_{\text{avg}}) = C_{\text{avg}}(\mathcal{P})$. The monotonicity of f preserves the inequality so $f(S_{\text{avg}}(\mathcal{G}^*_{\min})) \leq f(C_{\text{avg}}(\mathcal{P}))$ and therefore $f(S_{\text{avg}}(\mathcal{G}^*_{\min})) + B_\alpha^+ \leq f(C_{\text{avg}}(\mathcal{P})) + B_\alpha^+$. If $S_{\min}(\mathcal{G}^*_{\min}) \leq f(S_{\text{avg}}(\mathcal{G}^*_{\min})) + B_\alpha^+$ then $S_{\min}(\mathcal{G}^*_{\min}) \leq f(S_{\text{avg}}(\mathcal{G}^*_{\text{avg}})) + B_\alpha^+$ and therefore

$$\mathbb{P}\left[C_{\min}(\mathcal{P}) \leq f(C_{\text{avg}}(\mathcal{P})) + B_\alpha^+\right] \geq \mathbb{P}\left[S_{\min}(\mathcal{G}^*_{\min}) \leq f(S_{\text{avg}}(\mathcal{G}^*_{\min})) + B_\alpha^+\right].$$

Since (3.7) holds the right hand side of this equation is at least $1-\alpha$ which proves the second inequality.

The first inequality can be proven similarly due to the symmetry of the problem with respect to B_α^- and B_α^+. Using the definitions of $\mathcal{G}^*_{\min}$ and $\mathcal{G}^*_{\text{avg}}$, we have

$$\mathbb{P}\left[C_{\min}(\mathcal{P}) \geq f(C_{\text{avg}}(\mathcal{P})) - B_\alpha^-\right] = \mathbb{P}\left[S_{\min}(\mathcal{G}^*_{\min}) \geq f(S_{\text{avg}}(\mathcal{G}^*_{\text{avg}})) - B_\alpha^-\right].$$

Since $\mathcal{G}^*_{\min}$ maximizes $S_{\min}$, it is always true that $S_{\min}(\mathcal{G}^*_{\min}) \geq S_{\min}(\mathcal{G}^*_{\text{avg}})$. If

$S_{\min}(\mathcal{G}^*_{\text{avg}}) \geq f(S_{\text{avg}}(\mathcal{G}^*_{\text{avg}})) - B^-_\alpha$ then $S_{\min}(\mathcal{G}^*_{\min}) \geq f(S_{\text{avg}}(\mathcal{G}^*_{\text{avg}})) - B^-_\alpha$ and therefore

$$\mathbb{P}\left[C_{\min}(\mathcal{P}) \geq f(C_{\text{avg}}(\mathcal{P})) - B^-_\alpha\right] \geq \mathbb{P}\left[S_{\min}(\mathcal{G}^*_{\text{avg}}) \geq f(S_{\text{avg}}(\mathcal{G}^*_{\text{avg}})) - B^-_\alpha\right].$$

Since (3.7) holds the right hand side of this equation is at least $1-\alpha$ which proves the first inequality. □

Corollary 3.1. *Suppose there exists a graph, $\mathcal{G}$, and monotonically increasing function, $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, such that $S_{\min}(\mathcal{G}_i) = f(S_{\text{avg}}(\mathcal{G}_i))$ for every subgraph, $\mathcal{G}_i$, of $\mathcal{G}$ (i.e. (3.7) holds with $\nu = 0$). Then $C_{\min}(\mathcal{P}) = f(C_{\text{avg}}(\mathcal{P}))$ for every partition, $\mathcal{P}$, of $\mathcal{G}$.*

Theorem 3.1. *Suppose that $\mathcal{G}$ is such that (3.7) holds, then*

$$\mathbb{P}\left[C_{\min}(\mathcal{P}^*_{\text{avg}}) \leq C_{\min}(\mathcal{P}^*_{\min}) + B^-_\alpha + B^+_\alpha\right] \geq (1-\alpha)^2$$

where $\mathcal{P}^*_{\text{avg}}$ and $\mathcal{P}^*_{\min}$ are the partitions that minimize C_{avg} and $C_{\min}$ as defined in (3.5) and (3.3).

Proof. Some quantities used in this proof are colored according to Figure 3.5 to aid understanding. First, we define the events:

$$\mathsf{A} : C_{\min}(\mathcal{P}^*_{\text{avg}}) \leq C_{\min}(\mathcal{P}^*_{\min}) + B^-_\alpha + B^+_\alpha$$
$$\mathsf{B} : C_{\min}(\mathcal{P}^*_{\text{avg}}) \leq f(C_{\text{avg}}(\mathcal{P}^*_{\text{avg}})) + B^+_\alpha$$
$$\mathsf{C} : C_{\min}(\mathcal{P}^*_{\min}) \geq f(C_{\text{avg}}(\mathcal{P}^*_{\min})) - B^-_\alpha.$$

Since $\mathcal{P}^*_{\text{avg}}$ minimizes C_{avg}, it is always true that $C_{\text{avg}}(\mathcal{P}^*_{\text{avg}}) \leq C_{\text{avg}}(\mathcal{P}^*_{\min})$. The monotonicity of f preserves the inequality so $f(C_{\text{avg}}(\mathcal{P}^*_{\text{avg}})) \leq f(C_{\text{avg}}(\mathcal{P}^*_{\min}))$. If B

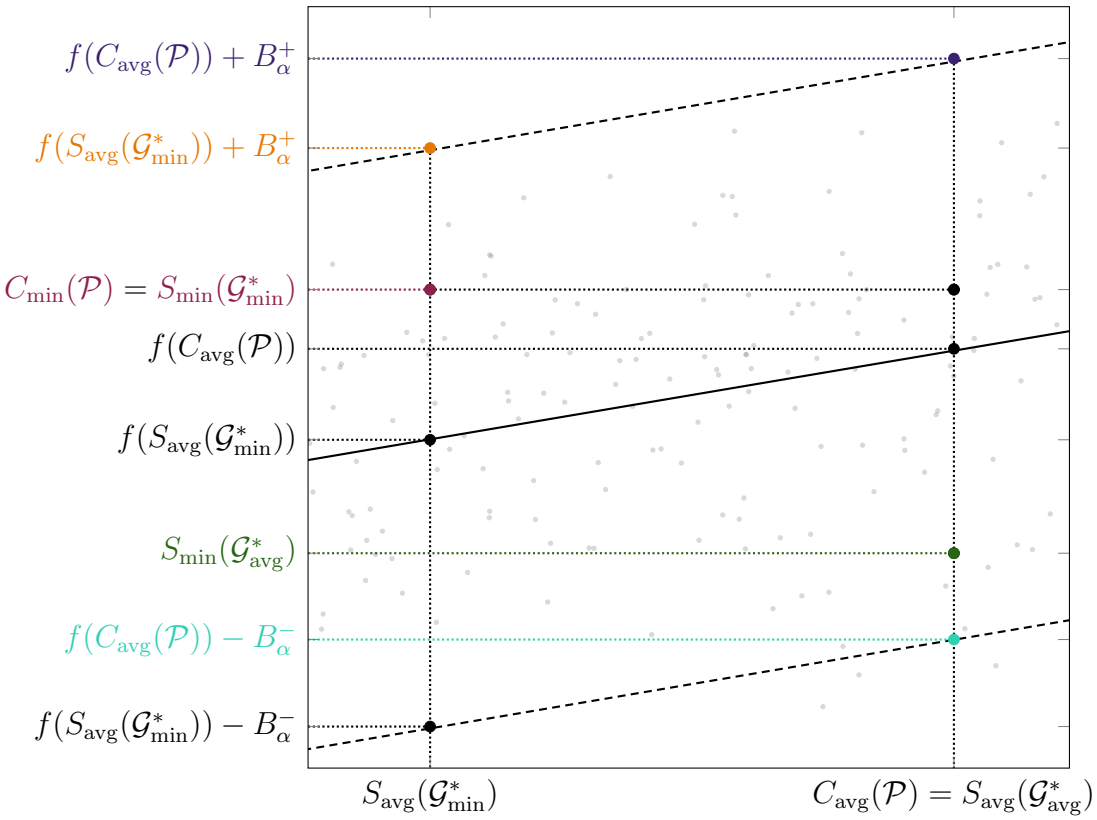


Figure 3.4: Magnified section of the bottom right graph of Figure 3.3 showing quantities involved in the proof of Lemma 3.1. Light gray dots represent $(S_{\text{avg}}(\mathcal{G}_i), S_{\text{min}}(\mathcal{G}_i))$ for subgraphs of `att48`, the graph of the lower 48 US capitals. The green and red dots represent two subgraphs which together form a partition $\mathcal{P} = \{\mathcal{G}^*_{\text{avg}}, \mathcal{G}^*_{\text{min}}\}$ of $\mathcal{G}$. The large black dot represents $(C_{\text{avg}}(\mathcal{P}), C_{\text{min}}(\mathcal{P}))$ for this partition.

holds then

$$C_{\text{min}}(\mathcal{P}^*_{\text{avg}}) \leq f(C_{\text{avg}}(\mathcal{P}^*_{\text{avg}})) + B_\alpha^+ \leq f(C_{\text{avg}}(\mathcal{P}^*_{\text{min}})) + B_\alpha^+.$$

If C also holds, then

$$\begin{aligned} C_{\text{min}}(\mathcal{P}^*_{\text{avg}}) &\leq f(C_{\text{avg}}(\mathcal{P}^*_{\text{min}})) - B_\alpha^- + B_\alpha^- + B_\alpha^+ \\ &\leq C_{\text{min}}(\mathcal{P}^*_{\text{min}}) + B_\alpha^- + B_\alpha^+ \\ &= C_{\text{min}}(\mathcal{P}^*_{\text{min}}) + B_\alpha^- + B_\alpha^+. \end{aligned}$$

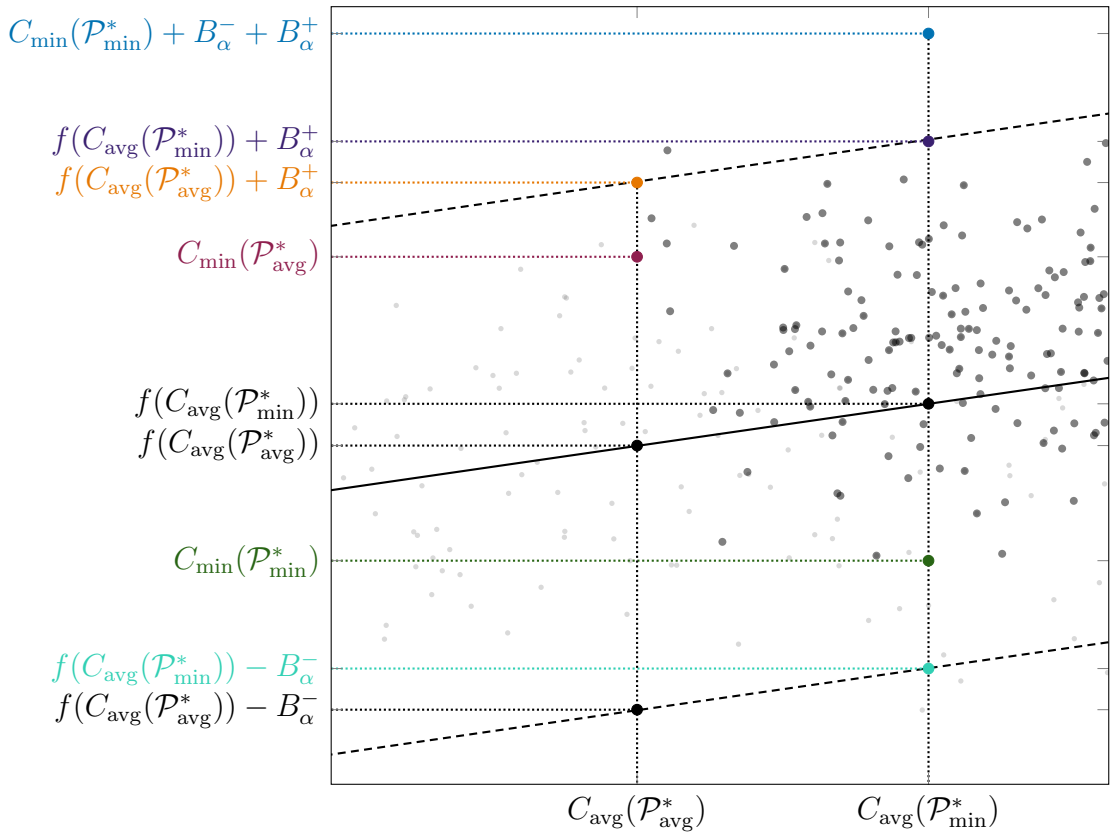


Figure 3.5: Magnified section of the bottom right graph of Figure 3.3 showing quantities involved in the proof of Theorem 3.1. Light gray dots represent $(S_{\text{avg}}(\mathcal{G}_i), S_{\min}(\mathcal{G}_i))$ for subgraphs $\mathcal{G}$. Dark gray dots represent $(C_{\text{avg}}(\mathcal{P}), C_{\min}(\mathcal{P}))$ for partitions of $\mathcal{G}$ into two subgraphs. The red and green dots represent the partitions which minimize C_{avg} and $C_{\min}$.

Therefore $\mathsf{B} \cap \mathsf{C} \Rightarrow \mathsf{A}$ so $\mathbb{P}[\mathsf{A}] \geq \mathbb{P}[\mathsf{B} \cap \mathsf{C}]$. As B and C depend on different variables ($\mathcal{P}^*_{\text{avg}}$ and $\mathcal{P}^*_{\min}$), they are independent so $\mathbb{P}[\mathsf{A}] \geq \mathbb{P}[\mathsf{B}] \times \mathbb{P}[\mathsf{C}]$. By Lemma 3.1, the probabilities of B and C are both at least $(1-\alpha)$ and therefore $\mathbb{P}[\mathsf{A}] \geq (1-\alpha)^2$. □

Corollary 3.2. *Suppose there exists a graph, $\mathcal{G}$, and monotonically increasing function, $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, such that $S_{\min}(\mathcal{G}_i) = f(S_{\text{avg}}(\mathcal{G}_i))$ for every subgraph, $\mathcal{G}_i$, of $\mathcal{G}$ (i.e. (3.7) holds with $\nu = 0$). Then $\mathcal{P}^*_{\text{avg}} = \mathcal{P}^*_{\min}$ and so the solution to the APP also solves the MPP.*

Theorem 3.1 establishes that C_{avg} is a good proxy for $C_{\min}$ when (3.7) holds. Experimental evidence suggests (3.7) holds for many common graphs (Figure 3.3).

This result motivates us to use a solution to the APP as a proxy for the solutions to the MPP when developing a task allocation heuristic. As the APP can itself be viewed as a heuristic approximation of the MPP, this approach enables us to find good solutions to the MPP by solving the APP, avoiding the problem of evaluating the MPP's cost function which is NP-hard.

Although I have applied a monotonic proxy to the MPP, it is a general technique that could be applied to other optimization problems. Since monotonically increasing functions commute with the max{} and min{} operators, the minimizer of an arbitrary cost function $J(\cdot)$ is also the minimizer of $f(J(\cdot))$ for any monotonically increasing f. This relationship even holds true if the exact form of f is not known! For other optimization problems where $J(\cdot)$ is difficult to compute but $f(J(\cdot))$ is easy to compute, a similar proxy could be very useful. This type of proxy is especially useful for minmax (or maxmin) problems since the monotonic function commutes with both max{} and min{}.

3.3.2 Hardness of the APP

My task allocation heuristic relies on solutions to the APP. Although the APP's cost function can be computed in polynomial time, the overall problem is still NP-hard so we will develop a heuristic for the APP instead of solving it exactly.

Theorem 3.2. *The APP is NP-hard.*

Proof. I will prove that the APP is NP-hard by reducing the known NP-hard number partition problem (NPP) [66] to it. The NPP asks "Given a multiset of positive integers $\mathcal{Z}$ with even sum K, does there exist a partition $\{\mathcal{Z}_1, \mathcal{Z}_2\}$ where $\mathcal{Z}_1$ and $\mathcal{Z}_2$ both sum to $\frac{K}{2}$?" I will reduce this problem to a decision version of the APP which asks "Given a complete graph, $\mathcal{G}$, with positive weights, does there exist a partition, $\{\mathcal{G}_1, \mathcal{G}_2\}$, such that the average cycle length on each subgraph is equal to L?"

For any instance $(\mathcal{Z}, K)$ of the NPP, we can construct an instance, $(\mathcal{G}, L)$, of the APP (Figure 3.6). Let $L = K$, $n = |\mathcal{Z}|$, and $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$ be a complete weighted graph with $|\mathcal{V}| = n$. Each $v_i \in \mathcal{V}$ corresponds to a $z_i \in \mathcal{Z}$. The weight of an edge $e = (v_i, v_j)$ is defined as $w(e) = z_i + z_j$. This reduction can be performed in quadratic time and can be applied to any instance of the NPP.

Next, we show that the NPP defined by $(\mathcal{Z}, K)$ is true if and only if the decision version of the APP defined by $(\mathcal{G}, L)$ is true. Let $\mathcal{J}_1$ be a subset of $\{1, \ldots, n\}$. It corresponds to a multiset of integers, $\mathcal{Z}_1$, and a set of vertices, $\mathcal{V}_1$, defined by

$$\mathcal{Z}_1 = \{z_j \in \mathcal{Z} \mid j \in \mathcal{J}_1\},$$
$$\mathcal{V}_1 = \{v_j \in \mathcal{V} \mid j \in \mathcal{J}_1\}.$$

It also corresponds to $\mathcal{G}_1$, a subgraph of $\mathcal{G}$ induced by $\mathcal{V}_1$. Using this definition, every subgraph of $\mathcal{G}$ corresponds to a unique subset of $\mathcal{Z}$ and vice versa.

Every cycle, p, on $\mathcal{G}_1$ corresponds to a permutation, $(j_1, \ldots, j_{n_1})$, of $\mathcal{J}$. This permutation corresponds to cycle vertices, $v_{j_1}, v_{j_2}, \ldots, v_{j_{n_1}}, v_{j_1}$, and edges, $e_{j_i} = (v_{j_i}, v_{j_{i+1}})$, where we define $j_{n_1+1} = j_1$. The length of this cycle is

$$\ell(p) = \sum_{i=1}^{n_1} w(e_{j_i}) = \sum_{i=1}^{n_1} (z_{j_i} + z_{j_{i+1}}) = \sum_{i=1}^{n_1} z_{j_i} + \sum_{i=2}^{n_1+1} z_{j_i}.$$

Since $j_{i+1} = j_1$, the two sums are equal so

$$\ell(p) = 2 \sum_{i=1}^{n_1} z_{j_i} = 2 \sum_{z \in \mathcal{Z}_1} z.$$

The length of the cycle does not depend on the order of the vertices, so all cycles have the same length and so the average cycle length is

$$S_{\text{avg}}(\mathcal{G}_1) = 2 \sum_{z \in \mathcal{Z}_1} z.$$

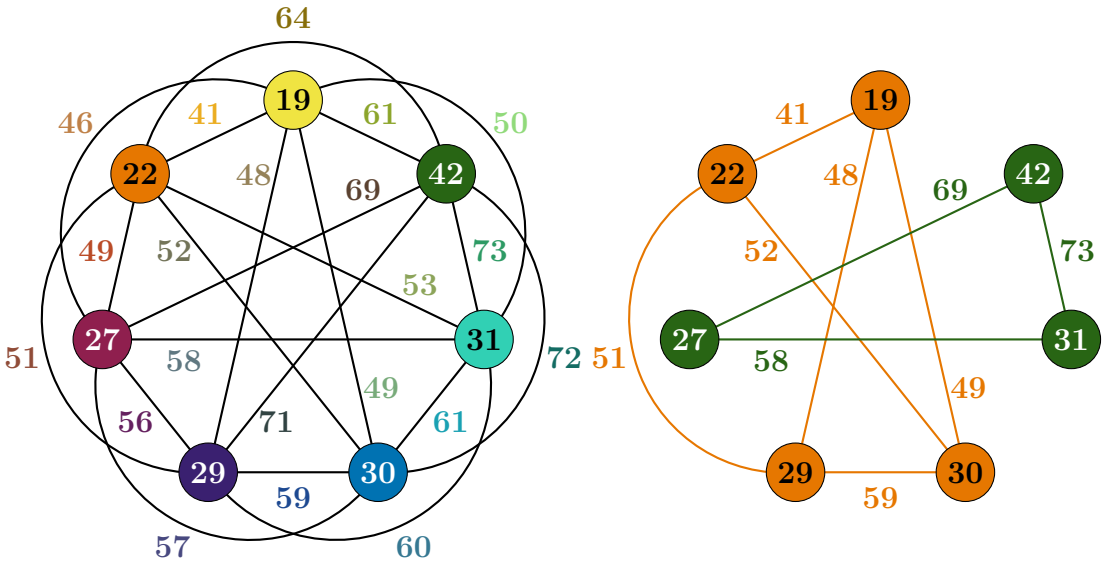


Figure 3.6: Example of a reduction of the NPP with $\mathcal{Z} = \{19, 22, 27, 29, 30, 31, 42\}$ and $K = 200$ to an instance of the APP (left). The solution of the APP (right) consists of two subgraphs which both have $S_{\text{avg}}(\mathcal{G}_i) = 200$. This solution corresponds to the solution $\mathcal{Z}_1 = \{19, 22, 29, 30\}$ and $\mathcal{Z}_2 = \{27, 31, 42\}$ which both sum to $\frac{K}{2} = 100$.

From $\mathcal{J}_1$, we can define $\mathcal{J}_2 = \{1, \ldots, n\} \setminus \mathcal{J}_1$ and the corresponding $\mathcal{Z}_2$ and $\mathcal{G}_2$. $\{\mathcal{Z}_1, \mathcal{Z}_2\}$ is a partition of $\mathcal{Z}$ and $\{\mathcal{G}_1, \mathcal{G}_2\}$ is a partition of $\mathcal{G}$. As $\mathcal{J}_2$ is also a subset of $\{1, \ldots, n\}$, it is also true that $S_{\text{avg}}(\mathcal{G}_2)$ equals the sum of $\mathcal{Z}_2$.

Suppose the NPP is true. Then there exists a partition, $\{\mathcal{Z}_1, \mathcal{Z}_2\}$, of $\mathcal{Z}$ which both sum to $\frac{K}{2}$. This partition corresponds to a partition, $\{\mathcal{G}_1, \mathcal{G}_2\}$, of $\mathcal{G}$. Both $S_{\text{avg}}(\mathcal{G}_1)$ and $S_{\text{avg}}(\mathcal{G}_2)$ are equal to the twice sums of $\mathcal{Z}_1$ and $\mathcal{Z}_2$ which equals $L = 2\frac{K}{2}$ so the APP is true. Similarly, if the APP is true, then there exists a partition, $\{\mathcal{G}_1, \mathcal{G}_2\}$, of $\mathcal{G}$ with $S_{\text{avg}}(\mathcal{G}_1)$ and $S_{\text{avg}}(\mathcal{G}_2)$ equal to L so in the corresponding $\{\mathcal{Z}_1, \mathcal{Z}_2\}$, both subsets sum to $\frac{K}{2} = \frac{L}{2}$ and the NPP is true. Therefore the NPP can be reduced to the APP and since the NPP is NP-hard, the APP must also be NP-hard. □

Corollary 3.3. *The MPP is NP-hard.*

Proof. We can prove this theorem using the same reduction from the NPP that

was used in the proof of Theorem 3.2. By the construction of w, all cycles on $\mathcal{G}_1$ were shown to have the same length so $S_{\min}(\mathcal{G}_1) = S_{\text{avg}}(\mathcal{G}_1)$. This equivalence makes the APP and MPP equivalent on the graphs defined in the reduction, so the reduction is valid for either problem. This result establishes that the MPP would be NP-hard even if $S_{\min}(\mathcal{G}_1)$ could be computed in polynomial time. □

3.4 A task allocation heuristic based on the APP

As the APP is NP-hard, I designed a novel heuristic for solving it (Algorithm 3.3). It consists of two alternating phases—improvement (Algorithm 3.1) and transferring outliers (Algorithm 3.2)—which modify an initial partition and create a near optimal solution to the APP. This solution is then used as a proxy for a solution to the MPP. My partitioning heuristic is explicitly based on a minmax criterion and only depends on the value of the edge weight function. Unlike other approaches, such as k-means clustering [144], it can be used on non-Euclidean graphs which are important in real-world problems where travel times are not proportional to as-the-crow-flies distances.

The two phases of my algorithm are used to find and escape local minima. The improvement phase (Subsection 3.4.1) performs a sequence of local moves (transfers and swaps) which each reduce the minmax cost for a pair of subgraphs. The result of this phase is a partition which is a local minimum of the minmax cost function with respect to the transfer and swap moves. For a given problem, there may be many partitions that are local minima. The local minimum computed by Algorithm 3.1 depends heavily on the initial random partition and the random order that we check possible transfers and swaps.

Rather than simply return the first local minimum we find, we use the outlier transfer phase (Subsection 3.4.2) to search for other nearby local minima. In this phase we identify some *outliers* which have a high individual contribution to

their subgraph's size relative to other vertices in that subgraph. These outliers get transferred to another subgraph despite increasing the minmax cost of the partition. This transfer results in a different partition which is not a local minimum any may be in the region of attraction of a different local minimum. We can then perform Algorithm 3.1 again to find this new local minimum. Transferring outliers is guaranteed to reduce the *minsum* cost, which is correlated with the minmax cost, so this new local minimum may have a better minmax cost than the previous local minimum. The combination of these two phases finds multiple local minimum and terminates when it finds a local minimum whose cost is the same, or worse, than the previous local minimum. Although the initial local minimum found by a single iteration of Algorithm 3.1 is quite sensitive to the initial random partition, the overall algorithm finds multiple local minima and is much less sensitive to the initial partition.

3.4.1 Improvement through transfers and swaps

The improvement phase (Algorithm 3.1) transfers and swaps vertices between pairs of subgraphs to decrease their maximum size. This algorithm improves a partition until it is a local minimizer of C_{avg} with respect to the transfer and swap operations. Algorithm 3.1 requires computation of $S_{\text{avg}}(\mathcal{G}_i)$ for many graphs which, if computed from its definition (3.6), would take $\mathcal{O}(n_i^2)$. However, if we want to compute S_{avg} for $\mathcal{G}_i$ with one vertex added or removed, we can use $S_{\text{avg}}(\mathcal{G}_i)$ to speed up computation. This quick update of S_{avg} uses the total edge weight, $W(\mathcal{G}_i)$, and the marginal edge weight, $\Delta W(\mathcal{G}_i, v)$, which are defined by

$$W(\mathcal{G}_i) = \sum_{e \in \mathcal{E}_i} w(e) \tag{3.8}$$

$$\Delta W(\mathcal{G}_i, v) = \sum_{v' \in \mathcal{V}_i} w((v, v')). \tag{3.9}$$

Combining (3.6) and (3.8), a subgraph's size can be written as

$$S_{\text{avg}}(\mathcal{G}_i) = \frac{2}{n_i - 1} W(\mathcal{G}_i). \tag{3.10}$$

The marginal edge weights will be used to efficiently update $W(\mathcal{G}_i)$ and $S_{\text{avg}}(\mathcal{G}_i)$ when a vertex is added or removed from the graph.

Transfers are the simplest modification of a partition. A transfer consists of a single vertex $v \in \mathcal{V}_i$ moving from $\mathcal{G}_i$ to $\mathcal{G}_j$. After a transfer (Figure 3.7), the new subgraphs $\mathcal{G}_i'$ and $\mathcal{G}_j'$ have sizes

$$S_{\text{avg}}(\mathcal{G}_i') = \frac{2}{n_i - 2}\Big(W(\mathcal{G}_i) - \Delta W(\mathcal{G}_i, v)\Big) \tag{3.11}$$

$$S_{\text{avg}}(\mathcal{G}_j') = \frac{2}{n_j - 0}\Big(W(\mathcal{G}_j) + \Delta W(\mathcal{G}_j, v)\Big). \tag{3.12}$$

If, for a given partition, we have precomputed $\Delta W(\cdot, \cdot)$ for all pairs for subgraphs and vertices, then these equations let us compute $S_{\text{avg}}(\mathcal{G}_i')$ and $S_{\text{avg}}(\mathcal{G}_j')$ in $\mathcal{O}(1)$. As there are n vertices which could be transferred, it is possible to check how every single potential transfer would affect the sizes of subgraphs in $\mathcal{O}(n)$. After checking all potential transfers, we can choose the best one (if one causes the maximum subgraph size to decrease) and implement it. After performing the transfer, $\mathcal{G}_i$ and $\mathcal{G}_j$ change so $\Delta W(\mathcal{G}_i, v')$ and $\Delta W(\mathcal{G}_j, v')$ must be updated for every vertex $v' \in \mathcal{V}$. After transferring v from $\mathcal{G}_i$ to $\mathcal{G}_j$, the updated marginal edge weights are

$$\Delta W(\mathcal{G}_i', v') = \Delta W(\mathcal{G}_i, v') - w\big((v, v')\big) \tag{3.13}$$

$$\Delta W(\mathcal{G}_j', v') = \Delta W(\mathcal{G}_j, v') + w\big((v, v')\big) \tag{3.14}$$

for all $v' \in \mathcal{V}$. Using these equations, each marginal weight is updated in $\mathcal{O}(1)$ and all the marginal weights are updated in $\mathcal{O}(n)$. By storing the nm marginal sizes,

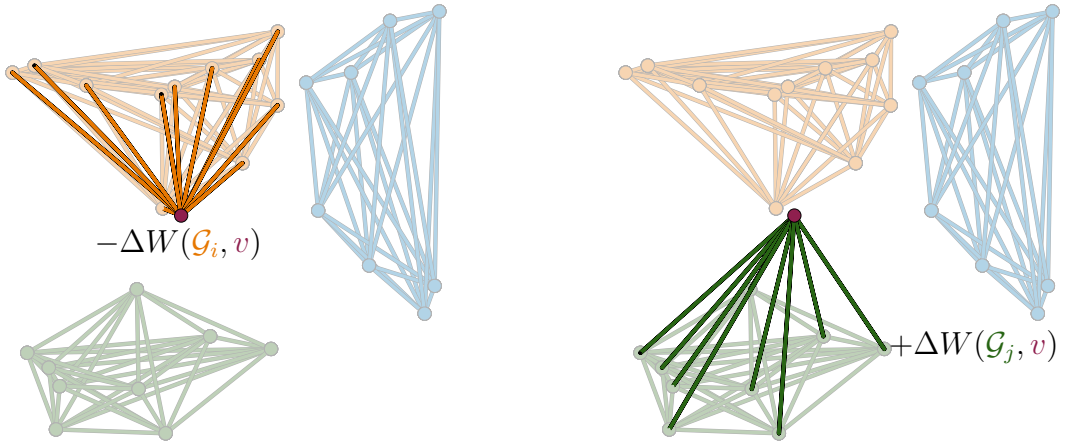


Figure 3.7: When transferring a vertex, v from $\mathcal{G}_i$ to $\mathcal{G}_j$, the sizes of both subgraphs change. The size of $\mathcal{G}_i$ decreases (left) according to (3.11) by subtracting $\Delta W(\mathcal{G}_i, v)$, the weight of all edges between v and vertices of $\mathcal{G}_i$, from $W(\mathcal{G}_i)$, the weight of all edges of $\mathcal{G}_i$. Similarly, the size of $\mathcal{G}_j$ increases (right) according to (3.12) by adding $\Delta W(\mathcal{G}_j, v)$ to $W(\mathcal{G}_j)$.

we can both find the best transfer and implement it in $\mathcal{O}(n)$. As the marginal edge weights are always positive, a transfer is guaranteed to reduce $S_{\text{avg}}(\mathcal{G}_i)$—usually by a large amount—while increasing $S_{\text{avg}}(\mathcal{G}_j)$. As they increase $S_{\text{avg}}(\mathcal{G}_j)$, transfers are the most useful when $S_{\text{avg}}(\mathcal{G}_i) \gg S_{\text{avg}}(\mathcal{G}_j)$.

When $S_{\text{avg}}(\mathcal{G}_i)$ and $S_{\text{avg}}(\mathcal{G}_j)$ are nearly equal, there are often no good transfers because $S_{\text{avg}}(\mathcal{G}_j)$ will increase too much. In this situation, we can offset the increase in $S_{\text{avg}}(\mathcal{G}_j)$ by simultaneously moving $v' \in \mathcal{V}_j$ from $\mathcal{G}_j$ to $\mathcal{G}_i$. After swapping v and v', we could update the subgraphs' sizes by applying (3.11)–(3.12). Using this approach, we would need to update some ΔW's in between the two updates. Instead, it is more efficient to compute the new sizes in a single step using the formulas

$$S_{\text{avg}}(\mathcal{G}_i') = \frac{2}{n_i - 1}\Big(W(\mathcal{G}_i) - \Delta W(\mathcal{G}_i, v) + \Delta W(\mathcal{G}_i, v') - w\big((v, v')\big)\Big) \tag{3.15}$$

$$S_{\text{avg}}(\mathcal{G}_j') = \frac{2}{n_j - 1}\Big(W(\mathcal{G}_j) + \Delta W(\mathcal{G}_j, v) - \Delta W(\mathcal{G}_j, v') - w\big((v, v')\big)\Big). \tag{3.16}$$

There are n_i potential swaps and $n_i n_j$ potential swaps so finding the best swap

takes is $\mathcal{O}(n_i n_j) \subset \mathcal{O}(n^2)$. After performing a swap, (3.14)–(3.13) can be applied twice for every vertex to update all the marginal edge weights in $\mathcal{O}(n)$. Swaps can improve a partition more than transfers can; however they are more complex moves because there are $n_i n_j$ potential swaps but only n_i potential transfers between a pair of subgraphs. A partition which cannot be improved by any transfer or swap is a local minimum of C_{avg} with respect to these two operations. We could define operations where more than two vertices move simultaneously to reduce the number of local minima; however, the number of possible moves is exponential in the number of vertices.

Algorithm 3.1: Improve partition

Input: Partition, $\mathcal{P} = \{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$
Output: Partition, $\mathcal{P}'$, with $C_{\text{avg}}(\mathcal{P}') \leq C_{\text{avg}}(\mathcal{P})$

```
1  while there are unchecked pairs do
2      (G_1, G_2) ← pair of subgraphs with unchecked transfers
3      if (G_1, G_2) exists then
4          v* ← best vertex to transfer from G_1 to G_2
5          if v* exists then
6              Transfer v* from G_1 to G_2                    /* (3.11)--(3.12) */
7      else
8          (G_1, G_2) ← pair of subgraphs with unchecked swaps
9          if (G_1, G_2) exists then
10             (v_1*, v_2*) ← best pair of vertices to swap between G_1, G_2
11             if (v_1*, v_2*) exists then
12                 Swap v_1* and v_2*                         /* (3.15)--(3.16) */
13     Update subgraph sizes and checked pairs                /* (3.13)--(3.14) */
14 return P' = {G_1, ..., G_m}
```

My improvement heuristic (Algorithm 3.1) repeatedly searches for transfers or swaps of vertices between pairs of subgraphs. In each iteration of the main loop (lines 1–13), it considers either transfers (lines 2–6) or swaps (lines 8–12) between a single pair of subgraphs. There are fewer potential transfers than swaps, so the heuristic prioritizes searching for transfers. We keep track of which pairs of subgraphs have been checked (line 13) and only recheck a pair if one of the

subgraphs has changed. We keep track of which pairs of subgraphs have had transfers and swaps checked using two binary variables, $\chi_{i,j}^{\text{tran}}$ and $\chi_{i,j}^{\text{swap}}$, per pair of subgraphs, $(\mathcal{G}_i, \mathcal{G}_j)$. Initially, all of these variables are set to `false` because no pairs have been checked for transfers or swaps. At the end of an iteration of the loop, these variables are updated (line 13). If a vertex was transferred or swapped, $\mathcal{G}_i$ and $\mathcal{G}_j$ have changed so every $\chi_{k,\ell}^{\text{tran}}$ and $\chi_{k,\ell}^{\text{swap}}$ with either k or ℓ equal to i or j is set to `false`. If no beneficial transfer was found between $\mathcal{G}_i$ and $\mathcal{G}_j$, it sets $\chi_{i,j}^{\text{tran}}$ to `true`; if no beneficial swap was found between $\mathcal{G}_i$ and $\mathcal{G}_j$, it sets $\chi_{i,j}^{\text{swap}}$ to `true`.

The main loop of Algorithm 3.1 starts by selecting an unchecked pair of subgraphs to check for transfers (line 2) or swaps (line 8) between. Although the transfers could be checked before swaps with equivalent performance, we check swaps first because the search space is smaller. Next, it searches for a transfer (line 4) or swap (line 10) which reduces $\max\{S_{\text{avg}}(\mathcal{G}_i), S_{\text{avg}}(\mathcal{G}_j)\}$ for this pair. We can compute the effect of a potential move on $S_{\text{avg}}(\mathcal{G}_i)$ and $S_{\text{avg}}(\mathcal{G}_j)$ in constant time using (3.10)–(3.16). If a move which reduces $\max\{S_{\text{avg}}(\mathcal{G}_i), S_{\text{avg}}(\mathcal{G}_j)\}$ is found, the vertex is transferred (line 6) or the pair of vertices is swapped (line 12) and $S_{\text{avg}}(\mathcal{G}_i)$ and $S_{\text{avg}}(\mathcal{G}_j)$ are updated (line 13). Once transfers and swaps have been checked for all pairs, the main loop terminates and the current partition, which is a local minimum, is returned (line 14). This local minimum (Figure 3.8) is guaranteed to have a lower (or equal) cost than the original partition and also tends to balance the sizes of the subgraphs.

We are able to check the size of subgraphs after a potential transfer or swap using (3.11)–(3.16) if the marginal weights, $\Delta W(\mathcal{G}_i, v)$, are known for all $\mathcal{G}_i \in \mathcal{P}$ and $v \in \mathcal{V}$. Each of these variables can be initially be computed in $\mathcal{O}(n_i)$ using (3.9) and should be included with $\mathcal{P}$ when Algorithm 3.1 is called. As the subgraphs collectively contain $\sum_{i=1}^{m} n_i = n$ vertices, we can initially compute $\Delta W(\mathcal{G}_i, v)$ for a single $v \in \mathcal{V}$ and all $\mathcal{G}_i \in \mathcal{P}$ in $\mathcal{O}(n)$. As there are n vertices total, we can initialize all of the $\Delta W(\mathcal{G}_i, v)$ in $\mathcal{O}(n^2)$. After a swap or transfer is performed, all the

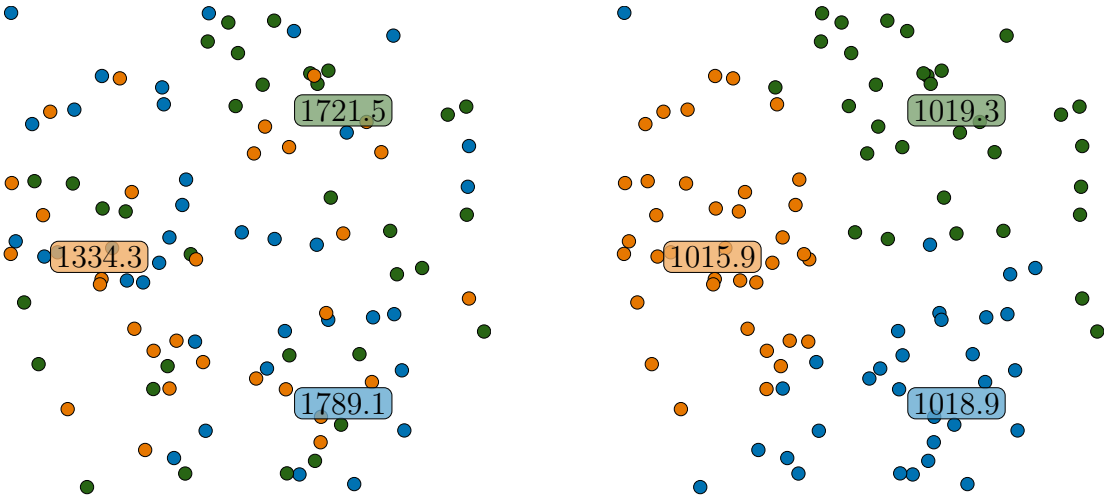


Figure 3.8: Random initial partition of a graph (left) and the partition of after improvements using Algorithm 3.1 (right). The improved partition has much lower cost and more balanced subgraph sizes.

marginal weights are updated in $\mathcal{O}(n)$ via (3.13)–(3.14) in line 13 of Algorithm 3.1. By storing the nm marginal sizes, we can compute the subgraph sizes after a potential move in $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$ and find the best move in $\mathcal{O}(n^2)$ instead of $\mathcal{O}(n^3)$.

Theorem 3.3. *For a partition, $\mathcal{P}^0$, let $\mathcal{P}^k$ represent the modified partition after k iterations of Algorithm 3.1. Then $C_{\text{avg}}(\mathcal{P}^k) \leq C_{\text{avg}}(\mathcal{P}^{k-1})$.*

Proof. Let $\mathcal{P}^k = \{\mathcal{G}_1^k, \dots, \mathcal{G}_m^k\}$. We can assume without loss of generality that $S_{\text{avg}}(\mathcal{G}_j^{k-1}) \leq S_{\text{avg}}(\mathcal{G}_i^{k-1})$. In each iteration of Algorithm 3.1, a move is only performed if $\max\{S_{\text{avg}}(\mathcal{G}_i^k), S_{\text{avg}}(\mathcal{G}_j^k)\} < S_{\text{avg}}(\mathcal{G}_i^{k-1})$. Since only these two subgraphs change, the size of all other subgraphs remains constant. Therefore for every $\mathcal{G}_i^k \in \mathcal{P}^k$, there exists $\mathcal{G}_{i'}^{k-1} \in \mathcal{P}^{k-1}$ such that

$$S_{\text{avg}}(\mathcal{G}_i^k) \leq S_{\text{avg}}(\mathcal{G}_{i'}^{k-1}) \leq C_{\text{avg}}(\mathcal{P}^{k-1}).$$

Therefore every subgraph of $\mathcal{P}^k$ is at most as large as the largest subgraph of $\mathcal{P}^{k-1}$ so $C_{\text{avg}}(\mathcal{P}^k) \leq C_{\text{avg}}(\mathcal{P}^{k-1})$. □

Theorem 3.4. *The sequence $\mathcal{P}^0, \mathcal{P}^1, \dots$ as defined in Theorem 3.3 is finite and*

its length is $\mathcal{O}(n^{m-1}m^2)$.

Proof. In each round of Algorithm 3.1, either a move is performed to improve a pair of subgraphs or a check variable, $\chi_{i,j}$, is set to `true`. If all check variables are simultaneously `true`, the algorithm terminates. Since the check variables are only reset to `false` in rounds where a move is performed, the number of rounds without any improvements equals the number of check variables. For a partition with m subgraphs, there are $m(m-1)$ check variables, so improvements must occur at least every $\mathcal{O}(m^2)$ rounds. When an improvement occurs, the size of the larger subgraph (in the pair) strictly decreases and, as there are a finite number of possible subgraphs, there a finite number of subgraph sizes and the amount it decreases by is lower bounded by some $\delta > 0$.

Although these improvements strictly reduce the maximum size of a pair of subgraphs, the cost of partition only decreases if the pair includes the largest subgraph of the partition. For $m = 2$ the pair always includes the largest subgraph. It has at most n vertices and its average edge length is at most $w_{\max}$—the length of the longest edge of $\mathcal{G}$—so its initial size is at most $nw_{\max}$. As its size is always positive and it decreases by at least δ each round, the maximum number of times it can decrease in size is $\frac{nw_{\max}}{\delta} \in \mathcal{O}(n)$. It decreases at least every $\mathcal{O}(m^2)$ rounds so there can be at most $\mathcal{O}(nm^2)$ rounds before the algorithm terminates when $m = 2$.

Now, I will prove the theorem by induction on m, having already proven the base case, $m = 2$. By the induction hypothesis, assume the theorem holds for $m-1$. We are interested in determining the maximum rounds without any improvement to the cost of the partition. Since the cost of the partition improves whenever the size of the largest subgraphs improve, this question is equivalent to determining the maximum number of rounds where only the smallest $m-1$ subgraphs improve. If we consider the graph without its largest subgraph as a separate partition problem, by the induction hypothesis, we know that after at most $\mathcal{O}(n^{(m-1)-1}(m-1)^2) = \mathcal{O}(n^{m-2}m^2)$ rounds the algorithm would terminate when running on this smaller

problem. Therefore, on the full problem, the largest subgraph's size, and thus the cost of the partition, must decrease at least every $\mathcal{O}(n^{m-2}m^2)$ rounds. As the cost of the partition can decrease by at most $nw_{\max}$ before the algorithm terminates and it must decrease by at least δ, it can decrease at most $\mathcal{O}(n)$ times. Combining these two facts, there can be at most $\mathcal{O}(n)\mathcal{O}(n^{m-2}m^2) = \mathcal{O}(n^{m-1}m^2)$ rounds of Algorithm 3.1. □

Theorem 3.5. *Each round of Algorithm 3.1 takes $\mathcal{O}(n^2)$.*

Proof. Each iteration of Algorithm 3.1 involves a search for a pair of subgraphs (line 2 or 8), a search for a transfer or swap (line 4 or 10), the transfer or swap (line 6 or 12), and an update of subgraph and marginal sizes (line 13). Searching for unchecked subgraphs requires $\frac{m(m-1)}{2}$ checks of binary variables and so is $\mathcal{O}(m^2)$. Searching for transfers or swaps is a search over $n_i < n$ or $n_i n_j < n^2$ possible transfers or swaps and computing $S_{\text{avg}}(\mathcal{G}_i)$ and $\mathcal{G}_j$ after each potential operation takes constant size so these steps are $\mathcal{O}(n^2)$. Performing a transfer or swap takes constant time. Updating each W or ΔW after an operation takes constant time but as there are $2n$ versions of ΔW that must be updated, this step takes $\mathcal{O}(n)$. Combining these steps, each iteration takes $\mathcal{O}(m^2 + n^2) \subset \mathcal{O}(n^2)$. □

3.4.2 Transfer of outliers

The partition produced by Algorithm 3.1 is a local minimum with respect to the transfer and swap moves. It cannot be improved further using any combination of these local moves. If we want to further improve this partition, we can instead try to find a new local minimum with a lower cost. We could simply start with a new random partition and apply Algorithm 3.1 to find a new local minimum, hopefully with a lower cost. This approach would be fairly computationally expensive because we have to compute a new local minimum from scratch, and it is no more likely to be better than the previous local minimum than it is likely to be worse.

Rather than use this naïve approach, we instead modify the existing local minimum by transferring several *outlier* vertices to get a new partition which is not a local minimum and may be in the region of attraction for some different local minimum. Since this partition is based on a local minimum, it is already mostly optimized and will not take many rounds of Algorithm 3.1 to reach a new local minimum. Transferring outliers does not usually improve the minmax cost (the size of the largest subgraph) but is guaranteed to reduce the minsum cost (the sum of all subgraph sizes). Since partitions with lower minsum costs tend have lower minmax costs, if transferring outliers moves the partition into a different region of attraction, the new local minimum is more likely to have a lower minmax cost as well.

The definitions (3.8) and (3.9) can be rearranged as

$$W(\mathcal{G}_i) = \frac{1}{2} \sum_{v \in \mathcal{V}_i} \Delta W(\mathcal{G}_i, v).$$

This identity tells us that $\Delta W(\mathcal{G}_i, v)$ is proportional to vertex v's *contribution* to $S_{\text{avg}}(\mathcal{G}_i)$. An effective way to escape local minima, therefore, is to search for vertices, $v \in \mathcal{V}_i$, with a large $\Delta W(\mathcal{G}_i, v)$ but a small $\Delta W(\mathcal{G}_j, v)$ for some other subgraph, $\mathcal{G}_j$. Such a vertex is *out-of-place* because it would have a smaller contribution to the size of $\mathcal{G}_j$ than it is currently having to the size of $\mathcal{G}_i$.

Although Algorithm 3.1 tends to move out-of-place vertices to more appropriate subgraphs, it only performs moves which decrease $\max\{S_{\text{avg}}(\mathcal{G}_i), S_{\text{avg}}(\mathcal{G}_j)\}$. When two subgraphs already have a similar size, transferring an out-of-place vertex would often violate this constraint. These out-of-place vertices could be moved to more suitable subgraphs if we allowed 2 or more vertices to be transferred back in exchange; however, such a move would have a larger—at least cubic—search space. Rather than use this larger search space, we simply transfer the outliers and then rerun Algorithm 3.1.

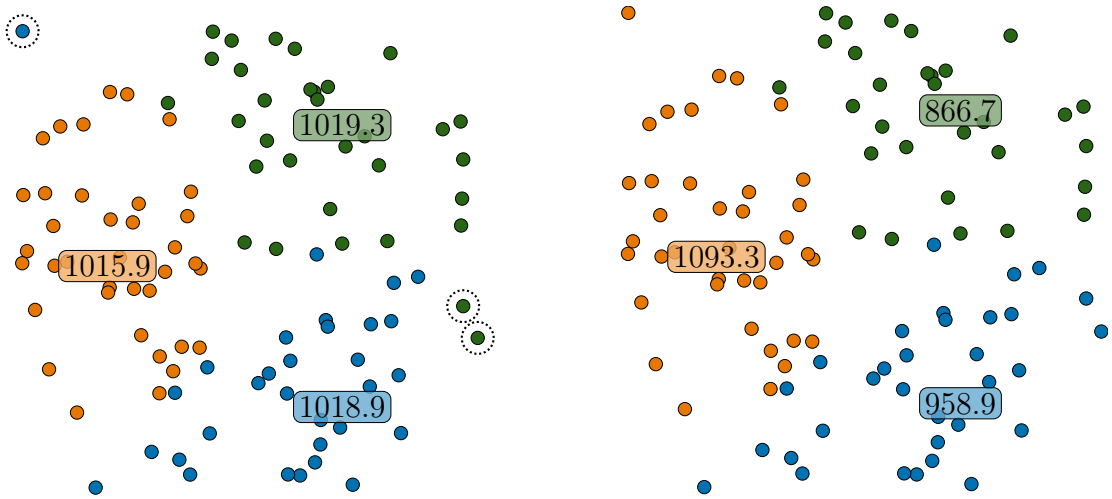


Figure 3.9: Partition of a graph before (left) and after (right) transferring outliers (circled) using Algorithm 3.2 with $\omega = 1.5$ (right). After the transfer, the obvious outlier vertex in the top left corner has been transferred to a more suitable subgraph; however the total cost of the partition has increased.

In the second phase of the heuristic (Algorithm 3.2) we allow some violation of the constraint that $\max\{S_{\text{avg}}(\mathcal{G}_i), S_{\text{avg}}(\mathcal{G}_j)\}$ when moving *outlier* vertices to better subgraphs. A vertex $v \in \mathcal{V}_i$ is an outlier if $\Delta W(\mathcal{G}_i, v) > \Delta W(\mathcal{G}_j, v)$ for some $\mathcal{G}_j$ and

$$\Delta W(\mathcal{G}_i, v) > \omega \sum_{v' \in \mathcal{V}_i} \frac{1}{n_i} \Delta W(\mathcal{G}_i, v') = \omega \frac{2}{n_i} W(\mathcal{G}_i).$$

This second criterion is that v contributes more to $S_{\text{avg}}(\mathcal{G}_i)$ than an average vertex of $\mathcal{G}_i$. The outlier detection threshold, $\omega \geq 1$, is used to control the number outliers detected which decreases as ω increases. I found that $\omega = 1.5$ gave good results. After transferring outliers (Figure 3.9), every vertex will be in a more suitable subgraph, but the largest subgraph may have grown even larger.

Algorithm 3.2 begins by identifying all outliers (lines 2–9). For each subgraph, it checks if each vertex's contribution is above the detection threshold (line 4) and if it is, checks if the vertex is an outlier (line 5). Every outlier, along with its current subgraph and the subgraph it fits best in, is added to a set (lines 7–9). After identifying all outliers, they are transferred to the subgraphs they fit best in

(line 11) and $W(\mathcal{G}_i)$ and $\Delta W(\mathcal{G}_i, v)$ are updated to reflect this transfer (lines 12–14). All outliers are identified before they are transferred because transferring outliers changes $W(\mathcal{G}_i)$ and $\Delta W(\mathcal{G}_i, v)$ which could affect which vertices are classified as outliers. After transferring outliers, some vertices which were not considered outliers may now meet the definition of an outlier using the updated partition. These new outliers could be transferred by running Algorithm 3.2 another time. This process of transferring outliers followed by swapping and transferring to reach a new local minimum can be repeated several times until the new local minimum is the same as the previous one.

Algorithm 3.2: Transfer outliers

Input: Partition, $\mathcal{P}$; and threshold, $\omega \geq 1$
Output: Partition, $\mathcal{P}'$, with outliers transferred

```
1  U ← {}                                                    /* Set of outliers */
2  for subgraph G_i ∈ P do
3      for vertex v ∈ G_i do
4          if ΔW(G_i, v) > (2ω/n_i) W(G_i) then      /* Is potential outlier */
5              G_j ← subgraph of P which minimizes ΔW(G'_i, v)
6              if G_j ≠ G_i then                              /* Is outlier */
7                  U ← U ∪ {v}
8                  G_old(v) ← G_i
9                  G_new(v) ← G_j
10 for outlier v ∈ U do
11     Transfer v from G_old(v) to G_new(v)
12     Update W(G_old(v)) and W(G_new(v))                     /* (3.8) */
13     for vertex v' ∈ V do
14         Update ΔW(G_old(v), v') and ΔW(G_new(v), v')       /* (3.9) */
15 return P' = {G_1, ..., G_m}
```

Theorem 3.6. *Algorithm 3.2 terminates in $\mathcal{O}(n^2)$.*

Proof. Algorithm 3.2 consists of an identification loop (lines 2–9) and a transfer loop (lines 10–14). There are $\sum_{i=1}^{m} n_i = n$ iterations of the identification loop. Each iteration involves finding $\mathcal{G}_j$ which requires m comparisons, and potentially saving v, $\mathcal{G}_i$, and $\mathcal{G}_j$ which requires constant time. Therefore identifying outliers

takes $\mathcal{O}(mn)$. The transfer loop involves at most n transfers. Each transfer takes constant time and is accompanied by an update of W and ΔW for $\mathcal{G}_{\text{old}}$, $\mathcal{G}_{\text{new}}$, and all $v' \in \mathcal{V}$ which takes $\mathcal{O}(n)$. As there are at most n outliers, transferring them all takes $\mathcal{O}(n^2)$ and since $m < n$, the overall algorithm terminates in $\mathcal{O}(n^2)$. □

3.4.3 Overall partition algorithm

Transferring outliers using Algorithm 3.2 and the transfers and swaps of Algorithm 3.1 are two effective ways to improve a partition. Alternating between these two algorithms is the basis of my main heuristic for the APP (Algorithm 3.3).

Algorithm 3.3: Average partition algorithm (APA)

Input: Complete graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$; and number of robots, m
Output: Partition, $\mathcal{P}^*$, nearly minimizing the average cost C_{avg}

1 $\mathcal{P} \leftarrow$ random partition of $\mathcal{G}$
2 Compute $W(\mathcal{G}_i)$, $\Delta W(\mathcal{G}_i, v)$, for all $\mathcal{G}_i \in \mathcal{P}$ and $v \in \mathcal{V}$ /* (3.8)--(3.9) */
3 Improve $\mathcal{P}$ by transfers and swaps /* Algorithm 3.1 */
4 $C^*_{\text{avg}} \leftarrow \infty$ /* Cost of best partition */
5 **while** $C_{\text{avg}}(\mathcal{P}) < C^*_{\text{avg}}$ **do** /* Improvements possible */
6 $\quad C^*_{\text{avg}} \leftarrow C_{\text{avg}}(\mathcal{P})$
7 $\quad \mathcal{P}^* \leftarrow \mathcal{P}$ /* Best partition */
8 $\quad$ Transfer outliers of $\mathcal{P}$ /* Algorithm 3.2 */
9 $\quad$ Improve $\mathcal{P}$ by transfers and swaps /* Algorithm 3.1 */
10 **return** $\mathcal{P}^*$

Algorithm 3.3 starts with a randomly generated partition (line 1) and improves it by alternating between Algorithms 3.1 and 3.2. It computes $W(\mathcal{G}_i)$ and $\Delta W(\mathcal{G}_i, v)$ for this partition (line 2) using their definitions (3.8) and (3.9) and then improves the partition as much as possible using transfers and swaps (line 3). In each round of the main loop (lines 5–9), outliers are transferred (line 8) and the resulting partition is improved (line 9). When outliers are transferred, $\Delta W(\mathcal{G}_i, v)$ changes if v has been transferred or $\mathcal{G}_i$ has had at least one vertex transferred to/from it. This phase (line 8) is therefore not guaranteed to improve $C_{\text{avg}}(\mathcal{P})$ so it is always followed immediately by a partition improvement phase (line 9). If

these two phases improve the partition, the algorithm continues and keeps the improved partition (line 5); otherwise, the algorithm returns the partition from before the outliers were transferred (line 10). In this way, Algorithm 3.3 never returns a worse partition as a result of transferring outliers. After an improvement phase, the improved partition is a local minimizer of C_{avg} with respect to the transfer and swap operations of Algorithm 3.1. Transferring outliers is used to escape local minima but usually increases $C_{\text{avg}}(\mathcal{P})$. After transferring these outliers, another improvement phase creates another partition that is a local minimizer and may have a higher or lower C_{avg} than before. As transferring and improving does not always increase C_{avg}, we make a copy of $\mathcal{P}$ called $\mathcal{P}'$ (line 7) and transfer outliers and improve this copy. If the modified $\mathcal{P}'$ has a lower cost than the original $\mathcal{P}$, we assign $\mathcal{P}'$ to $\mathcal{P}$ and perform another round of the main loop (line 5). If the modified $\mathcal{P}'$ has a higher cost, we exit the loop and return the best partition found, $\mathcal{P}$ (line 10).

The partition produced by Algorithm 3.3 is a local minimum of the APP with respect to transfers and swaps and it is our final solution to the APP. Furthermore, by transferring outliers it effectively finds a nearby local minimum and continues improving the solution if a nearby local minimum is better. As a result, Algorithm 3.3 is guaranteed to produce partitions which are at least as good the partitions produced by the improvement phase (Figure 3.10) on its own, and usually produces better solutions. On our example graph (Figure 3.10), the cost of the final partition is $C_{\text{avg}}(\mathcal{P}) = 983.4$ which is a 3.6% improvement on the original cost of 1019.3 obtained by improving the partition without transferring any outliers.

3.5 From a partition to cycles

Earlier, I had proposed using the APP as a proxy for the MPP so that we can use the solution to the APP as if it were a solution to the MPP. In Section 3.4,

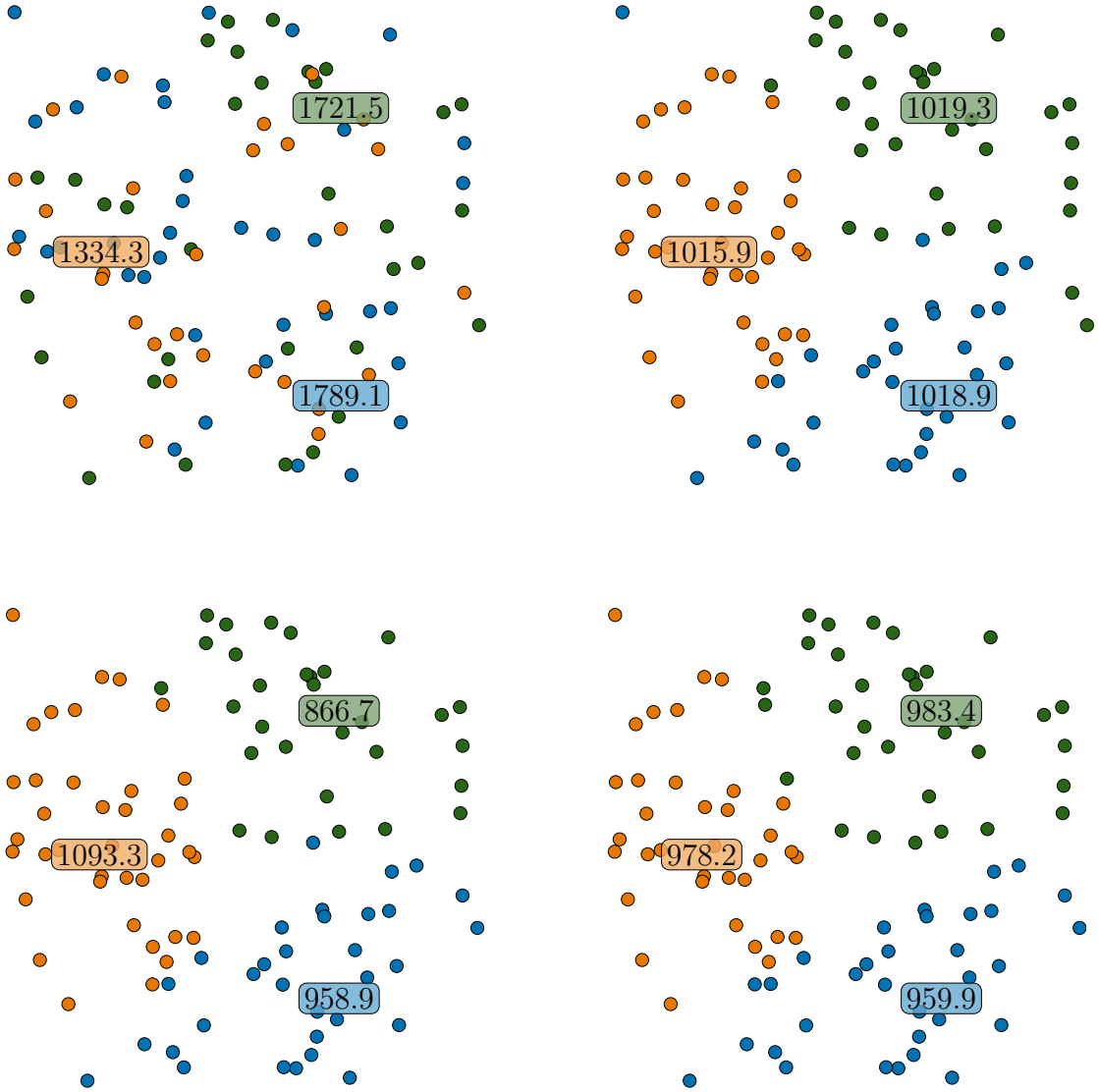


Figure 3.10: Starting with a random partition (top left), Algorithm 3.3 alternates between Algorithm 3.1 to improve the partition (top right) and Algorithm 3.2 to transfer outliers (bottom left). Alternating between these two algorithms to produce a final partition (bottom right) which is a better solution to the APP than would be obtained by either algorithm on their own.

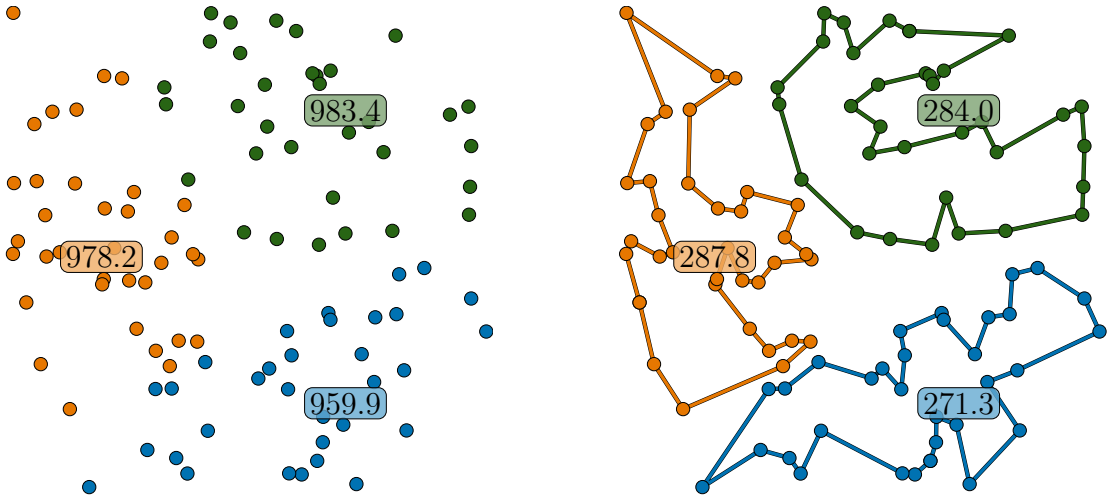


Figure 3.11: Partition of a graph based on minimizing the average cycle length of the largest subgraph (left) and shortest cycles on each of these subgraphs (right). The shortest cycles are 28.26%, 28.88%, and 29.42% the length of their subgraph's size, indicating that average cycle length is a good proxy for shortest cycle length. The shortest paths were approximated using the Concorde 1-TSP solver [42].

I presented a heuristic for the APP which partitions a graph into m subgraphs which is a local minimum of the average cost, (3.5), with respect to transfers and swaps. By Corollary 3.2, if $S_{\min}$ and S_{avg} are related by a monotonically increasing function, then this solution is also a local minimum of the minimum cost, (3.3). In reality, the relationship between S_{avg} and $S_{\min}$ is not perfectly monotonic (Figure 3.3), so the optimal partitions for the APP and MPP differ slightly. The solution to the APP is still useful as an initial partition for an improved m-TSP algorithm (Algorithm 3.4). The initial m-TSP solution is obtained by solving the 1-TSP on each subgraph of the partition (Figure 3.11). This solution is improved by transferring vertices between cycles to reduce its minmax cost. The best transfer can be found in $\mathcal{O}(n^2)$ by checking all pairs of vertices in the longer cycles and locations for insertion in the shorter cycle. The algorithm alternates between transferring vertices between cycles and solving the 1-TSP for each cycle until no more improvements can be made.

Algorithm 3.4 involves solving m instances of the 1-TSP. Although the 1-TSP is

NP-hard [150], several open-source solvers [42, 76] have very good performance and runtimes. Furthermore, we are solving m instances of the 1-TSP with $n_1, \ldots, n_m$ vertices each ($n_1 + \cdots + n_m = n$) instead of a single instance with n vertices. Solving these m smaller instances is faster than solving the single large instance because the runtime of TSP solvers is slower than linear in the number of vertices. Although Algorithm 3.4 alternates between solving the individual 1-TSPs and transferring vertices between cycles, the additional solutions of the 1-TSPs often return the exact same cycle. A slightly faster algorithm—one which solves the 1-TSP exactly m times—could therefore be obtained by only using one iteration of the inner loop of Algorithm 3.4 (lines 4–8).

Algorithm 3.4: m-TSP path algorithm (MPA)

Input: Complete graph, $\mathcal{G}$; and number of robots, m
Output: Set of m cycles, $\mathcal{C}$, solving the minmax m-TSP

```
1 P ← solution of APP for G with m robots          /* Algorithm 3.3 */
2 C ← solutions to 1-TSP on each subgraph of P
3 C* ← ∞                                  /* Cost of best set of cycles */
4 while C(C) < C* do                          /* Improvements possible */
5 |   C* ← C(C)
6 |   Improve C by transferring vertices between cycles
7 |   P ← partition induced by C
8 |   C ← solutions to 1-TSP on each subgraph of P
9 return C
```

The initial cycles could alternatively be improved by a more sophisticated search heuristic such as tabu search or simulated annealing. Despite using a relatively simple improvement heuristic, I was able to solve large minmax m-TSP problems and obtain better solutions than other approaches. As the average cycle length is a good proxy for the shortest cycle length, Algorithm 3.4 usually only needs to transfer a few vertices. In our example (Figure 3.12), only three transfers—decreasing the solution's cost by 0.15%—were needed before reaching the final solution. The success of this approach demonstrates that a good initial partition can offset the need for a good cycle improvement heuristic.

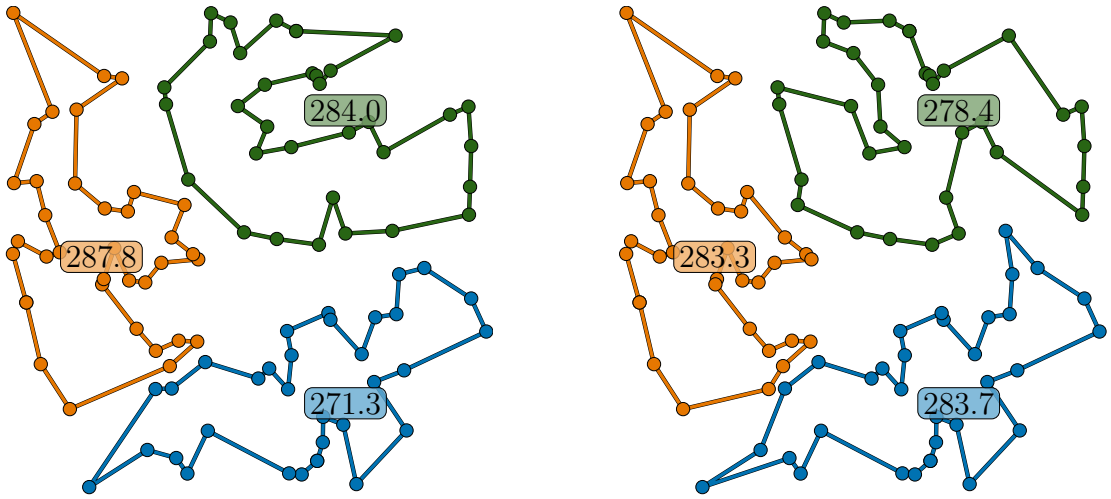


Figure 3.12: The paths produced by solving the 1-TSP on each subgraph of a partition from Algorithm 3.3 (left) can be further improved by transferring individual vertices between paths. The resulting paths after transferring vertices (right) are slightly shorter and thus a better solution to the MPP and m-TSP.

3.6 Heterogeneous robots

Robots may be required to work in *heterogeneous* teams where different robots have different abilities. These teams may consist of physically different robots where only some robots can complete certain tasks, or may be teams of robots which appear identical but one robot is a bit slower because it is older, has a lower battery, or has something stuck in its wheel. When allocating tasks to a heterogeneous team, a balanced allocation is one where each robots' assigned tasks will take a similar amount of time based on its abilities so that the team finishes as quickly as possible.

To assign tasks in a heterogeneous team, we need to use separate weight functions, $w_1, \ldots, w_m$, for each robot in the team. Each $w_i : \mathcal{E} \to \mathbb{R}_{\geq 0}$ is for the same graph, but has different weights based on that robots' ability. If robot i is completely unable to do a certain task, then $w_i(e) = \infty$ for any edge incident to that task's vertex. Otherwise, $w_i(e)$ equals the full time needed for robot i to travel between the two tasks plus half the time needed to complete the tasks. Then, when

computing the partition via Algorithm 3.3, $W(\mathcal{G}_i)$ and all of the $\Delta W(\mathcal{G}_i, v)$'s are defined using the w_i for that robot. Additionally, the initial partition should be chosen so that every task is initially assigned to robot that can actually complete it. With these two changes, Algorithm 3.3 will produce a balanced partition or as close to one as possible if some robots are much slower or have fewer abilities than others. Once the partition has been computed, each 1-TSP must be solved with the correct weight function for that robot, and the correct weight functions must be used when transferring vertices between paths.

3.7 Decentralization

The version of my task allocation and routing heuristic that I've presented so far (Algorithm 3.4) is centralized. However, as the majority of the computation is based on exchanges of vertices between pairs of robots, the algorithms can easily be converted to a decentralized form where each robot manages its own list of tasks. We assume that that all robots are able to communicate with each other to share the $m(m-1)$ binary check variables, $\chi_{i,j}^{\text{swap}}$ and $\chi_{i,j}^{\text{tran}}$, and to share the initial random partition. The decentralized algorithm (Figure 3.13) is divided into two phases—a partition phase equivalent to Algorithm 3.3 and a cycle phase equivalent to Algorithm 3.4—which consist of exchanges happening between pairs of robots.

Algorithm 3.1 consists of exchanges (transfers in lines 4–6 or swaps in lines 10–12) of vertices between pairs of subgraphs. As these exchanges only involve 2 robots' graphs, multiple pairs of robots can compute exchanges simultaneously resulting in a decentralized version of Algorithm 3.1. In this decentralized version, robot i maintains $\mathcal{G}_i$, $W(\mathcal{G}_i)$, and $\Delta W(\mathcal{G}_i, v)$ for all $v \in \mathcal{V}$. In its default idle state, it examines the check variables to find a robot j that it has unchecked transfers or swaps with and attempts to connect with robot j. If robot j is busy computing an exchange with some robot k, robot i will not be able to connect to robot j and will

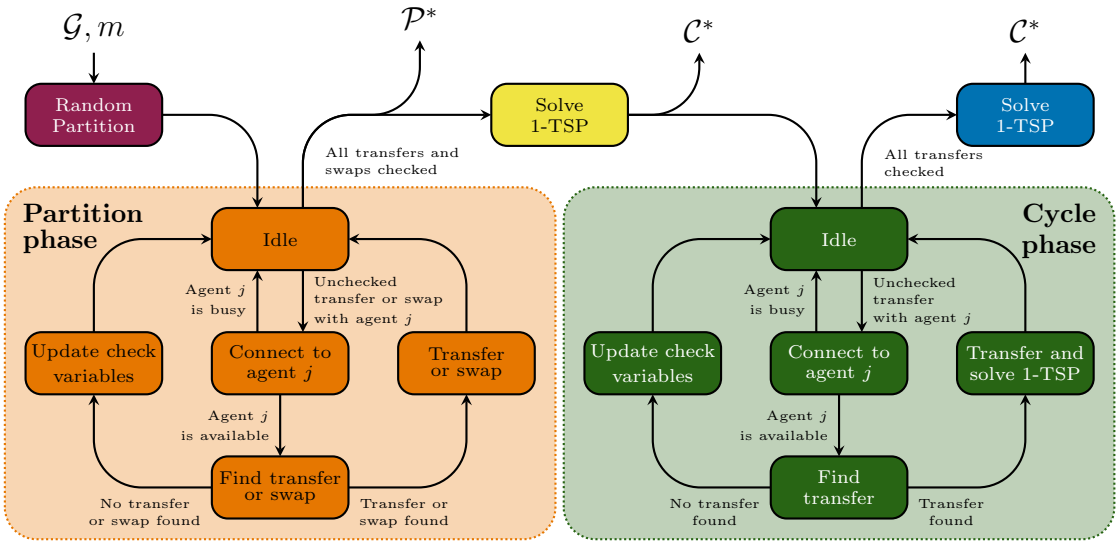


Figure 3.13: Decentralized task allocation algorithm from the perspective of robot i. $\mathcal{P}^*$ is the partition produced by Algorithm 3.3; $\mathcal{C}$ is the set of cycles obtained by each robot solving the 1-TSP on its own set of tasks. $\mathcal{C}^*$ is the final solution produced by Algorithm 3.4.

instead search for another robot that it needs to check transfers or swaps with. If robot j is available, then robots i and j share their subgraphs and marginal edge weights with each other to search for the best transfer or swap. If they find a move which reduces the maximum size of subgraphs i and j, they implement this move and update the relevant check variables that are shared by all robots. This process continues until all pairs of robots have searched for transfers or swaps between their subgraphs without finding any improvements. As this algorithm continually improves pairs of subgraphs, it has the same overall behavior as Algorithm 3.1 despite no robot knowing the overall partition. Overall the decentralized version may be faster than the centralized one as multiple pairs of robots can search for transfers and swaps simultaneously, effectively parallelizing the main loop of Algorithm 3.1.

Once the robots have all finished transferring and swapping tasks as much as possible, they use a similar approach to search for and transfer outliers between pairs of robots. Algorithm 3.2 consists of a search for outliers in each graph which

doesn't modify any of the graphs (lines 2–9) and then a transfer of these outliers after they have all been identified (lines 10–14). It can be decentralized by having each robot identify outliers in its own graph followed by pairwise communication with other robots to transfer the outliers that were identified. By alternating between transferring or swapping based on maximum subgraph size and transferring outliers, the team of robots will obtain the same partition as would be produced by Algorithm 3.3. This decentralized version also requires some synchronization so all the robots know which kind of transfers to check at any time and when the partition is complete.

The remainder of Algorithm 3.4 consists solution of the 1-TSP on each subgraph (line 8) and transfers of vertices between pairs of cycles (line 6). After the partition phase is complete, each robot solves the 1-TSP on its partition to get a cycle. As the solution of the 1-TSP on $\mathcal{G}_i$ does not depend on any other $\mathcal{G}_j$, the robots can each compute their own cycle. Finally, the robots transfer vertices between their initial cycles. This process is quite similar to Algorithm 3.1 and can be decentralized in essentially the same way with robots using shared check variables to determine which robots to search for transfers with. Since the robots are now exchanging vertices between cycles, they must optimize over which vertex to transfer and the location in the shorter cycle to transfer it to. Therefore, they only consider transfers and not swaps. When two robots transfer a vertex, each robot can optionally recompute its own 1-TSP cycle based on its new tasks in case their is a better route. Once all pairs of robots have searched for transfers without finding any improvements, the algorithm is complete. The resulting set of cycles, $\mathcal{C}^*$, equivalent to the optimal solution computed by Algorithm 3.4.

3.8 Paths with depots

Most robots have some constraints about where they must start and end a mission. Delivery robots must start and end their deliveries at the warehouse or postal depot where undelivered packages are stored at. Robotic vacuum cleaners start and end cleaning missions at their charging station. If there is a team of delivery robots, they all have the same warehouse, whereas each robotic vacuum cleaner has its own charging station. A robotic arm performing a repetitive motion, such as drilling holes in circuit boards, can start and end its path in any location. If it needs to return to the same location to perform that motion again, the start and end points must be the same. However, if it can do every other motion in reverse, the start and end points can be different. Robots that replan during a mission need to use their current location, wherever it may be, as their start point while keeping the same end location.

As different robotic applications can require many different start and end constraints, I will consider general forms of start and end constraints. I will use the term *depot* to refer to any location where a robot must start or end its path. An individual robot's path can be classified in one of five categories (Figure 3.14) depending on its depot constraints:

1. Cycle with 0 depots: $v_{\text{start}} = v_{\text{end}}$
2. Cycle with 1 depot: $v_{\text{start}} = v_{\text{end}} = v_{\text{depot}}$
3. Open path with 0 depots: No constraints
4. Open path with 1 depot: $v_{\text{start}} = v_{\text{depot}}$
5. Open path with 2 depots: $v_{\text{start}} = v_{\text{depot}} \neq v'_{\text{depot}} = v_{\text{end}}$

Within a team of robots, different robots' paths may fit in different categories. If multiple robots have depots, they may be distinct or unique physical locations.

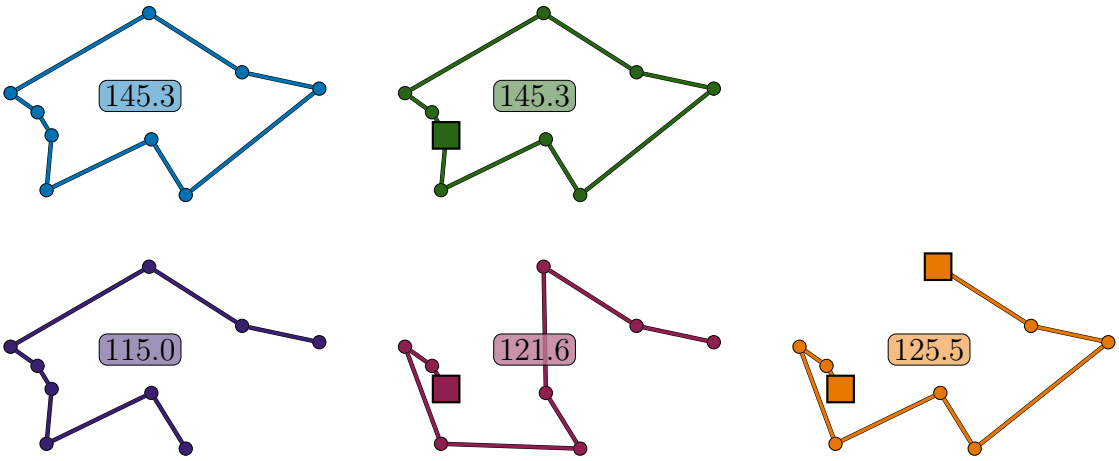


Figure 3.14: A robot's depot constraints can be classified in 5 different ways depending on whether it has a cyclic or open path and how many depots it has. In general, open paths are shorter than cycles and additional depots result in longer paths.

Problems with different depot constraints can be solved using Algorithm 3.4 with slight modifications to Algorithms 3.1–3.4. In the initial partition, each subgraph must contain its robot's depots and these depots cannot be transferred or swapped in Algorithms 3.1 or 3.2. If multiple robots share a depot, additional copies of this vertex should be added so that there is a unique depot vertex per robot. For open paths, (3.6) should be modified to become

$$S_{\text{avg}}(\mathcal{G}_i) = \frac{2}{n_i} \sum_{e \in \mathcal{E}_i} w(e)$$

as open paths only contain $n_i - 1$ edges. Once a partition is found, the 1-TSP is solved with the relevant depot constraint. When the paths are being improved by Algorithm 3.4, the depots again cannot be transferred. This approach can be used to generate solutions to the various categories of depot constraints (Figure 3.15). This flexibility of depot configurations makes my approach novel as existing approaches require either a single shared depot [6, 24, 30, 62, 72, 123, 125, 144, 149, 172, 176, 191, 194] or one unique depot per robot [103, 143]. My algorithm works for unique or shared depots, open paths or cycles, and 0, 1, or 2 required depots

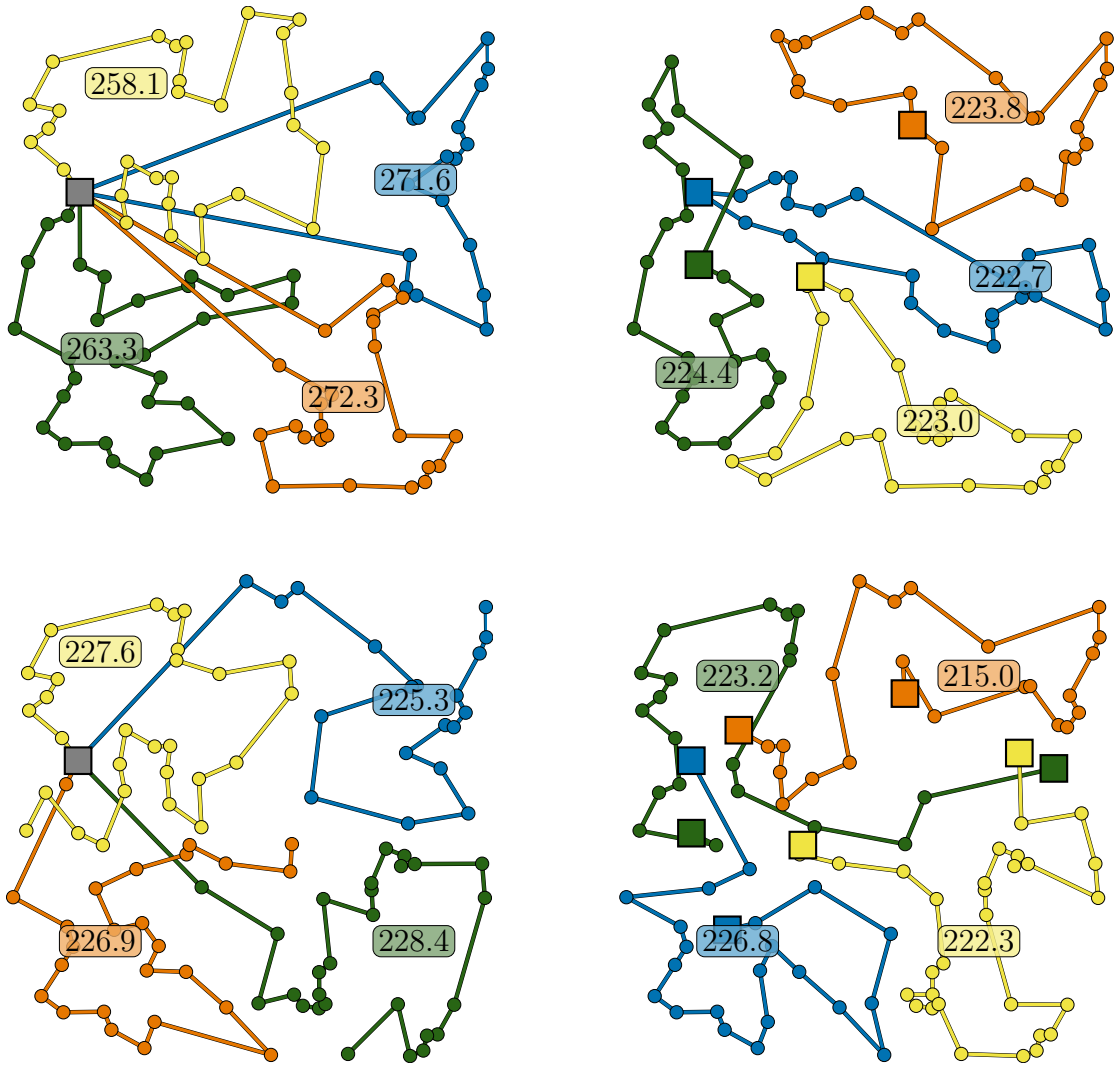


Figure 3.15: Solutions for the same task allocation problem when the robots have cyclic paths with a shared depot (top left), cyclic paths with unique depots (top right), open paths with a shared depot (bottom left), and open paths with two unique depots each (bottom right).

per robot. Furthermore, there is no requirement that each robot has the same kind of depot constraints.

3.9 Results

I compared my algorithm against two state-of-the-art algorithms for problems with $50 \leq n \leq 5000$ and $3 \leq m \leq 100$ and different depot configurations. My algorithm was implemented in Python and solutions were computed with $\omega = 1.5$ using a

Table 3.1: Comparison of cost, C, and runtimes, t, achieved by HMS [103] with MPA (Algorithm 3.4). Results for HMS are from the single solutions computed in [103] for 5000 vertices uniformly distributed on $[0, 100] \times [0, 100]$ with 10 or 100 robots. Results for MPA are based on 20 solutions with different random seeds.

		HMS		MPA		
n	m	C	t	$C_{\min}^{\min}$	$C_{\min}^{\text{avg}}$	t_{avg}
5000	10	577	12000	**513.66**	**516.56**	**6199.36**
5000	100	64.738	76477	**55.65**	**56.73**	**4786.58**

Linux desktop computer with a 3.40 GHz processor and 8 GB of memory. For each comparison, I computed 20 different solutions to the same problem using Algorithm 3.3 with different random seeds. The costs reported are $C_{\min}^{\min}$, the minimum $C_{\min}$ across these 20 solutions, and $C_{\min}^{\text{avg}}$, the average $C_{\min}$ across these 20 solutions. (These are the same summary statistics reported by Wang *et al.* [194].)

3.9.1 Problems with multiple depots

I compared my algorithm with the hierarchical market-based solution (HMS) from Kivelevitch *et al.* [103] for cycles with unique depots for each robot. They considered $n = 5000$ vertices on the square $[0, 100] \times [0, 100]$ with $m \in \{10, 100\}$ robots. As they did not publish the exact locations of the vertices in their instances, I randomly generated a new set of 5000 vertices from the same distribution for each of the 20 tests. My results show an average improvement of approximately 10% and had a worst case with lower cost than their result for both 10 and 100 robots (Table 3.1). Furthermore, my solutions required less computation time. The best solutions I found for 10 robots is shown in Figure 3.16.

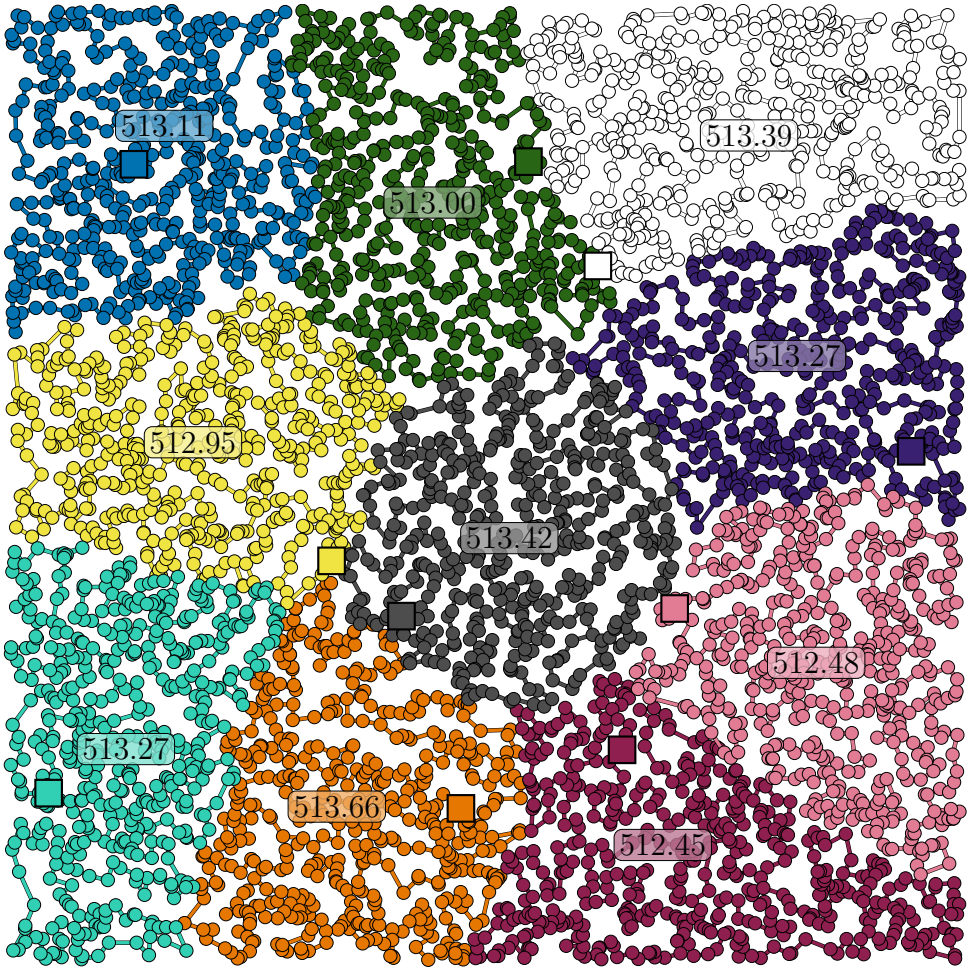


Figure 3.16: The best solution found for 5000 uniformly distributed vertices with 10 robots and 10 depots.

3.9.2 Problems with one depot

For the minmax m-TSP with one shared depot, I compared my approach with the best results found by any of the 6 heuristics that Wang *et al.* [194] compared (Table 3.2). They computed cyclic solutions for several problems from TSPLIB [155] using the first vertex in the dataset as a shared depot for all robots. As the number of solutions for the minmax m-TSP increases exponentially with both n and m, a heuristic's performance can be best evaluated by its performance on large problems. For the largest problem, with $n = 1173$, my heuristic produced better solutions with lower minmax costs for both the best solution found and

Table 3.2: Comparison of costs for solutions to several TSPLIB [155] problems obtained using invasive weed optimization (IWO) and a memetic algorithm (MA) [194] with MPA. Results are based on 20 solutions with different random seeds.

		IWO		MA		MPA	
n	m	$C_{\min}^{\min}$	$C_{\min}^{\text{avg}}$	$C_{\min}^{\min}$	$C_{\min}^{\text{avg}}$	$C_{\min}^{\min}$	$C_{\min}^{\text{avg}}$
318	3	**16200.2**	**16340.3**	16206.3	16477.9	16804.8	17265.1
318	5	**11730.0**	11908.2	11752.4	**11896.7**	12159.8	12673.3
318	10	9845.4	9955.4	**9731.2**	**9818.8**	9826.8	9971.9
318	20	**9731.2**	**9731.2**	**9731.2**	**9731.2**	**9731.2**	**9731.2**
532	3	32989.0	33687.3	**32403.1**	**33424.8**	34376.6	35171.6
532	5	23519.7	24029.6	**22619.6**	**23079.3**	24763.1	25697.2
532	10	19136.5	19439.5	**18390.4**	**18515.7**	18579.5	18958.1
532	20	17850.8	18051.0	**17641.1**	**17662.1**	17642.7	17680.1
783	3	3458.0	3497.6	**3279.1**	**3336.6**	3377.2	3414.7
783	5	2273.8	2303.1	**2092.7**	**2134.0**	2220.5	2286.5
783	10	1542.1	1564.7	**1432.3**	**1452.7**	1475.0	1515.9
783	20	1311.3	1333.1	1260.9	1270.3	**1240.9**	**1249.1**
1173	3	24008.5	24300.3	22443.2	22781.6	**20733.3**	**20999.2**
1173	5	16057.2	16274.6	14557.3	14861.4	**13876.3**	**14179.2**
1173	10	16057.2	10668.0	9222.9	9352.6	**8698.4**	**8871.3**
1173	20	8063.2	8207.9	7063.2	7276.7	**6595.9**	**6670.2**

average solution cost for 3, 5, 10, and 20 robots. The best solutions I found for the largest problem, `pcb1173`, are shown in Figure 3.17. For smaller problems ($n \in \{318, 532, 783\}$), my algorithm performs better or similarly (within 1%) when $m = 20$ but has worse performance for smaller m. As smaller m results in a problem more similar to the 1-TSP, this decreased performance may be a result of using an pre-existing 1-TSP solver and not heavily optimizing the routing portion of the algorithm.

My algorithm had average runtimes ranging from less than 1 s to 426 s. For the largest problem ($n = 1173$), my algorithm took between 146 s and 426 s which is the same order of magnitude as the 236 s used by Wang *et al.* [194]. However, as the problems were run on different computers, I cannot make more detailed comparisons.

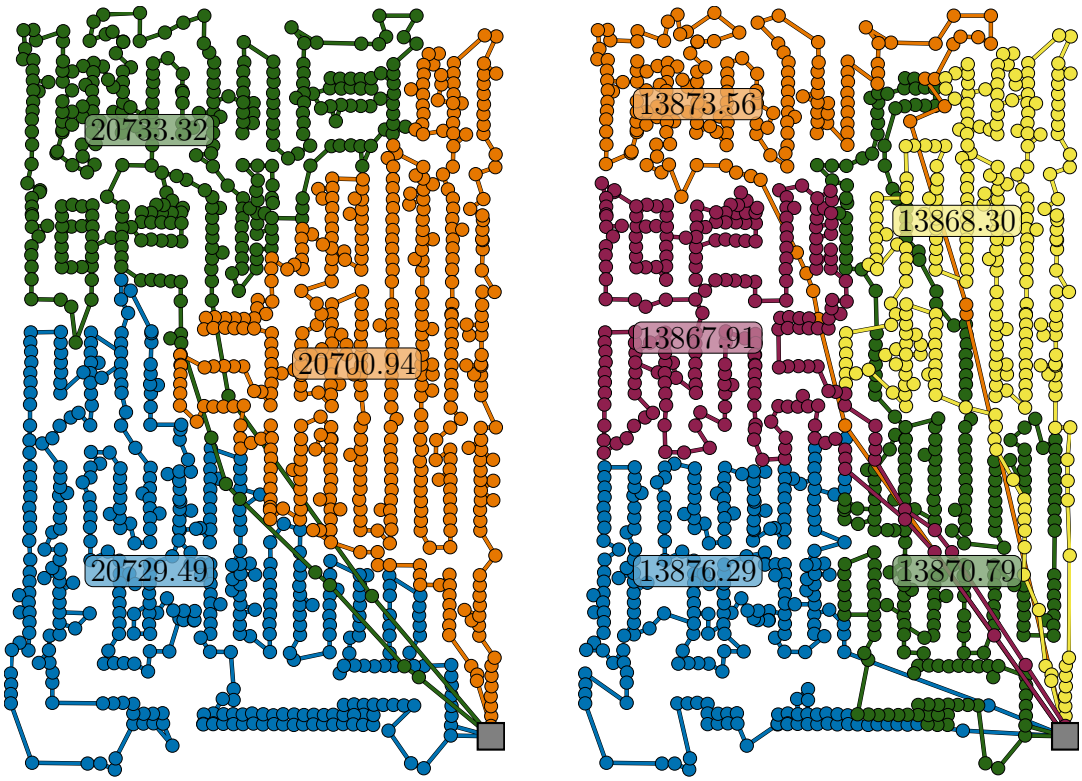


Figure 3.17: Best solutions found for `pcb1173`with 3 (left), 5 (right) robots.

3.9.3 Runtime analysis

I analyzed average runtimes for 31 test problems from TSPLIB [155]. These problems have $n \in \{51, 100, 150, 200, 318, 532, 783, 1173\}$ and $m \in \{2, 3, 10, 20\}$. The problems with $n \in \{318, 532, 783, 1173\}$ are the same problems as in Table 3.2. The runtimes are averaged over 40 trials of each problem. I assumed the runtime follows a monomial model

$$t_{\text{avg}} = k_0 n^{k_1} m^{k_2} \exp(\nu) \tag{3.17}$$

where k_0, k_1, and k_2 are parameters to be estimated and ν is zero-mean noise. I estimated k_1 and k_2 by taking the logarithm of both sides of (3.17) and the performed linear regression to obtain the model

$$\widehat{t}_{\text{avg}} = (3.1944 \times 10^{-5}\,\text{s}) n^{2.111} m^{0.325}.$$

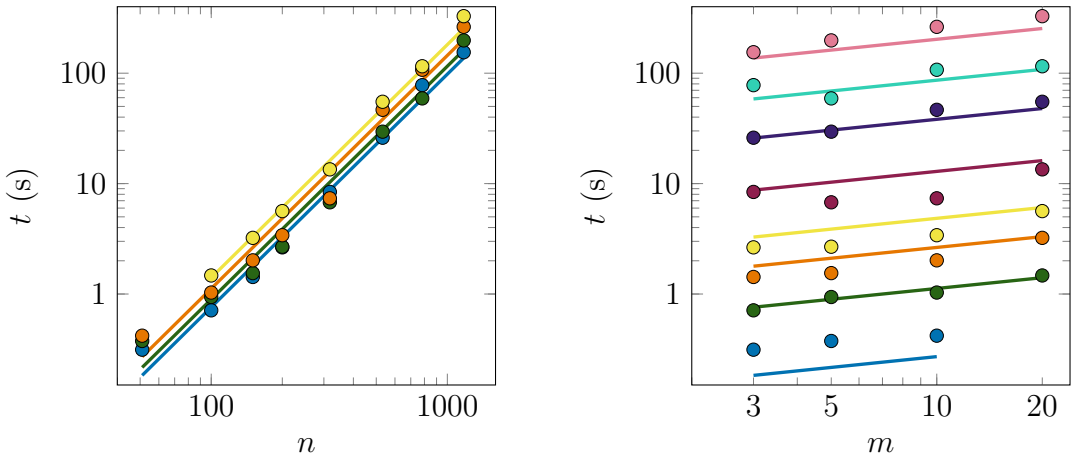


Figure 3.18: Actual average runtimes (dots) and predicted average runtimes (lines) for Algorithm 3.4. On the left, trendlines are for fixed $m \in \{3, 5, 10, 20\}$ and varying n. On the right, trendlines are for fixed $n \in \{51, 100, 150, 200, 318, 532, 783, 1173\}$ and varying m.

The estimates produced using this overall model (Figure 3.18) are close to the actual average runtimes.

3.10 Conclusions

Task assignment and routing are coupled problems for teams of mobile robots. I formulated these combined problems as a partition problem, the MPP, which is equivalent to the minmax m-TSP. Solutions to the MPP are similar to the APP—whose cost function is easier to evaluate—because their cost functions are nearly related by a monotonic function.

As these problems are NP-hard, I developed a heuristic algorithm, MPA, for the combined task assignment and routing problem. It exploits the relationship between the MPP and APP to partition a graph using a minmax criterion based on the APP's cost function. Despite the simplicity of the APP's cost function, there is a close relationship between the solutions of the two problems. MPA uses a solution to the APP and computes routes by solving the 1-TSP. The routes are improved slightly by transferring vertices between them resulting in a set of cycles

which minimizes the length of the longest cycle. These cycles solve the combined task allocation and routing problems.

Using this approach, I solved large task allocation problems and obtained better solutions than have previously been reported using a variety of algorithms. These problems had up to 5000 tasks and 100 robots and included problems with a single shared depot and one unique depot per robot. For n tasks and m robots, the algorithm's runtime was proportional to $n^{2.111}m^{0.325}$.

Chapter 4

Turn-minimizing coverage

Coverage is an example of a common but complex robotic task (Section 2.4). A robot performing coverage must travel over—or move its tool over—every point in a large region. Examples of coverage problems include:

- A farming robot harvesting a field of crops must pass its harvesting tool over an entire field of crops while staying within the region enclosed by nearby fences;
- An autonomous boat mapping a seafloor with a laser scanner needs to follow a path so that every part of the seafloor gets scanned at least once;
- A robotic arm painting a car door must cover the entire door with a uniform layer of paint by passing its spray nozzle over the unpainted door;
- An autonomous snowplow has to plow every street of the city without straying onto the sidewalk; and
- A robotic vacuum cleaner cleans the entire house by passing its vacuum head over every part of the floor.

All of these problems are characterized by a coverage region, a coverage tool, and a reachable region. The robot's objective is to move its coverage tool over the entire coverage region while staying within the reachable region. Although the reachable region may be larger than the coverage region (e.g. a spray painting robotic arm

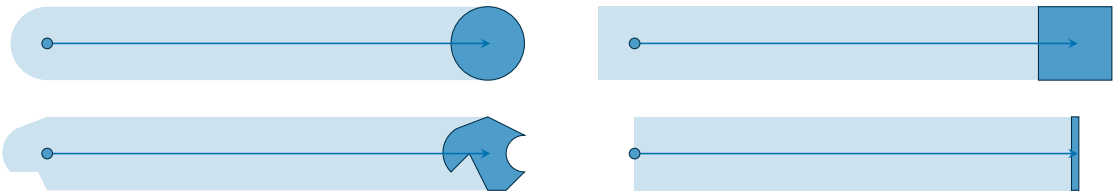


Figure 4.1: Regions covered by robots with circular, square, irregular, and straight line tools when moving along a straight path.

can usually move beyond the extent of the object it is painting), I will assume that the reachable and coverage regions are identical. The exact region traced out by a given robot depends on the footprint of its coverage tool (Figure 4.1). These coverage paths are all identical except for some small irregular regions at the start and end of the path.

The most common objective of coverage planning is to find the shortest coverage path. For most applications, this objective is flawed. An idealized robot—one that moves at a constant speed and can follow any path exactly—can cover a region most efficiently by following the shortest path. However, real robots cannot travel at constant speeds and cannot follow arbitrary paths. Most robots can accurately follow straight paths but have difficulty following a winding path exactly. If a robot tries to follow a winding coverage path but doesn't follow it exactly, it will miss spots. Additionally, real robots typically rely on a finite set of preprogrammed *behaviors* which make them more efficient at travelling along straight than along curved paths. These discrete behaviors generally mean that it is not possible to treat coverage planning as a kinodynamic planning problem where the goal is to minimize the coverage time. Instead, coverage planning is based on finding a polygonal path which minimizes how much the robot turns in addition to the length of its path. Since for many environments, many different paths all minimize length (Figure 4.2), it is important to also minimize turns.

In many applications, fewer turns also benefits the quality of coverage. A boat scanning the seafloor often cannot use any data obtained while turning as its laser

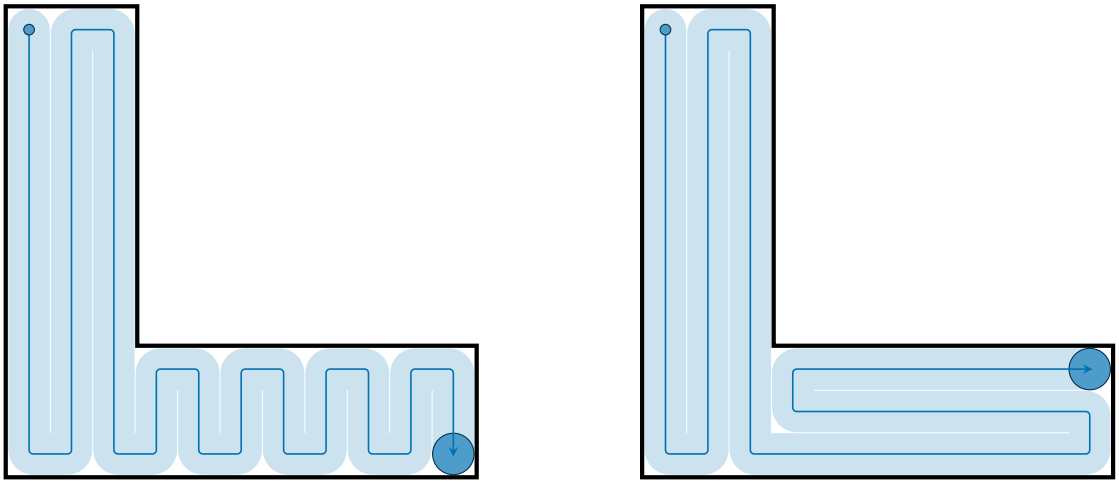


Figure 4.2: A coverage path with a single orientation (left) requires more turns than one with two orientations (right). Both paths have the same length.

scanner is angled. A spray paint nozzle does not leave a uniform layer of paint while following a curved path. An autonomous snowplow can control where the plowed snow ends up better when moving straight, and may end up piling some snow on the road if it turns with a lot of snow collected in front of its plow. As robotic vacuum cleaners tend to turn after bumping into an obstacle and often get stuck under obstacles that they bump into, minimizing turns means the robot is less likely to get stuck. These qualitative problems also motivate the need for a coverage plan with fewer turns.

In this chapter, I present a new coverage planning algorithm that explicitly considers turn-minimization and works for any polygonal environment. It minimizes turns using a novel asymptotically optimal partitioning heuristic which divides the environment into a minimal number of ranks that completely cover the environment. These *ranks* are long straight rectangles which are classified as either perimeter, horizontal, or vertical ranks (Figure 4.3). This rank partition is converted into a coverage path by solving a constrained version of the TSP using an existing solver. This approach can also be used for multirobot coverage using the exact same rank partition and then solving a constrained version of the minmax

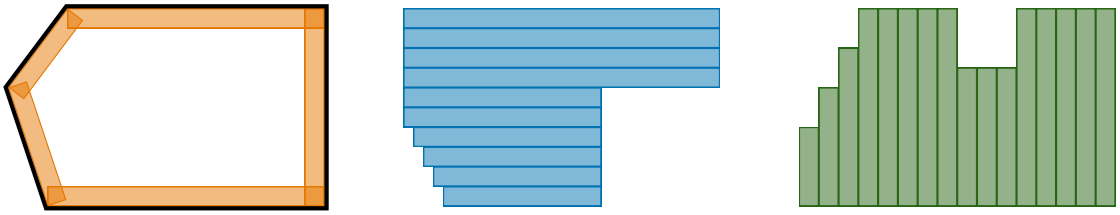


Figure 4.3: My coverage algorithm uses perimeter ranks (left) and interior ranks which are either horizontal (center) or vertical (right).

m-TSP using the algorithm from Chapter 3. My heuristic's computational runtime scales quadratically with the number of vertices in the polygon and is able to solve problems an order of magnitude larger than those solved by the most similar approach [29]. Furthermore, I have successfully used it to create coverage plans for teams of 1–5 robots in real environments that were mapped experimentally. This chapter is an expanded version of my paper "Turn-minimizing multirobot coverage" [188].

4.1 Related work

Two basic coverage strategies are the contour-parallel and direction-parallel paths [74] (Figure 2.9). In these strategies, the path either follows the environment's perimeter or moves back and forth in straight lines called *ranks*. For non-convex polygons, these strategies can be applied by first decomposing the environment into convex regions using a method such as the boustrophedon decomposition [38]. The order that the cells are covered by contour- or direction-parallel motion is determined by solving the travelling salesperson problem (TSP). Like the TSP, the problems of finding the shortest and time-minimal coverage paths are NP-hard [14].

Geometric decompositions form the basis of other coverage approaches [37, 65]. A decomposition consists of a set of smaller, simpler regions called *cells* and decompositions can be classified as exact or approximate based on the properties of

Table 4.1: Comparison of the cells of the two common types of coverage decompositions with the one used in this chapter.

Decomposition	Shape of cell	Size of cell	Number of cells	Dimension
Approximate	Square	Small	Many	0
Exact	Irregular	Large	Few	2
Ranks	Long rectangle	Medium	Medium	1

these cells (Table 4.1). Exact decompositions, such as the boustrophedon decomposition and its variants [1, 201], have a small number of large cells with irregular shapes. Approximate decompositions, such as Agmon *et al.* 's minimum spanning tree (MST) approach [2], use many small cells in a (usually square) grid. Approximate decompositions can be thought of having "zero-dimensional" cells as each cell can be contained in the footprint of a robot at a point. Exact decompositions, on the other hand, have "two-dimensional" cells as their cells are larger than the robot's footprint along both dimensions. My coverage algorithm uses a rank decomposition which does not fit neatly into these categories of exact and approximate decompositions. Its cells are long thin rectangles which are narrower than the robot's footprint along one direction, but longer than its footprint in the other direction and can thus be thought of as "one-dimensional" cells. As its dimension is intermediate between the exact and approximate decompositions, its cells are intermediate in size and the total number of cells is also intermediate.

These three types of decompositions also result in qualitatively different coverage paths (Table 4.2). Paths for an approximate decomposition are obtained by solving the TSP on the set of grid cells. As there are many small cells, the TSP will consider a very large number of possible paths—resulting in a high computational burden—and the best path will be minimal in repeat length, but usually has a large number of turns as it resembles a space filling curve. When using an exact decomposition, the path is obtained by connecting direction-parallel paths

Table 4.2: Comparison of the paths of the two common types of coverage decompositions with the one used in this chapter.

Decomposition	Length	Number of turns	Computational burden
Approximate	Minimal	Many	High
Exact	Slightly longer	Medium	Low
Ranks	Slightly longer	Minimal	Medium

on each cell by solving the TSP. The direction-parallel paths force the robot to follow an exact path for many parts of the environment giving the TSP less freedom, which results in a slightly longer path but a lower computational burden. These paths also tend to have fewer turns because the direction-parallel paths on each cell do not require as many turns as the winding paths typical of approximate decomposition approaches (Figure 4.4). My rank decomposition is constructed with the explicit objective of minimizing turns so the resulting path has fewer turns than either existing type of decomposition. As it minimizes turns, the paths are slightly longer than approximate decomposition paths; however, as the ranks are still connected by solving the TSP, the paths are typically not much longer. The computational burden—proportional to the number of cells—is intermediate. Furthermore, in a multirobot setting, the rank decomposition is computed before assigning ranks to robots so it also minimizes the team's total number of turns in multirobot coverage.

Existing coverage planners which attempt to minimize turns are based on exact decompositions. The two-dimensional cells of an exact decomposition can be covered in many different ways by using parallel ranks aligned with different directions. The turns needed to cover any cell can be minimized by using ranks parallel to the direction which minimizes the altitude of that cell [85] (Figure 2.13). For the correct decomposition, minimizing turns on each cell would result in minimizing turns for the whole environment; however, for most decompositions, this property would not be true. One way to find a decomposition with this property is to merge

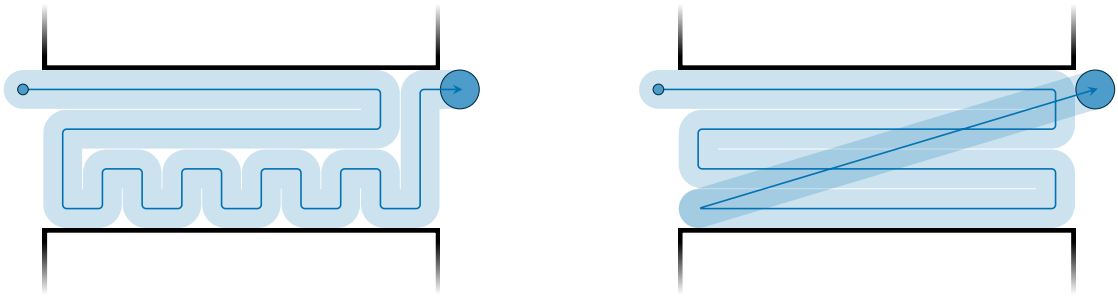


Figure 4.4: Approximate decompositions often result in winding paths (left) especially when travelling along a corridor whose width is an even multiple of the robot's width. In the same corridor, an exact decomposition's path (right) would use straight paths but have to make one additional redundant pass to get to the correct side of the corridor.

neighboring cells of a "sufficiently fine" decomposition into a coarser decomposition suitable for turn-minimization [85, 170]. The sufficiently fine decomposition can be obtained by extending all edges next to concave corners until they reach another edge (Figure 4.5 left). Cutting the polygon in this way creates an exponential number of cells with respect to the number of concave corners, and the optimization procedures for merging them require exponential time to compute. If instead of making two cuts at each concave vertex, a single cut is made somewhere in between the two edges [29, 49], a turn-minimizing exact decomposition can be found somewhat faster. Turn-minimizing coverage has been applied successfully to UAV applications [15, 120, 133] where turn-minimization is important because UAVs with fixed sensors cannot take useful measurements while turning.

Coverage time can be decreased by using more robots. If the environment is first divided up into regions with equal area, each robot can plan its coverage independently [22, 78]. This approach can be made more robust by replanning during the coverage mission to account for variable speeds [3] or changes in the environment [109, 156]. Alternatively, the robots can plan cooperatively using a modified boustrophedon decomposition [99] or MST-based strategy [95]. I am not aware of any existing multirobot coverage strategies for non-convex polygons that

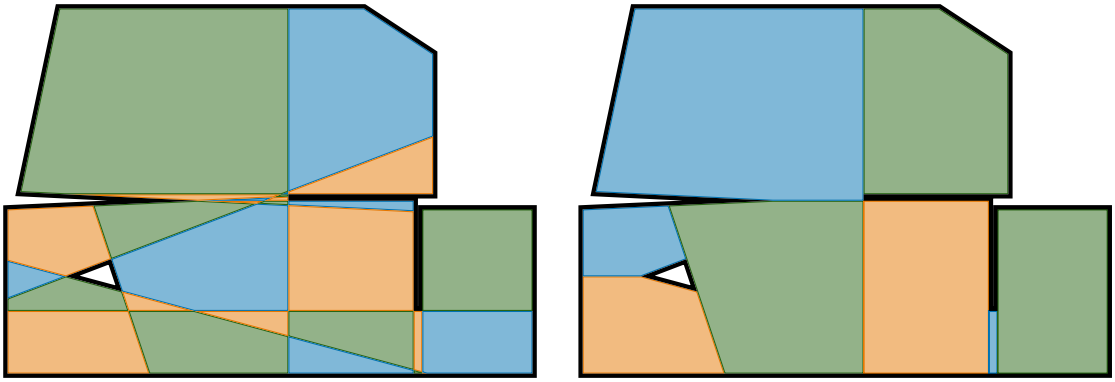


Figure 4.5: A fine decomposition—whose cells will be merged to obtain a turn-minimizing decomposition—can be obtained by cutting each vertex along lines extended from each edge adjacent to a concave corner (left). Making a single cut at each corner, between the two extended edges, can also produce a turn-minimizing decomposition (right).

use turn-minimization.

4.2 Partitioning the environment

The total time a robot takes to follow a path, including the time needed to slow down for turns, can be approximated by

$$t_{\text{total}} = \frac{\ell_{\text{path}}}{s_{\text{robot}}} + n_{\text{turn}} t_{\text{turn}},$$

where ℓ_{path} is the path length, s_{robot} is the robot's linear velocity, n_{turn} is the number of turns on the path, and t_{turn} is the time needed to make one turn including the time wasted decelerating and accelerating before and after it. A turn is considered any motion between two long straight segments of a robot's path, which usually are by an angle of 180° but may be other angles. Although turning time varies somewhat with the angle of the turn, we approximate the problem by using a fixed turning time because most of the turning time is spent accelerating and decelerating.

Since the covered area is equal to the tool width times the path length, a

complete coverage path's length is bounded by the environment's area divided by the robot's tool's width.

$$\ell_{\text{path}} \geq \ell_{\text{min}} = \frac{A_{\text{environment}}}{w_{\text{tool}}}$$

where $A_{\text{environment}}$ is the environment's area and w_{tool} is the robot's tool's width. This path length is achieved by any path which covers each point of the robot's environment, $\mathcal{Q} \subset \mathbb{R}^2$, exactly once. Any paths with no redundant coverage have the same path lengths but can vary drastically in their number of turns (Figure 4.2). Many robots cannot make precise turns quickly so t_{turn} can be quite large and it is also important to minimize the number of turns.

On a coverage path, the number of turns is equal to the path's number of straight line segments. Each straight line segment results in the coverage of a long thin rectangle called a *rank*. My goal, therefore, is to partition the environment into a minimum number of ranks which cover the entire space.

Problem 4.1. *For a polygonal environment, $\mathcal{Q} \subset \mathbb{R}^2$, find a set of unit width rectangles, $\mathcal{R}$, such that $\cup_{r\in\mathcal{R}} r = \mathcal{Q}$ while minimizing $|\mathcal{R}|$.*

In Problem 4.1, the robot's environment is represented by a polygon with holes, $\mathcal{Q} \subset \mathbb{R}^2$. The problem is scaled so that the robot's tool has unit width tool and the coverage ranks are represented by unit width rectangles which may be rotated.

Problem 4.1 is continuous-space version of the set cover problem [100] where $\mathcal{Q}$ is the set to be covered and covering set contains all unit width rectangles, of any length or angle, which are contained within $\mathcal{Q}$. If the rectangles are not allowed to overlap, then Problem 4.1 is a continuous-space version of the set partition problem. Both of these problems are NP-hard [100] when defined for finite sets. On the other hand Problem 4.1 involves an uncountable set of all unit width rectangles contained within $\mathcal{Q}$ so we will use a custom heuristic, which uses the

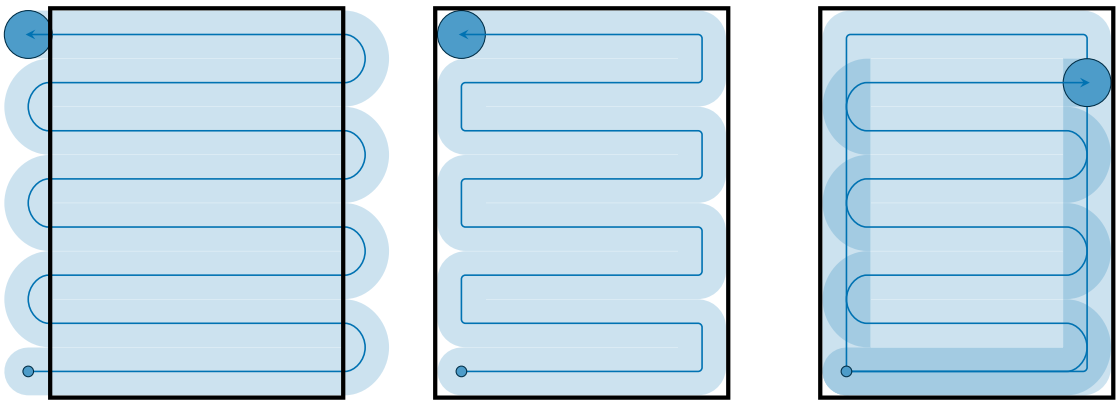


Figure 4.6: If a robot can travel outside of the boundary of the coverage region, it can guarantee complete coverage by turning outside of the coverage region (left). In many applications, the boundary of the coverage region is a physical barrier, so the robot must turn inside the coverage region and will miss some small regions near the boundary (center). By including perimeter ranks, the robot achieves near-complete coverage while turning within the coverage region (right).

topology of $\mathcal{Q} \subset \mathbb{R}^2$, when solving it.

4.2.1 Perimeter following

An environment's perimeter is difficult to cover because the robot needs to turn around when it reaches the perimeter. If the robot can travel outside the perimeter, it can achieve complete coverage by turning around outside the environment (Figure 4.6). If it is constrained to the environment, it must follow ranks along the perimeter to achieve near perfect coverage. Due to the shape and size of the robot, some small regions in the corners cannot be covered by any path. Although the precise geometry of these corner regions depends on the physical size and shape of the robot and its tool, these unreachable regions are always small. We therefore assume that the polygon, $\mathcal{Q}$, in Problem 4.1 has had these small unreachable areas removed.

For problems where the robot is constrained to the environment, we always include one *perimeter rank* per edge of the perimeter (Figure 4.7, Algorithm 4.1). If the angle the edge makes with the next edge is between 90–180°, the adjacent

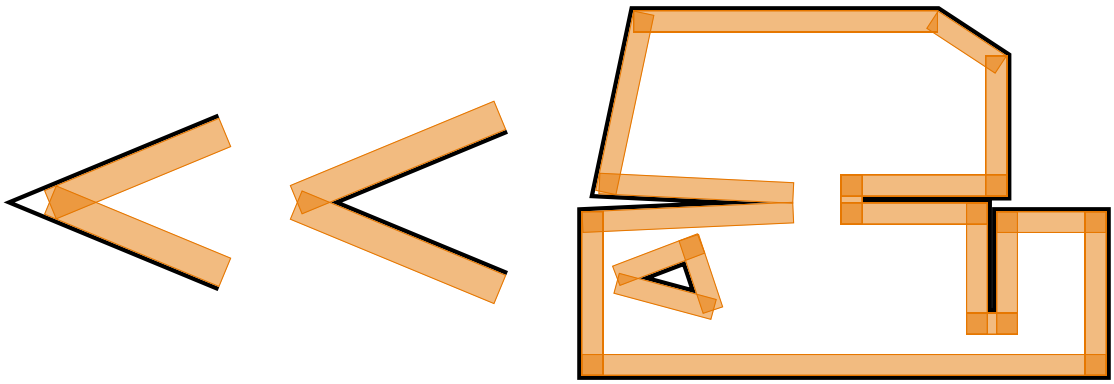


Figure 4.7: Perimeter ranks adjacent to corners with angle less than 90° are shortened to remain inside the coverage region (left). Perimeter ranks adjacent to corners with angles greater than 180° are lengthened to ensure complete coverage near the corner (center). These perimeter ranks result in near perfect coverage of all locations within a distance of one robot width from the boundary (right).

ranks end exactly at the corner. If the angle is less than 90°, the rank is shortened to be contained within the environment. If the angle is greater than 180°, the rank is extended by the width of the robot to prevent missed coverage near the corner.

Algorithm 4.1: Perimeter ranks

Input: Polygonal region, $\mathcal{Q} \subset \mathbb{R}^2$
Output: Set of perimeter ranks, $\mathcal{R}_{\text{per}}$

$\mathcal{R}_{\text{per}} \leftarrow \{\}$ `/* Set of perimeter ranks */`
for edge $e \in \partial\mathcal{Q}$ **do**
  $r \leftarrow$ unit width rectangle adjacent to e
  for vertex $v \in$ `endpoints`(e) **do**
    $\theta \leftarrow$ angle between edges of $\partial\mathcal{Q}$ incident to v
    if $\theta > 180°$ **then**
      Extend the end of r near v by one unit length
    else if $\theta < 90°$ **then**
      Shorten the end of r near v so that $r \subset \partial\mathcal{Q}$
  $\mathcal{R}_{\text{per}} \leftarrow \mathcal{R}_{\text{per}} \cup r$
return $\mathcal{R}_{\text{per}}$

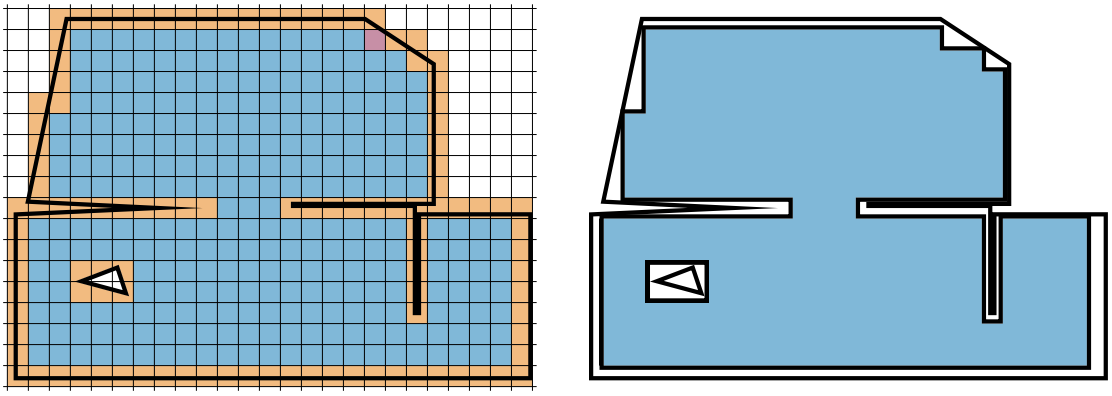


Figure 4.8: Overlayed grid (left) used to define the rectilinear contraction (right). Orange cells are part of the polygon under both definitions; red cells are part of the polygon only under one of the definitions; blue cells are never part of the contraction.

4.2.2 A rectilinear contraction

Regions of $\mathcal{Q}$ not covered by perimeter ranks need to be covered by *interior ranks*. If $\mathcal{R}_{\text{per}}$ is the set of perimeter ranks, then the region that remains be covered is $\mathcal{Q}_{\text{int}} = \mathcal{Q} \setminus \cup_{r \in \mathcal{R}_{\text{per}}} r$. Coverage can be achieved by covering any region $\mathcal{Q}_{\text{rect}}$ with $\mathcal{Q}_{\text{int}} \subseteq \mathcal{Q}_{\text{rect}} \subseteq \mathcal{Q}$. We will choose $\mathcal{Q}_{\text{rect}}$ to be a rectilinear polygon with integer side lengths. For an integer rectilinear polygon, Problem 4.1 always has a disjoint solution consisting of some vertical ranks and some horizontal ranks. Most indoor environments are roughly rectilinear anyways so they can be efficiently covered by these two directions. Although some environments, such as the agricultural fields in [148] are highly non-rectilinear or even curved, if a robot is not able to precisely follow curved paths or make irregular turns, a rectilinear coverage approach may still be more appropriate for these problems.

The rectilinear contraction, $\mathcal{Q}_{\text{rect}}$, can be obtained by overlaying a unit width grid on top of $\mathcal{Q}$ and $\mathcal{Q}_{\text{int}}$. This grid should be rotated to maximize the length of perimeter that aligns with the grid axes. Once a grid has been chosen, the contracted rectilinear polygon can be computed in one of two ways (Figure 4.8):

1. The largest possible $\mathcal{Q}_{\text{rect}} \subseteq \mathcal{Q}$ is the union of all grid cells fully contained

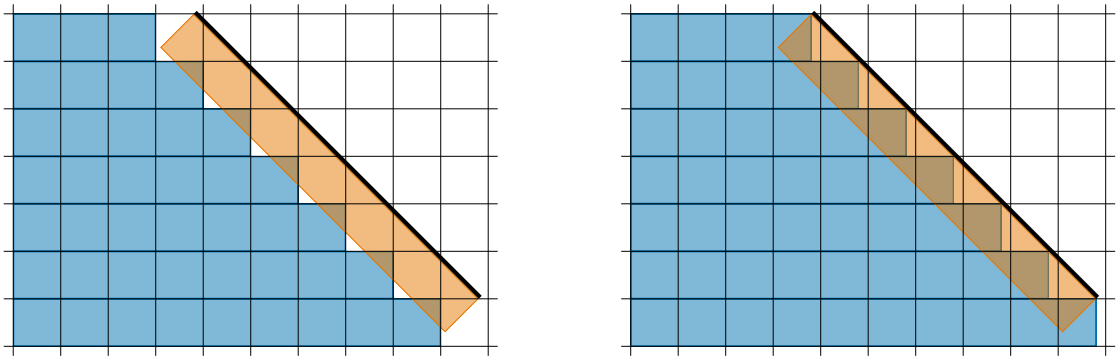


Figure 4.9: If the interior ranks all have integer lengths, there may be some small regions that are not covered by perimeter or interior ranks (left). By extending the interior ranks until the wall, we guarantee that everything gets covered (right).

in $\mathcal{Q}$; or

2. The smallest possible $\mathcal{Q}_{\text{rect}}$ with $\mathcal{Q}_{\text{int}} \subseteq \mathcal{Q}_{\text{rect}}$, is the union of all grid cells fully or partially contained in $\mathcal{Q}_{\text{int}}$.

If a cell is partially contained in $\mathcal{Q}_{\text{int}}$ but not fully contained in $\mathcal{Q}$ (red cell in Figure 4.8), these two definitions will be different. I will use the first definition so that the region covered by interior ranks is fully contained in the coverage region, in case its boundary represents a physical barrier. As this choice may result in small missed regions near the problematic cells, the interior ranks will later be extended to reach the boundary of the environment (Figure 4.9). Extending the interior ranks guarantees no missed coverage between perimeter and interior ranks.

Algorithm 4.2: Rectilinear contraction

Input: Polygonal region, $\mathcal{Q} \subset \mathbb{R}^2$
Output: Rectilinear interior region, $\mathcal{Q}_{\text{rect}}$

```
Q_rect ← {}                                   /* Rectilinear interior region */
Compute bounding box of Q with integer coordinates
for unit square in bounding box do
    if unit square is fully contained in Q then
        Q_rect ← Q_rect ∪ unit square
return Q_rect
```

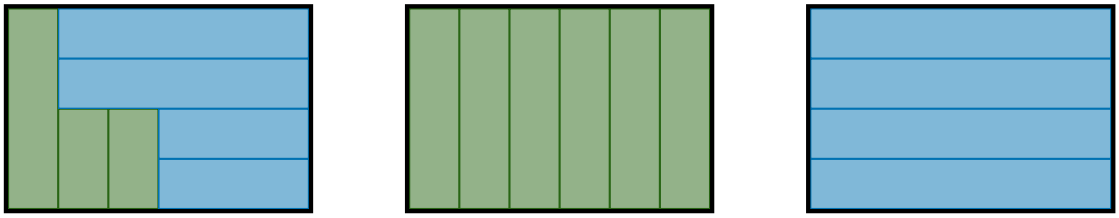


Figure 4.10: Covering a rectangle with two directions of ranks (left) always requires more ranks than covering with a single direction. Coverage parallel to the rectangle's short edge (right) requires more ranks than the minimal rank partition which only uses ranks parallel to the rectangle's long edge (right).

4.2.3 A coarse checkerboard partition

It is not obvious how an arbitrary rectilinear polygon—potentially with holes—can be partitioned into a minimal set of ranks. As we are interested in an exact disjoint partition, we cannot use diagonal ranks and all the ranks will have to be either horizontal or vertical. For the simpler case of a rectangle (the only kind of convex rectilinear polygon), finding the minimal rank partition is trivial. It should be covered by a single direction of ranks which are parallel to its longest side (Figure 4.10). In the special case of a square, both directions of coverage result in the same number of ranks. For any other rectangle, the length and width are different so there is a unique minimal rank partition.

Based on the simplicity of partitioning rectangles into ranks, an obvious way to partition a rectilinear polygon to first partition it into rectangles. How well this procedure will work depends on how the rectilinear polygon is partitioned into rectangles. There are many possible rectangle partitions so I will use the following method of partitioning an arbitrary rectilinear polygon into ranks:

1. Partition the rectilinear polygon into a set of disjoint rectangles.

2. Choose a direction of coverage for each rectangle in the partition. This direction may depend on the directions of nearby rectangles and is not necessarily the optimal direction for the same rectangle in isolation.

3. Merge adjacent rectangles with the same direction into one larger rectangle requiring fewer ranks.

4. Cover each of the large rectangles with a single direction of ranks.

How well this method works depends on the initial rectangle partition (Figure 4.11). If this partition is too coarse, it may not be possible to obtain an optimal set of ranks by assigning a single direction to each rectangle. On the other hand, a very fine partition, such as the unit grid can be used to find an optimal partition, however it has a high computational burden as it has many rectangles. The ideal initial partition is somewhere in between these two extremes—fine enough that only one direction per rectangle is needed to find an optimal set of ranks, but coarse enough that the computational burden is low.

A coarse partition may need multiple directions on a single rectangle if some but not all of that rectangle's ranks can be merged with the neighbors' ranks. This situation occurs when the two rectangles do not share a full edge. In general, if all of a rectangle's neighbors share an entire edge with it, the optimal rank decomposition has a single orientation on that rectangle. This observation motivates us to use a *checkerboard partition* where every rectangle has the same width as its vertical neighbors and same height as its horizontal neighbors.

Checkerboard partitions are closely related to the polygon's concave vertices. In any checkerboard partition, each edge of a rectangle extends until it intersects with an orthogonal edge of the rectilinear polygon's boundary. As the edges of the rectilinear polygon are guaranteed to be edges of some rectangle in the partition, edges incident to concave vertices must be extended in any checkerboard partition. The coarsest checkerboard partition can be obtained by using only these edges (Figure 4.12, Algorithm 4.3). We will use this partition when computing the rank decomposition. The number of rectangles in this partition is proportional to the square of the number of convex vertices and is usually much smaller than a grid

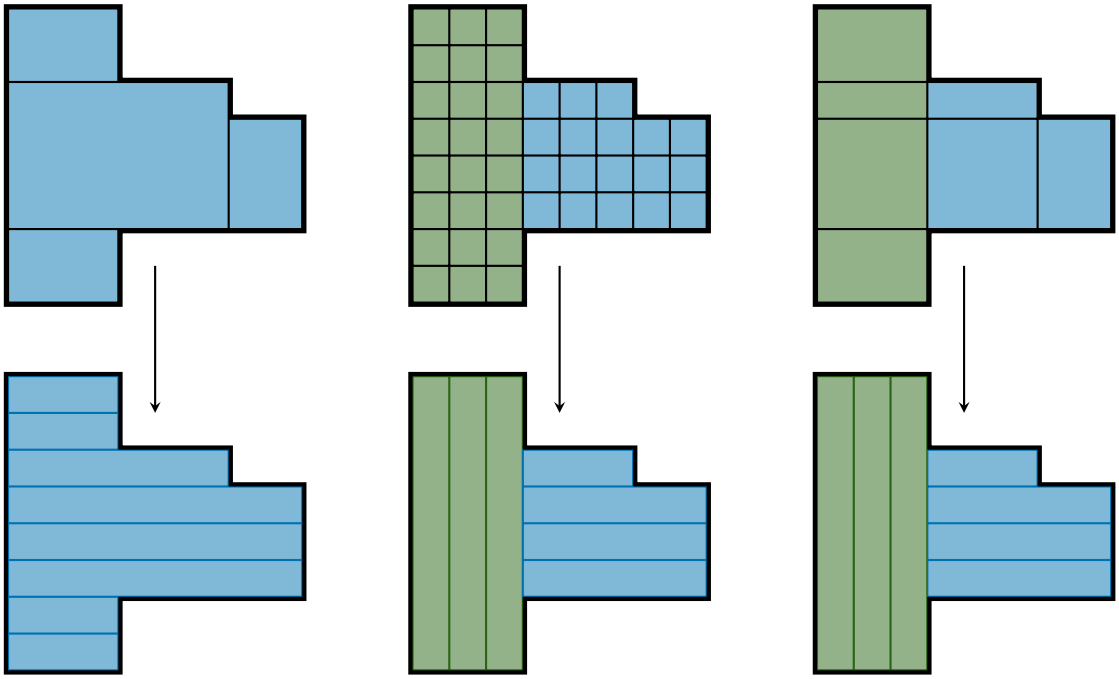


Figure 4.11: Three different partitions of a rectilinear polygon into rectangles. If the partition is too coarse (left), it is not necessarily possible to find the optimal rank partition by assigning one orientation to each rectangle. If the partition is too fine (center), finding the optimal rank partition will take too long. The checkerboard partition (right) is the coarsest partition which is guaranteed to only need a single orientation per rectangle when computing the optimal rank partition.

partition whose number of rectangles equals the area of the rectilinear polygon.

4.2.4 Orienting the rectangles

Assigning an orientation—whether the direction of coverage is horizontal or vertical—to each rectangle of the checkerboard partition defines a rank partition. The objective is to assign orientations to minimize the number of ranks and solve Problem 4.1. For a checkerboard partition with n rectangles, there are 2^n possible assignments so it is not feasible to check them all. Instead, I use a heuristic which creates a locally optimal assignment.

Local optimality means that the number of ranks from the assignment cannot be improved by changing the orientation of a single rectangle. Rectangles in a

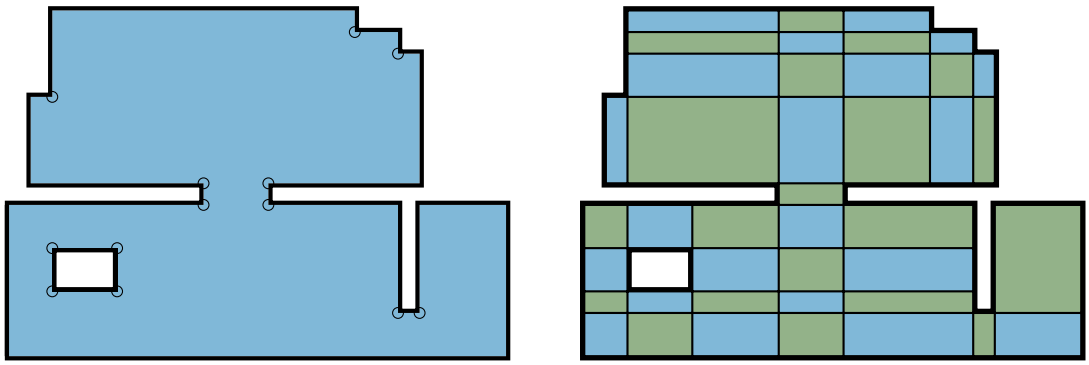


Figure 4.12: Concave vertices (left) define the coarsest checkerboard partition (right). The partition is obtained by extending each edge incident to a concave vertex until it intersects with another edge of the polygon's boundary.

Algorithm 4.3: Checkerboard partition

Input: Rectilinear polygon, $\mathcal{Q}_{\text{rect}} \subset \mathbb{R}^2$
Output: Set of rectangles, $\mathcal{H}$, which are a disjoint partition of $\mathcal{Q}_{\text{rect}}$

$\mathcal{H} \leftarrow \mathcal{Q}_{\text{rect}}$ `/* Checkerboard partition */`
for vertex $v \in$ `corners`$(\partial\mathcal{Q})$ **do**
  if angle at v is greater than $180°$ **then**
    $v_0, v_1 \leftarrow$ corners of $\partial\mathcal{Q}$ adjacent to v
    for vertex $v_i \in \{v_0, v_1\}$ **do**
      $\theta \leftarrow$ direction from v to v_i
      $v' \leftarrow$ intersection of ray leaving v in direction $-\theta$ with $\partial\mathcal{Q}_{\text{rect}}$
      $e \leftarrow$ edge from v to v'
      for rectilinear polygon $h \in H$ **do**
        if e bisects h **then**
          $h_1, h_2 \leftarrow$ polygons obtained by cutting h along e
          $\mathcal{H} \leftarrow \mathcal{H} \cup \{h_1, h_2\} \setminus \{h\}$
return $\mathcal{H}$

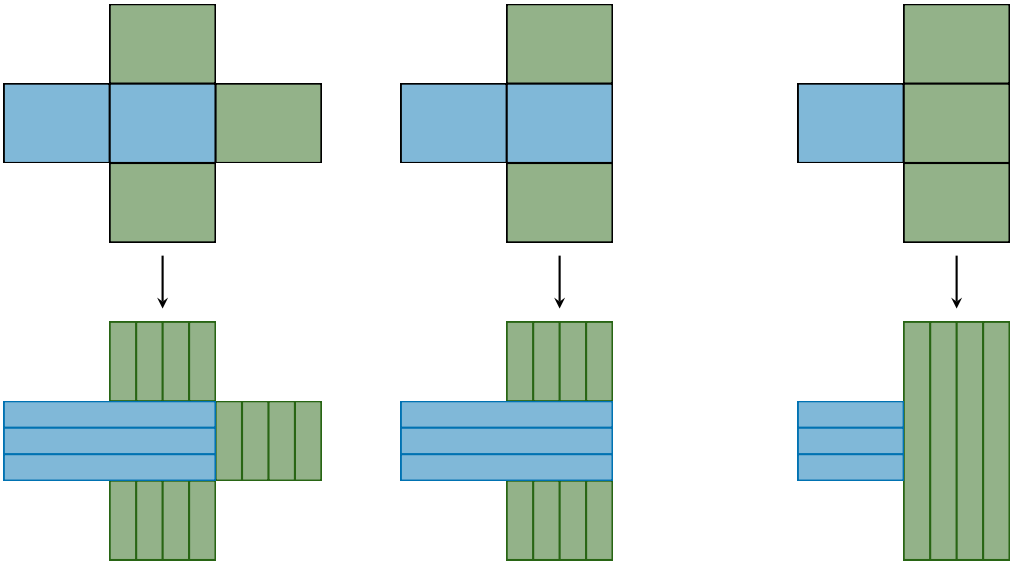


Figure 4.13: A horizontally oriented cell can merge its ranks with its left- and right- neighbors if they are also oriented horizontally (left). If one of these neighbors is oriented vertically, it is incompatible for merging, and from the perspective of the central rectangle, the situation is equivalent to one where only compatible neighbors exist (center). Based on which compatible neighbors a rectangle has, there may be a different orientation which is locally optimal (right).

checkerboard partition can have up to 4 neighbors and their ranks can be merged with the ranks of compatible neighbors (Figure 4.13). Two neighboring rectangles are *compatible* if the direction between the rectangles equals both rectangles' rank directions. Treating a given rectangle's neighbor orientations as fixed, the locally optimal orientation for the given rectangle is the orientation which minimizes the total number of ranks needed to cover it and its neighbors. In a locally optimal assignment, the orientations of all the cells are simultaneously locally optimal.

The locally optimal orientation maximizes the number of ranks merged minus the number of new ranks added. Up to symmetry, there are six possible cases of how many compatible neighbors a cell has (Figure 4.14):

(a) **No compatible neighbors**: The optimal orientation is aligned with the longest edge to minimize new ranks added.

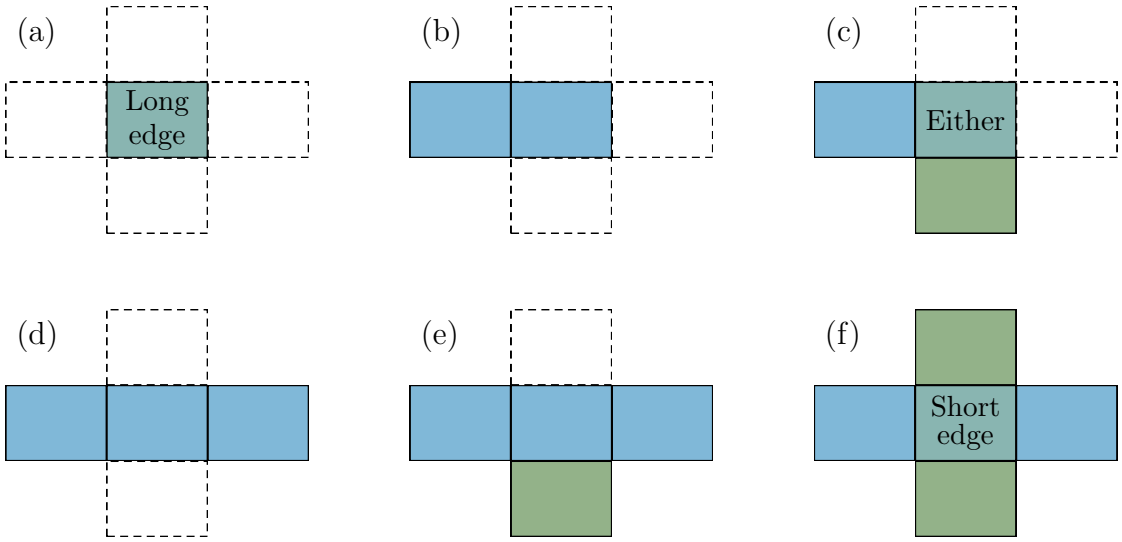


Figure 4.14: Possible cases for a rectangle's four neighbors and their orientations. Blue represents horizontal ranks; green represents vertical ranks. If the central rectangle has more compatible neighbors in one direction than the other (cases (b), (d), and (f)) it should be oriented along that direction. If it has the same number of compatible neighbors in both directions (cases (a), (c), and (f)), it may be oriented horizontally or vertically depending on its dimensions.

(b) **One compatible neighbor**: The optimal orientation is aligned with that neighbor so no new ranks are added.

(c) **Two compatible neighbors in different directions**: Both orientations are optimal and neither would add new ranks.

(d) **Two compatible neighbors in the same direction**: The optimal orientation is aligned with both neighbors to reduce the total number of ranks.

(e) **Three compatible neighbors**: The optimal orientation is aligned with the direction in which it has two neighbors to reduce the total number of ranks.

(f) **Four compatible neighbors**: The optimal orientation is aligned with the shorter edge to maximize the number of ranks merged.

The criteria for local optimality can also be used to convert any assignment into

a locally optimal one by repeatedly flipping the orientations of rectangles whose orientations are not locally optimal. Flipping the orientation causes a strict decrease in the cost by the difference in number of ranks needed for each orientation—an integer. As the cost is bounded below by the cost of the globally optimal assignment, this procedure is guaranteed to terminate after a finite number of steps.

In case (c), where a cell has one compatible neighbor in each direction, both orientations are equivalent. With either orientation, all of the rectangle's ranks will be merged with the same number of ranks in a neighboring rectangle resulting in no change in the total number of ranks due to this rectangle. At first, I thought this case was very uninteresting, and assumed that nothing needed to be done for these rectangles. After all, changing its orientation doesn't affect the total number of ranks needed. What I initially overlooked is that flipping this rectangle's orientation changes which rectangles it is a compatible neighbor for. Both of its neighbors will now be a different one of the six cases and one of them may be able to then change it's orientation in a way that actually improves the total number of ranks! Case (c) essentially lets us escape one local minimum and find a better local minimum by first making some neutral moves (Figure 4.15). As rectangles are more likely to merge if they have the same orientation, the best way to exploit this trick is to always make case (c) rectangles have the same orientation, called the *bias*. Once we find a local minimum where all the case (c) rectangles have the same bias, we can change the bias and potentially get an even better local minimum. We can also treat squares in case (a) or (f) as if they are case (c) as the optimal orientations for these cases depends on the side lengths and both orientations are optimal when the sides have the same length.

These two procedures are the basis of a heuristic (Algorithm 4.4) for generating locally optimal solutions to Problem 4.1. First, it chooses a random orientation for each rectangle (line 1). In each round of the algorithm (lines 3–20), the orientations of rectangles are repeatedly flipped if not locally optimal or set to the *bias* if there

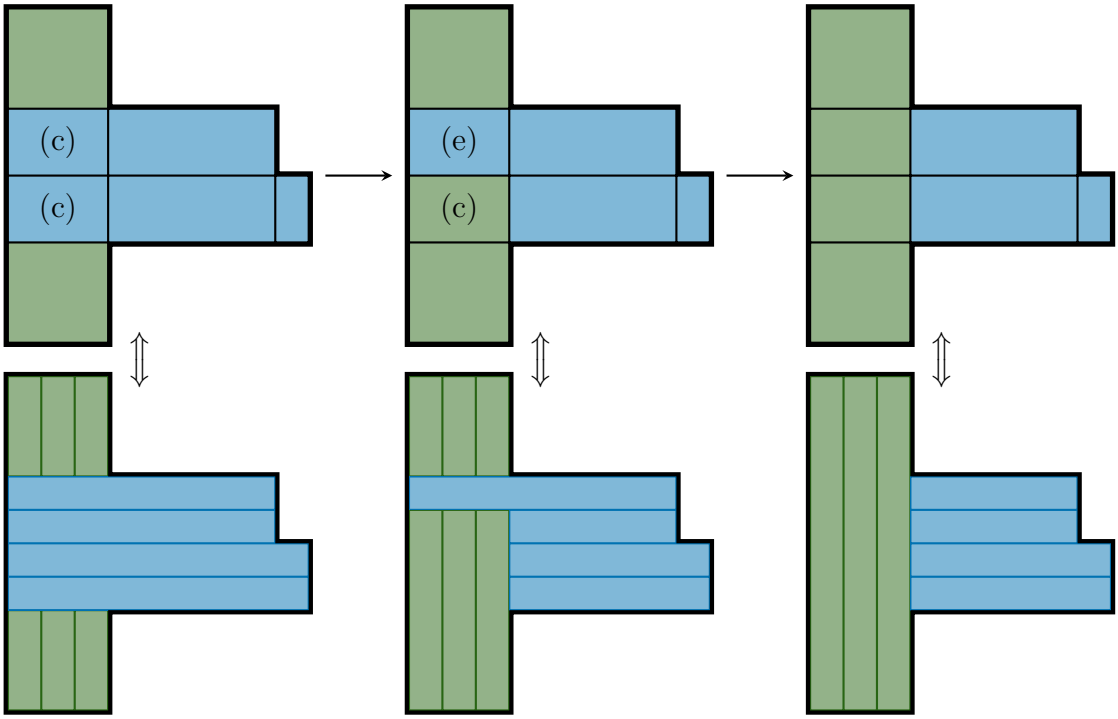


Figure 4.15: A locally optimal assignment cannot be improved by changing the orientation of any single cell (left). However, if any cell has case (c), its orientation can be flipped to obtain a different assignment with the same number of ranks (center). Flipping a case (c) rectangle can change the case of other rectangles, and so this new assignment is not necessarily a local minimum. From this new assignment, it may be possible to change the orientation of another rectangle and find an assignment with fewer ranks (right).

are two locally optimal orientations. The bias (line 2) is fixed in each round and is used for case (c) and for cases (a) and (f) if the rectangle is square as both orientations are optimal (line 9). By using a bias we change rectangles' orientations without changing the cost which may enable a different cell to flip later to decrease the cost. In each round, we keep track of which rectangles have already been checked (line 16) and uncheck rectangles if their neighbor flips (lines 10 and 13). Once all rectangles have been checked, the bias is flipped (line 18) and a new round begins if any improvements were made in the previous round. Improvements are defined as flips which decrease the cost of the assignment (line 15). The algorithm terminates after a round where no improvements were made

(line 20).

Algorithm 4.4: Orient rectangles

Input: Checkerboard partition, $\mathcal{H}$
Output: Checkerboard partition, $\mathcal{H}$, with optimized orientations of rectangle

```
 1 Θ ← {horizontal, vertical}                    /* Possible orientations */
 2 for rectangle h ∈ H do
 3     θ(h) ← random orientation in Θ
 4 bias ← random orientation in Θ
 5 improved ← true
 6 while improved do
 7     Set rectangles in H to unchecked
 8     improved ← false
 9     while there are unchecked rectangles do
10         h ← random unchecked rectangle in H
11         Θ*(h) ← locally optimal orientations for h
12         if θ(h) ∉ Θ*(h) then          /* Orientation is not optimal */
13             Flip θ(h)
14             Set h's neighbors to unchecked
15             improved ← true
16         else if (|Θ*(h)| = 2) and (θ(h) ≠ bias) then
17             Flip θ(h)
18             Set h's neighbors to unchecked
19         Set h to checked
20     if improved then
21         Flip bias
22 return H
```

If a different bias is used in the last round of Algorithm 4.4, different locally optimal assignments with the same cost may be returned (Figure 4.16). Algorithm 4.4 is guaranteed to reach a local optimum, but not the global optimum. As each iteration of the innermost loop (lines 7 to 17) can be performed in constant time, the algorithm runs very fast and can be repeated multiple times to increase the probability of finding the global optimum (Figure 4.17).

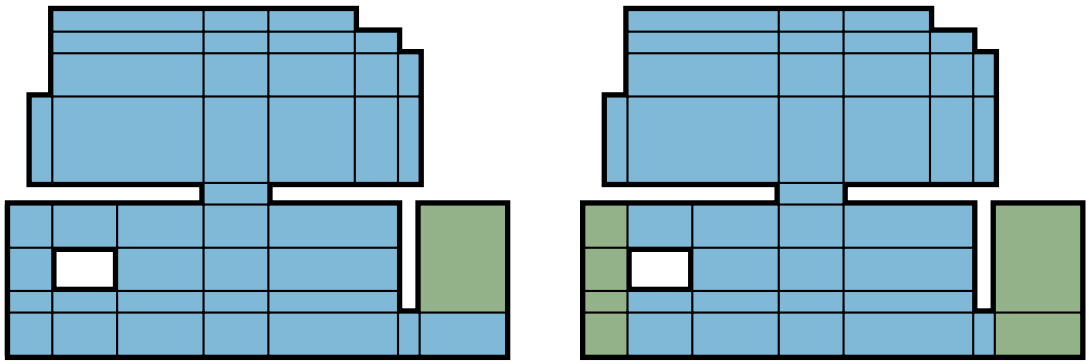


Figure 4.16: Optimal orientations for the rectangles (blue is horizontal; green is vertical) in a checkerboard partition which were obtained using Algorithm 4.4. Both solutions result in the same number of ranks. The left solution was optimized with a horizontal bias in the final round of Algorithm 4.4; the right solution finished with a vertical bias.

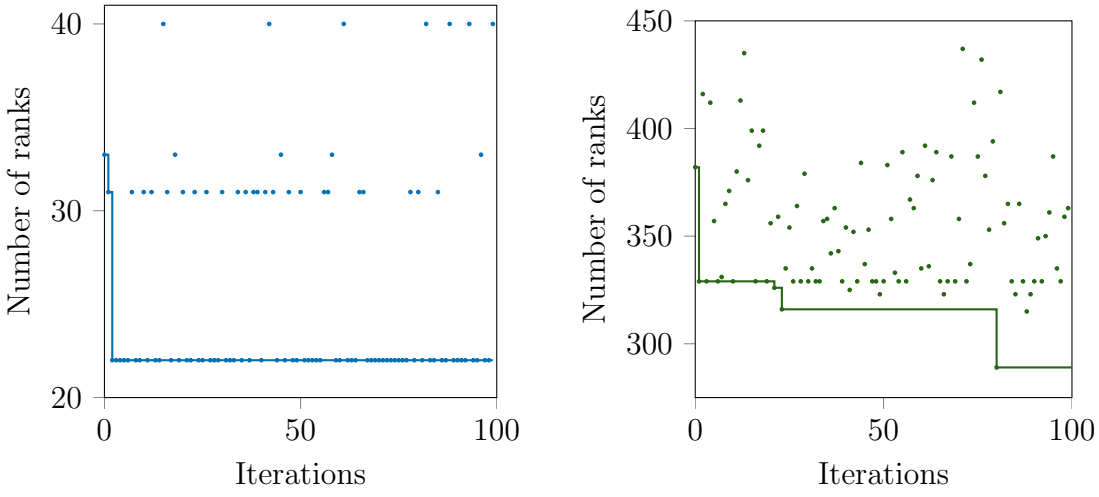


Figure 4.17: Improvement in solution quality when iterating Algorithm 4.4 for the example environment in Figure 4.7 (left) and the real environment in Figure 4.23 (right). Points represent the number of ranks returned in each round of the algorithm and the solid line represents the best number of ranks found so far.

Algorithm 4.5: Interior ranks

Input: Coverage region, $\mathcal{Q}$; interior region, $\mathcal{Q}_{\text{rect}}$; and checkerboard partition, $\mathcal{H}$

Output: Minimal set of interior ranks, $\mathcal{R}_{\text{int}}$

```
1  H_hor ← rectangles of H with oriented horizontally
2  H_ver ← rectangles of H with oriented vertically
3  Merge rectangles in H_hor which share a left or right edge
4  Merge rectangles in H_ver which share a top or bottom edge
5  R_int ← {}                                   /* Set of interior ranks */
6  for horizontal rectangle h ∈ H_hor do
7      R_hor ← ranks obtained by slicing h horizontally
8      for horizontal rank r ∈ R_hor do
9          if left_edge(r) touches ∂Q_rect then
10             Extend r left to ∂Q
11         if right_edge(r) touches ∂Q_rect then
12             Extend r right to ∂Q
13     R_int ← R_int ∪ R_hor
14 for vertical rectangle h ∈ H_ver do
15     R_ver ← ranks obtained by slicing h vertically
16     for vertical rank r ∈ R_ver do
17         if top_edge(r) touches ∂Q_rect then
18             Extend r up to ∂Q
19         if bottom_edge(r) touches ∂Q_rect then
20             Extend r down to ∂Q
21     R_int ← R_int ∪ R_ver
22 return R_int
```

4.2.5 The final rank partition

The locally optimal assignment of orientations for the checkerboard partition can be converted into a rank partition which solves Problem 4.1 for $\mathcal{Q}_{\text{rect}}$. First, adjacent compatible neighbors are merged into larger rectangles. These rectangles are sliced along their long axes into unit width rectangles which are the ranks of the partition which solves Problem 4.1 for $\mathcal{Q}_{\text{rect}}$ (Figure 4.18 left). These ranks are extended to the perimeter of $\mathcal{Q}$ to get the interior ranks that, together with the perimeter ranks from Subsection 4.2.1, solve Problem 4.1 on $\mathcal{Q}$ (Figure 4.18 right). Extending the interior ranks guarantees that the combination of perimeter

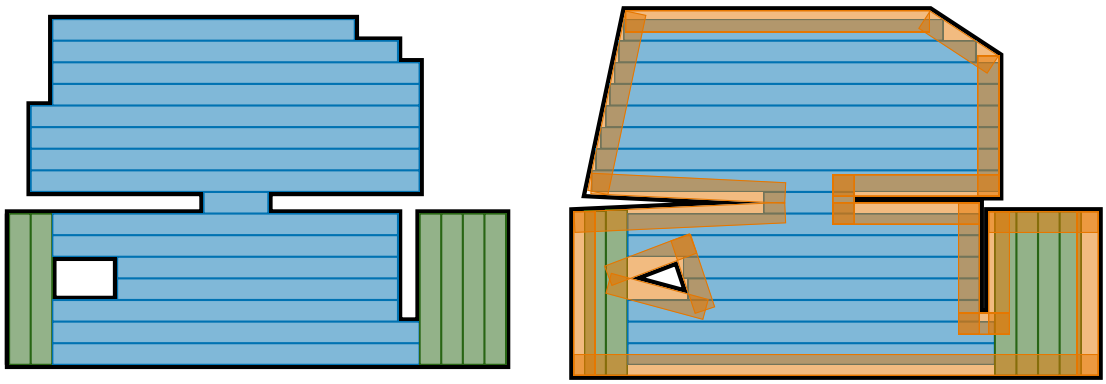


Figure 4.18: The interior ranks cover the rectilinear polygon (left) and the combination of perimeter and extended interior ranks cover the whole environment (right).

and interior ranks covers the entirety of $\mathcal{Q}$ (assuming every portion is reachable given the robot's shape and size). The overall algorithm (Algorithm 4.6) therefore produces a locally optimal feasible solution to Problem 4.1. By using different initial orientations in the inner loop (lines 5–10), after many iterations the algorithm finds a global optimum almost surely and it is therefore asymptotically optimal.

Algorithm 4.6: Rank Partition

Input: Polygonal region, $\mathcal{Q} \subset \mathbb{R}^2$; and number of iterations, $n_{\text{iteration}}$
Output: Minimal set of ranks, $\mathcal{R}$ which cover $\mathcal{Q}$

1 $\mathcal{R}_{\text{per}} \leftarrow$ set of perimeter ranks of $\mathcal{Q}$ `/* Algorithm 4.1 */`
2 $\mathcal{Q}_{\text{rect}} \leftarrow$ rectilinear contraction of $\mathcal{Q}$ `/* Algorithm 4.2 */`
3 $\mathcal{H} \leftarrow$ checkerboard partition of $\mathcal{Q}_{\text{rect}}$ `/* Algorithm 4.3 */`
4 $n^*_{\text{rank}} \leftarrow \infty$ `/* Best number of ranks */`
5 **for** iteration $i \in \{1, \dots, n_{\text{iteration}}\}$ **do**
6 Optimize orientations for $\mathcal{H}$ `/* Algorithm 4.4 */`
7 $\mathcal{R}_{\text{int}} \leftarrow$ interior ranks corresponding to $\mathcal{H}$ `/* Algorithm 4.5 */`
8 **if** $|\mathcal{R}_{\text{int}}| + |\mathcal{R}_{\text{per}}| < n^*_{\text{rank}}$ **then**
9 $n^*_{\text{rank}} \leftarrow |\mathcal{R}_{\text{int}}| + |\mathcal{R}_{\text{per}}|$
10 $\mathcal{R} \leftarrow \mathcal{R}_{\text{int}} \cup \mathcal{R}_{\text{per}}$
11 **return** $\mathcal{R}$

4.2.6 Generalizations to other spaces

Both Problem 4.1 and Algorithm 4.6 have been developed for coverage of two-dimensional Euclidean space. For other applications, such as painting curved automotive parts [11], a robot may be required to cover some curved two dimensional space. These spaces can be described as non-Euclidean two-dimensional manifolds embedded in three-dimensional space. Using an appropriate atlas, such a manifold can be locally transformed to a subset of $\mathbb{R}$ and Algorithm 4.6 could be applied on this transformed space and the solution transformed back to the manifold. However, as the charts of an atlas do not, in general preserve distance, points that are within a unit width rectangle in the transformed Euclidean space may not be within a unit width region on the original manifold. For this reason, planning will work better using charts which do not heavily distort distances, and may benefit from using a narrow tool width to ensure that adjacent rectangles indeed overlap on the manifold.

In other applications, a robot may be required to cover three-dimensional Euclidean space. This problem is much harder because the region covered by the robot travelling in a straight line is a prism with a base whose shape depends on the geometry of the robot's tool. A circular tool would result in cylindrical coverage regions; a square tool would result in a cuboid coverage region; and an irregular tool would result in a very complex coverage region. These differences mean that a three dimensional version of Algorithm 4.6 based on a rectilinear polyhedron would not necessarily work since the robot does not necessarily cover cuboids while following straight paths.

4.3 Connecting ranks into paths

The ranks produced by Algorithm 4.6 are a set of simple coverage tasks—just moving along a straight path. If a robot, or team of robots performs all of these

tasks they will cover the whole environment. The best order for a single robot to complete these coverage tasks as quickly as possible can be determined by solving the 1-TSP. Similarly, the best strategy for a team of m robots to complete these tasks can be determined by solving the m-TSP. By solving the appropriate version of the TSP, we obtain a set of coverage paths (Algorithm 4.7) which minimizes total coverage time for the team, including the time needed to turn. Typically, these paths are computed based on a set, $\mathcal{L} = \{(q_1, q_1'), \dots (q_m, q_m')\}$, of fixed start and end points for each robot. The start point, $q_i \in \mathcal{Q}$, is the robot's current position and the end point, $q_i' \in \mathcal{Q}$, is the location of a charging station or depot where the robot typically stays in between coverage missions.

Algorithm 4.7: Plan coverage paths

Input: Polygonal region, $\mathcal{Q} \subset \mathbb{R}^2$; and start/end locations, $\mathcal{L} = \{(q_1, q_1'), \dots, (q_m, q_m')\}$

Output: Set of coverage paths, $\mathcal{C}$

$\mathcal{R} \leftarrow$ set of turn-minimizing ranks covering $\mathcal{Q}$ `/* Algorithm 4.6 */`
$\mathcal{V} \leftarrow \{\}$ `/* vertices of m-TSP graph */`
$\mathcal{E}_{\text{req}} \leftarrow \{\}$ `/* required edges */`
for rank r in $\mathcal{R}$ **do**
    $v_0, v_1 \leftarrow$ endpoints of r
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{v_0, v_1\}$
    $\mathcal{E} \leftarrow \mathcal{E} \cup \{(v_0, v_1)\}$
for endpoint pair $(q_i, q_i') \in \mathcal{L}$ **do**
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{q_i, q_i'\}$
$w \leftarrow$ symmetric weight function from $\mathcal{V} \times \mathcal{V}$ to $\mathbb{R}_{\geq 0}$
for vertex pair (v, v') in $\mathcal{V} \times \mathcal{V}$ **do**
    $w(v, v') \leftarrow$ length of shortest path from v to v'
$\mathcal{G} \leftarrow$ complete weighted graph $(\mathcal{V}, \mathcal{V} \times \mathcal{V}, w)$
$\mathcal{C} \leftarrow$ solution to m-TSP on $\mathcal{G}$ with $\mathcal{E}_{\text{req}}, \mathcal{L}$ `/* Algorithm 3.4 */`
return $\mathcal{C}$

In my formulation of the m-TSP (Chapter 3), each task is represented by a single vertex and the edge between two vertices is the time needed to travel between these two tasks. This formulation is too simplistic for coverage. During a single coverage task—travelling from one end of a rank to the other—the robot

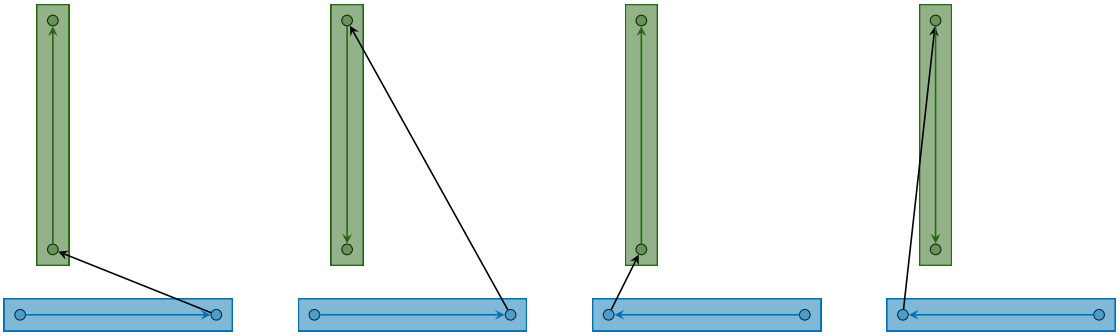


Figure 4.19: As all ranks can be covered in one of two directions, there are four possible ways to cover any pair of ranks sequentially. In general, these four paths all have different lengths.

performing the task moves. As any rank could be travelled in each direction, there are four possible paths between any pair of ranks (Figure 4.19).

Rather than arbitrarily choose one of these paths to use for the distance between the two tasks' vertices, we will use two vertices per rank so that the graph has 4 edges between the vertices of these two ranks. The length of each of these edges is the minimum time needed for the robot to travel between the corresponding ends of the ranks. This minimum time is the time needed to travel along the shortest path, which includes turning times. The shortest path can be computed by Dijkstra's algorithm (Section B.1) or the A* algorithm (Section B.2) on a visibility graph (Appendix A). As shortest paths will be needed between all pairs of rank endpoints, they can all be computed simultaneously using the Floyd-Warshall algorithm (Section B.3) which is slightly more efficient than computing each path individually.

Solving the m-TSP on the graph of rank endpoints with shortest travel times as edge weights does not guarantee a solution where both endpoints of a rank appear consecutively resulting in coverage of the rank (Figure 4.20). Rank partitions often consist of several blocks of many parallel ranks that are approximately the same length. For a block of horizontal ranks, the fastest path would visit all the left ends of the ranks before travelling horizontally to visit the right ends of the ranks.

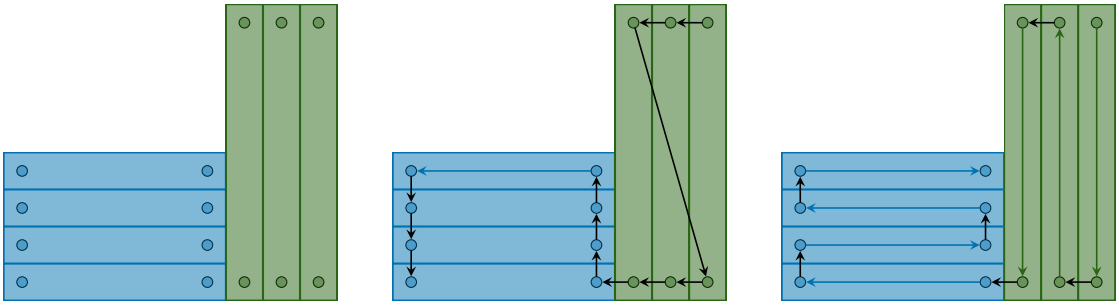


Figure 4.20: The graph used to compute a coverage path by solving the TSP has one vertex for each end of every rank (left). Solving the TSP on this graph usually results in a path which does not properly cover the environment (center). Constraining the path to include all edges between all vertices of the same rank results in the TSP producing the shortest coverage path (right).

Such a path would not cover the region, as it does not actually involve going along most of the individual ranks.

To achieve coverage, we need to constrain the solution of the TSP so that it includes the entire set, $\mathcal{E}_{\text{req}}$, of edges between endpoints of the same rank (Figure 4.21). Ordinary TSP solvers such as the LK heuristic (Section D.3) are unconstrained. However as they are based on local exchanges of edges, the constraints can be enforced by making two small changes:

1. The initial cycle must contain all rank edges; and

2. Any exchange of edges can only break non-rank edges.

These two changes ensure that the initial cycle and every transformed cycle satisfy the rank constraint. Similarly, my m-TSP heuristic (Chapter 3) can be modified to enforce the rank constraint by making four small changes:

1. Both vertices of the same rank are added to an initial partition simultaneously;

2. Transfers and swaps are based on pairs of vertices belonging to the same rank instead of individual vertices;

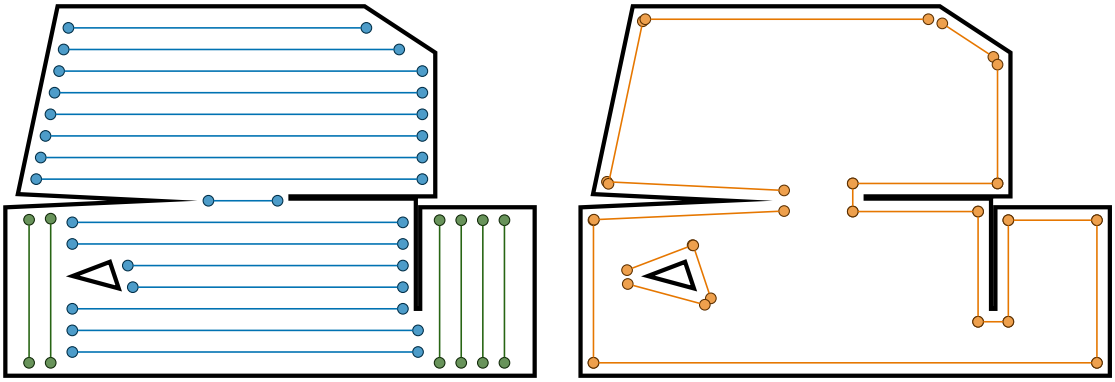


Figure 4.21: Endpoints of interior ranks (left) and perimeter ranks (right) which are vertices in the graph used by the TSP solver to generate the coverage path. The edges shown in these figures are the rank edges and they all must be included in any valid coverage path.

3. The 1-TSP solver satisfies the rank constraint; and

4. Transfers between different robots' cycles include both vertices of a rank instead of a single vertex.

Solving the TSP on the complete weighted graph consisting of all rank endpoints—with the constraint that all edges in $\mathcal{E}_{\text{req}}$ must be included—gives a time-minimizing path on the graph (Figure 4.22 left). For multirobot coverage, we can find paths for each robot by solving the minmax m-TSP (Chapter 3) on the same graph to minimize the time taken by the slowest robot (Figure 4.22 right). As my m-TSP solver can be used with various depot constraints, it can also be used for coverage planning with constraints, $\mathcal{L} = \{(q_1, q_1'), \ldots, (q_m, q_m')\}$, on where the robots start and end their paths.

A minmax m-TSP solver can also be used for planning under energy constraints. Many coverage robots have limited batteries and may need to recharge after only covering part of the coverage region [196]. This battery constraint requires a planned coverage path which visits the charger multiple time throughout the mission so that the parts of the path between visits to the charger are all shorter than the maximum distance the robot can travel on a single charge. This

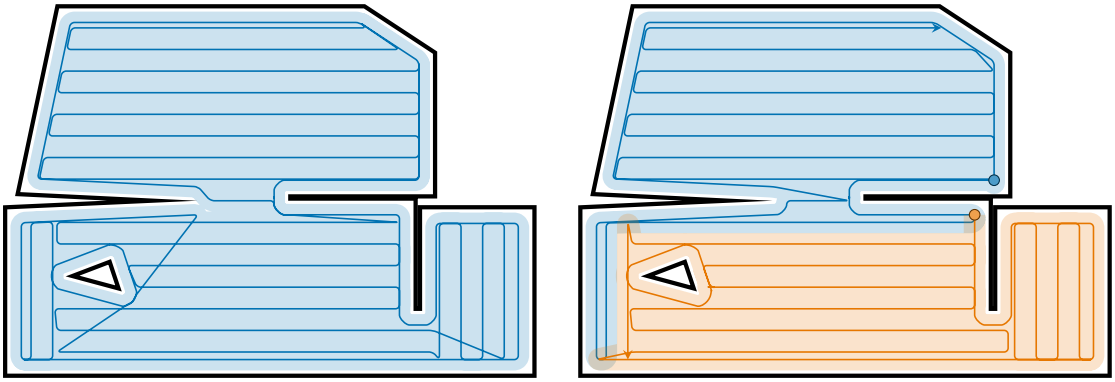


Figure 4.22: Turn-minimizing coverage strategies for one robot with no depot (left) and two robots with one depot each (right).

constraint can be incorporated by iteratively solving the coverage problems for increasing values of m—with each "robot" sharing the same depot—until the length of the longest path is shorter than the maximum coverage distance for a single charge. Then, the resulting coverage paths are each performed consecutively by the same robot, which charges in between, instead of by multiple robots consecutively.

The path created by the TSP solution on the set of ranks results in four types of distinctive behavior for the robot. It can

1. Follow closely around the perimeter of a wall or obstacle;

2. Move in consecutive long straight parallel ranks in the interior of the environment;

3. Move in consecutive long straight parallel ranks in a direction orthogonal to the direction of the first set of ranks; or

4. Travel in an efficient path from the end of one rank to the start of a different rank via critical points near concave corners.

Using these four types of motion the robot is able to minimize the number of turns it makes and hence minimize the coverage time.

4.4 Results

During my PhD, I had the opportunity to travel to Pasadena, California and intern with iRobot, the makers of the Roomba™ robotic vacuum cleaners. As of 2019, robotic vacuum cleaners are one of the most commercially successful kinds of consumer robots, and they perform coverage. The current coverage strategy used by Roombas is a modified boustrophedon strategy which simultaneously explores and covers the environment [70]. While I was at iRobot, they were developing their latest version of the Roomba, the i7+™. One of the most exciting features of this robot is that it makes a map of its environment and saves this map so that the next time it cleans, it can use that map to plan a more efficient coverage strategy. This feature makes the i7+™ a perfect test platform for my coverage strategy.

Implementing my strategy on the real robot would require lots of additional effort to ensure the robot actually performs the strategy as intended despite mapping errors, poor localization, possibilities that the robot might get stuck or run out of battery, and other practical problems (Chapter 6). Before working on this implementation (see Chapter 6), I decided to simply test the strategy on maps made by the robot without actually getting the robot to perform the resulting strategy. During the robot's development, the team of engineers used the robot to experimentally map 25 real indoor test environments using a simultaneous localization and mapping (SLAM) system [19, 53]. These test environments are furnished home and office environments with areas ranging from $10\,\mathrm{m}^2$ to $107\,\mathrm{m}^2$. The combined area of the 25 environments is $1285\,\mathrm{m}^2$. The maps are built by the robot's SLAM system—combining sensor data from the robot's camera, bumper, and wheel odometry—and are stored as occupancy grids where a pixel in the grid is marked as either free, occupied, or unknown [124]. Finally, the occupancy grid is processed into a smoothed polygonal map with straighter walls and fewer small

obstacles that could be easily driven around (see Section 6.4).

For these maps, I computed coverage plans using two strategies: the turn-minimizing strategy with two rank orientations presented in this chapter and a similar strategy with only one rank orientation. The single orientation effectively behaves as a boustrophedon strategy with perimeter following. The two strategies were compared on the basis of total path length, total number of turns, and expected mission time when all 25 environments are covered by teams of 1–5 robots (Table 4.3). Sample paths for a team of two robots in the largest of the 25 environments using both strategies are shown in Figure 4.23. The two approaches have nearly identical path lengths; however, my turn-minimization approach reduced turns by 6.7% resulting in a 3.8% reduction in total mission time. When m robots are used, the total path length and number of turns remain similar but the expected mission time, decreases by a factor of approximately $1/m$ because the robots are covering the environment simultaneously.

The improvements due to turn-minimization can vary significantly depending on the geometry of the environment (Table 4.4). For some environments, particularly ones which are nearly rectangular or have few narrow regions, the turn-minimizing strategy only uses one direction of interior ranks and so nothing is gained by turn-minimization. Other environments with more complex geometries can gain significantly from turn-minimization, potentially reducing mission time by more than 10%.

When computing optimal rank partitions, Algorithm 4.4 ran 50 times with different random initial conditions and I recorded the number of iterations of the inner loop (lines 6–16) and computation time needed to reach the local minimum. The number of iterations scaled linearly with the number of rectangles in the checkerboard partition and the computational runtime scaled proportional to $n_{\text{vertex}}^{1.59}$ where n_{vertex} is the number of vertices in $\partial\mathcal{Q}$ (Figure 4.24) and only required 15 ms of computing time for the largest real environment.

Table 4.3: Cumulative path lengths, numbers of turns, and expected mission times when 25 test environments are covered by teams of 1–5 robots using two different strategies. The 25 environments have a combined coverable area of 1285 m^2 and the robots have a tool width of 10 cm. The expected mission times are for robots which travel at 30 cm/s and take 5 s per turn.

m	Strategy	ℓ (km)	n_{turn}	t (hh:mm:ss)
1	1 orientation	15.260	12414	31:22:06
	2 orientations	15.337	11542	30:13:18
	Improvement	-0.50%	7.02%	3.66%
2	1 orientation	15.326	12260	15:35:14
	2 orientations	15.303	11380	14:58:10
	Improvement	0.15%	7.18%	3.96%
3	1 orientation	15.479	12335	10:28:36
	2 orientations	15.461	11533	10:05:51
	Improvement	0.12%	6.50%	3.62%
4	1 orientation	15.637	12410	7:55:05
	2 orientations	15.564	11586	7:35:49
	Improvement	0.46%	6.64%	4.05%
5	1 orientation	15.757	12485	6:22:53
	2 orientations	15.715	11663	6:08:37
	Improvement	0.27%	6.58%	3.72%

4.5 Conclusions

Many robots are slow at turning so the time needed to follow a path depends on the path's length and the number of turns. I presented a multirobot coverage strategy which explicitly considers the number of turns when planning short coverage paths. Turns are minimized by partitioning the environment into long unit-width rectangles called ranks. Perimeter ranks are parallel to the perimeter of the environment; interior ranks are oriented horizontally or vertically. The interior ranks are constructed using a novel heuristic which minimizes the number of ranks needed to cover the interior of the environment. The overall coverage strategy consists of one path per robot. Coverage paths are generated for m robots by

Table 4.4: Cumulative path lengths, number of turns, and expected mission times for single robot coverage in each of the 25 test environments. The turn-minimizing strategy decreases turns by up to 16.67% and mission time by up to 10.33%.

	1 orientation		2 orientations		Improvement	
Environment	n_{turn}	t	n_{turn}	t	n_{turn}	t
0	1015	2:35:54	962	2:32:07	5.22%	2.43%
1	427	1:16:44	371	1:12:26	13.11%	5.60%
2	661	1:40:24	568	1:33:15	14.07%	7.12%
3	333	0:49:42	300	0:46:59	9.91%	5.47%
4	542	1:28:07	492	1:23:50	9.23%	4.86%
5	707	1:44:39	602	1:35:50	14.85%	8.42%
6	409	0:58:25	409	0:58:25	0 %	0 %
7	388	1:05:26	363	1:03:18	6.44%	3.26%
8	733	1:41:08	653	1:34:37	10.91%	6.44%
9	359	0:55:36	356	0:55:29	0.84%	0.21%
10	798	2:11:09	771	2:09:22	3.38%	1.36%
11	780	2:04:36	770	2:03:55	1.28%	0.55%
12	221	0:37:44	221	0:37:44	0 %	0 %
13	125	0:17:24	125	0:17:24	0 %	0 %
14	510	1:10:01	425	1:02:47	16.67%	10.33%
15	549	1:16:32	510	1:13:32	7.10%	3.92%
16	664	1:33:11	646	1:32:17	2.71%	0.97%
17	259	0:43:34	256	0:43:22	1.16%	0.46%
18	387	0:57:20	360	0:55:21	6.98%	3.46%
19	382	0:56:32	382	0:56:32	0 %	0 %
20	418	1:01:15	395	0:59:22	5.50%	3.07%
21	378	0:57:28	361	0:56:21	4.50%	1.94%
22	386	0:55:28	338	0:51:32	12.44%	7.09%
23	572	1:25:08	526	1:21:14	8.04%	4.58%
24	411	0:58:39	380	0:56:17	7.54%	4.04%
Total	12414	31:22:06	11542	30:13:18	7.02%	3.66%

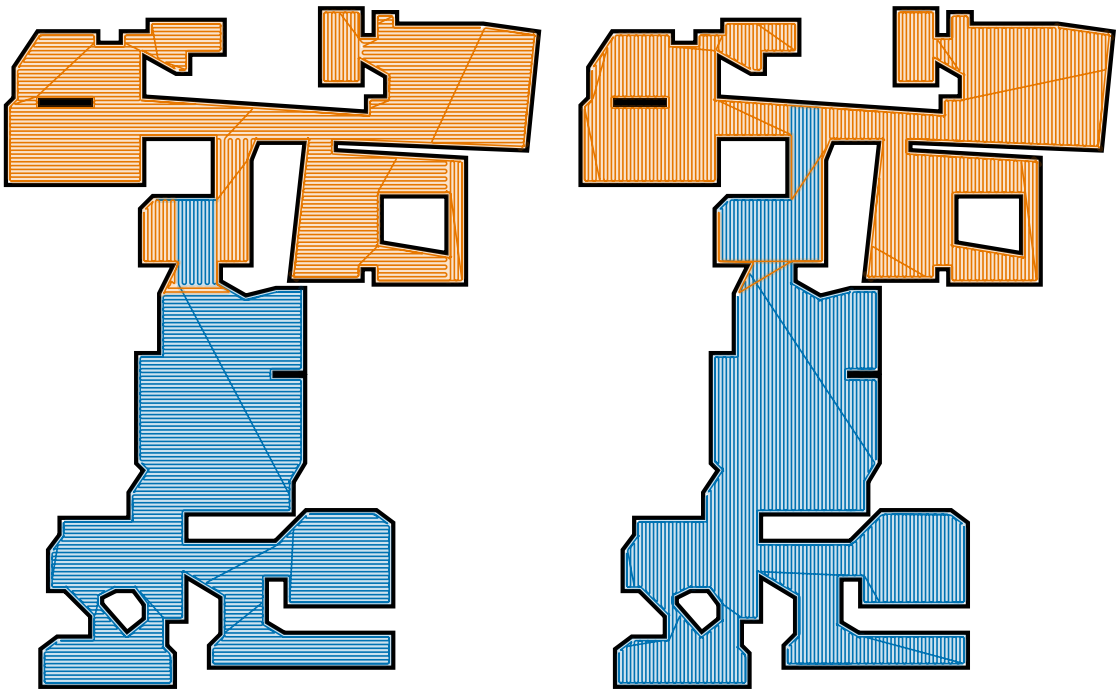


Figure 4.23: Comparison of robot coverage plans for a team of two robots in a 107 m^2 test environment using one orientation based on the environment's bounding box (left) and two orientations obtained by Algorithm 4.6 (right). For the 1 orientation strategy, the robots have expected coverage times of 1:21:39 (blue) and 1:21:36 (orange). The 2 orientation strategy's mission time is 13.0% faster with expected coverage times of 1:11:01 (blue) and 1:10:58 (orange).

solving a constrained version of the minmax m-TSP presented in Chapter 3.

I compared this strategy with one which does not minimize the number of turns on 25 real indoor environments with a combined area of 1285 m^2 mapped by the iRobot Roomba i7+™. For coverage with 1–5 robots, this strategy reduced turns by 6.7% and the coverage time by 3.8% on average. For real robots, minimizing turns also has the added benefit of reducing the likelihood of the robot getting stuck or having localization errors, both of which are more common when turning.

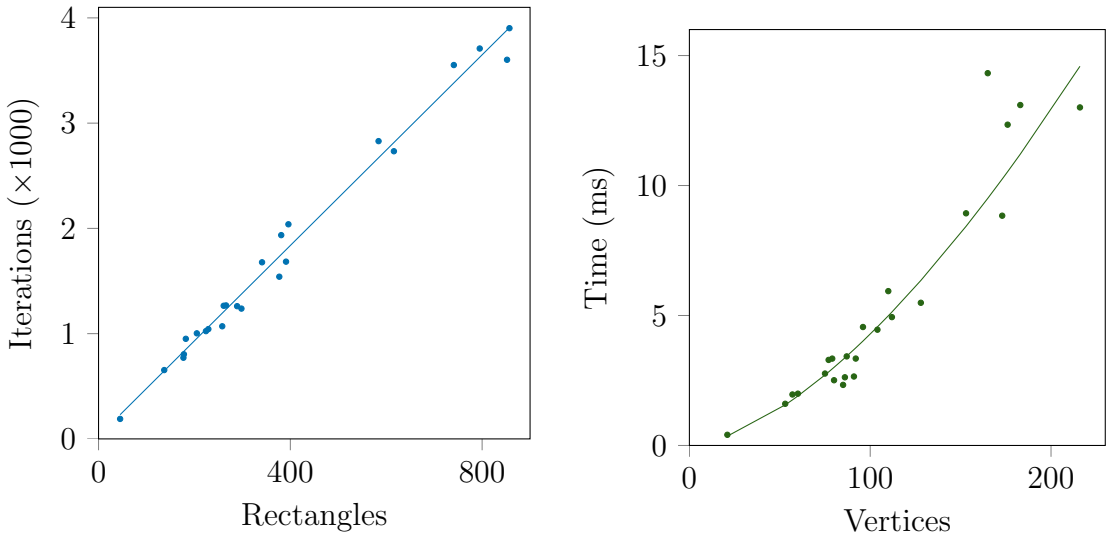


Figure 4.24: Regression results showing linear relationship ($\widehat{y} = 4.53x + 26.24$) between number of iterations of the inner loop of Algorithm 4.4 and the number of rectangles in a checkerboard partition (left); and relationship of $\widehat{y} = 0.002826x^{1.59}$ between the computational runtime of Algorithm 4.4 and the number of vertices in a polygon. Computations were performed in C++ on a standard consumer laptop running Ubuntu.

Chapter 5

Coordinated multirobot search

Communication is essential for the successful completion of most tasks performed by teams of mobile robots. In real environments, robots often communicate over inexpensive ad-hoc networks which have limited connectivity that is affected by distance and line of sight [141]. The robots may lose connectivity as they move throughout their environment. There are several possible solutions to this problem (Figure 5.1).

- **Constant connectivity** [46, 151, 163, 179, 205] is when the robots' motion is restricted to maintain connectivity. Although this constraint enables constant communicate, it forces the team of robots to remain near each other making them less effective at other tasks that benefit from spreading out.

- **Periodic connectivity** [80, 94, 156] is when the team is allowed to separate temporarily if it has a plan of where they will meet back up. Regular or preplanned meetings give robots some flexibility to separate, but are inconvenient when tasks take unpredictable lengths of time, as some robots will be forced to wait for others. Even worse, if one robot gets stuck or cannot reach the meeting point, the team will never get reconnected.

- **Intermittent connectivity** [186] is when the team can separate without a plan for when they will reconnect. This approach is the most robust to unexpected circumstances but requires the robots to search for each other when they want to communicate.

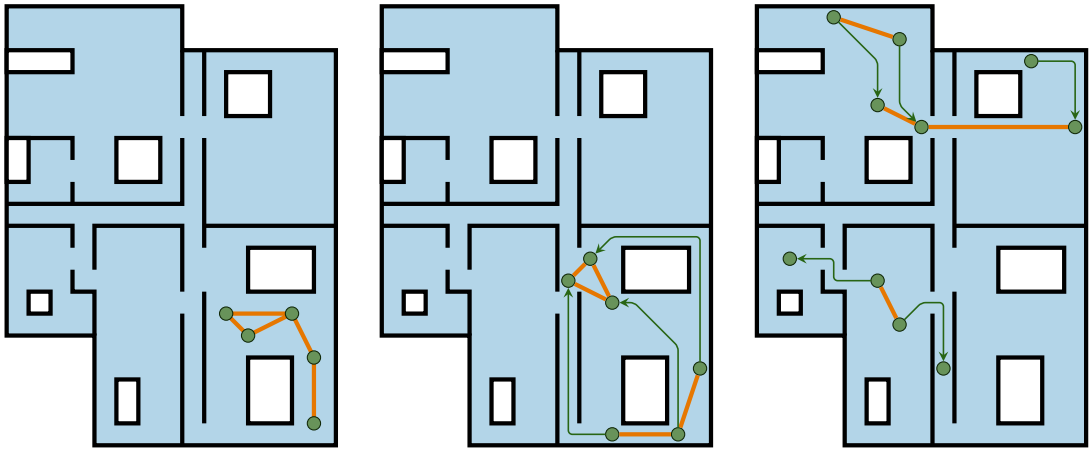


Figure 5.1: Three communication strategies for teams of robots are constant connectivity (left) where the team never separates, periodic connectivity (center) where the team can separate with a planned meeting, and intermittent connectivity (right) where the team can separate without a planned meeting.

Although constant and periodic connectivity are useful in some applications, intermittent connectivity is the most flexible. The best approach for real robots depends on a range of factors, including the size of the environment and how predictably the robots behave. For predictable robots in a large environment, a conservative strategy with preplanned meetings may be best. However, for real robots which rarely follow plans exactly and often work in unpredictable environments, intermittent connectivity is likely the best approach.

When robots communicate intermittently, they do not have a prearranged meeting and therefore have to find each other without sharing any common information. This problem can be described in one of three ways depending on the target robot's behavior. Its behavior can be a) cooperative, b) adversarial, or c) neutral. These problems are commonly known as rendezvous, pursuit-evasion, and search. In practice, a searcher often does not know whether its target is cooperative, adversarial, or neutral and should use a strategy which can be effective regardless of its target's objectives.

In this chapter, I present a flexible communication strategy that can be used when completing a cooperative task. This strategy allows for varying degrees

of communication so that robots can benefit from cooperation without wasting excessive energy to communicate. As my method does not require constant or periodic communication, the team will in general be disconnected. If a robot wants to communicate with a disconnected robot, it searches for that target robot using its belief of the target's position. This belief is estimated using a probabilistic model of the target's motion and the communication structure of the environment is explicitly considered when planning search paths. Reconnection is successful if two robots are within communication range, which depends on the environment's and robots' properties. The robots do not need to be in the exact same location to successfully reconnect.

I originally described a similar communication strategy in my paper "Re-establishing communication in teams of mobile robots" [186]. Although this chapter is loosely based on that paper, it has been updated, with three major changes:

1. I have replaced the semi-Markov model with a more general hidden Markov model (HMM). Similar to semi-Markov models, an HMM can describe the variable speed of a robot and possibility that it might get stuck or stop moving. Additionally, the HMM is able to model the momentum of the robot and historic or simulated paths can be used to compute realistic transition probabilities.

2. I have chosen to describe the HMM using matrix instead of tensor notation. Although tensors are an elegant way to describe semi-Markov models, they are less suited for the HMM used in this chapter as the set of possible states cannot be decomposed using a Cartesian product (i.e. not all states have velocity or direction information attached). Therefore, I have opted for matrix notation which is more accessible to most readers.

3. I have added a sampling-based method for planning search paths. This method is generally superior to the branch-and-bound method I used in my

previous paper as it has lower computational complexity and is not constrained to paths on a grid.

The results of these changes is a more accurate model of the target robot's behavior, which results in improved beliefs and more effective search paths. An abridged version of this chapter is currently under review for publication as "Sampling based search for a semi-cooperative target" [189].

5.1 Related work

Search theory dates back to the 1940s motivated by the US Navy's antisubmarine missions during World War II [110]. Although most early efforts [28, 79, 177] were focused on searching for stationary targets, some authors also considered targets which moved randomly between finite sets of cells, indifferent to the searcher [31, 55, 195]. This problem, known as one-sided search, has more recently been studied by the robotics community [40, 159]. The typical approach to search involves a belief of a target's location, updated using a motion model, and then planning a path which maximizes the probability of finding the target over some horizon. Different authors have used different techniques for both belief estimation and path planning.

A *belief* is a probabilistic description of where a target robot might be based on information available to a searcher. Beliefs change over time as the searcher expects its target to move. Three main methods of describing and maintaining beliefs are:

1. **Markov models** represent the random motion of a target on a graph—corresponding to a discretization of the environment—using transition probabilities which only depend on the robot's current state and stores the belief as a probability vector over the graph's vertices [21, 81, 115]. Variants of Markov models, such as second-order Markov models [203], semi-Markov

models [186], and hidden Markov models [23] provide more realistic descriptions of the target's motion within the same framework.

2. **Particle filtering algorithms** use a finite set of particles which each move in continuous space according to the target's dynamics, using different values for each particle's control inputs [35, 69, 157]. Each particle represents one possible behavior of the target, and with an appropriate distribution of control inputs, the entire set of particles approximates the distribution of target locations. A probability hypothesis density filter extends this method by using Gaussian distributions instead of point-like particles [45, 180].

3. **Historical data** can also be used to build a model, if enough data is available or can be simulated [162]. A model based on historic paths can take into account previous locations along a path, and so is more realistic than a memoryless Markov model.

Any of these techniques can provide a searcher with a belief which it can use to prioritize where it searches for its target. In this chapter, I will use a hidden Markov model which combines a second-order Markov model with a semi-Markov model and can model a target robot's momentum and the variability of its speed. Additionally, if historic or simulated date are available, they can be used to determine the transition probabilities of the HMM.

Using its belief, the searcher can plan its search strategy. This strategy is often a path which minimizes the expected time to find the target [135, 167] or which maximizes:

(a) The probability of finding the target over a finite horizon [69, 115, 162, 186, 203];

(b) The probability of finding the target per unit time [157];

(c) A discounted reward which values finding the target quickly [18, 81, 173]; or

(d) A fairness-based reward which values regularly observing multiple different targets [21].

As the problem of finding the optimal path is NP-hard, methods such as branch-and-bound [81, 115, 203], mixed integer linear programming [162], inverse reinforcement learning [173], multi-level optimization [167], and depth-first search [69] are used to plan near-optimal paths quickly. A fast alternative to planning an entire path is to simply move to the single location which has the highest probability of finding the target [23, 71]. My search algorithm uses a discounted reward function, which behaves well with the sampling-based planner that I use.

As search involves multiple robots, it is natural to also consider search algorithms for multiple searchers. If two searchers have different beliefs, they can combine their beliefs by taking the element-wise minimum value of two probabilistic belief vectors and renormalizing [83]. As combinatorial path planning algorithms scale exponentially with the number of robots if all of their paths are planned simultaneously, individual robots' paths are often planned sequentially [81, 157, 203]. A variant of Lloyd's algorithm, weighted based on the target belief can also be used to spread out searchers looking for a common target [45].

My approach to re-establishing connection between robots uses a sampling-based planner. Sampling-based path planners have been popular in robotics since the development of probabilistic roadmaps (PRM) [101] and rapidly-exploring random trees (RRT) [116] in the late 1990s. Both techniques involve randomly sampling the configuration space to incrementally construct a graph that fills the space and whose edges represent feasible paths between nearby configurations. The major difference is that RRT constructs a tree making it fast for planning individual paths, whereas PRM constructs a graph with cycles making it more efficient for repeatedly planning paths. An improved version of RRT, called RRT*, rewires the tree as more vertices are added, resulting in an asymptotically optimal planner [96]. These planners can often be made more effective using non-uniform sampling [117]

based on historic data of the robot moving in a desirable way [87, 113] . Although originally used to minimize distance, RRT has also been used to:

- Minimize localization error [86];
- Minimize mechanical work when traversing uneven terrain [90];
- Minimize probability of capture in a pursuit-evasion game [97];
- Minimize or distance from a moving target with known location [174]; and
- Maximize information gain along a path [82].

My sampling-based planner finds a path which maximizes a discounted reward function based on finding the target robot as quickly as possible.

5.2 Communication in crowded environments

Robots often communicate directly with other robots forming ad-hoc wireless networks. The strength of this network's signals determines the probability that two robots will communicate successfully [7]. The two main factors influencing the wireless signal strength between two robots trying to communicate are distance and line of sight [140] (Section 2.3). These two factors can result in a variety of different communication models (Figure 2.6) ranging from full communication everywhere to communication that is blocked by any obstacle and decreases quickly with distance.

5.2.1 Known robot locations

Suppose there are two robots located at positions q_0, q_1 in some environment $\mathcal{Q}$. The probability that these two robots can communicate is $C(q_0, q_1) \in [0, 1]$. This probability depends on many factors including the properties of the environment,

the type of wireless signals, and whether they can communicate through intermediate devices. Assuming that obstacles completely block communication, the communication probability is

$$C(q_0, q_1) = \begin{cases} 0 & \text{if } q_0 \text{ is not visible from } q_1 \\ C_{\text{vis}}(\|q_0, q_1\|) & \text{if } q_0 \text{ is visible from } q_1 \end{cases}$$

where $C_{\text{vis}} : \mathbb{R}_{\geq 0} \to [0, 1]$ is a monotonically decreasing function which describes the effect of distance on communication probability. Signal strength tends to decrease with distance according to an inverse square law [141]. I will assume that when the signal strength is above a certain threshold, communication is guaranteed and below this threshold the communication probability is proportional to the signal strength. Under this assumption, the effect of distance on communication probability is

$$C_{\text{vis}}(d) = \begin{cases} 1 & \text{if } d \leq d_{\text{threshold}} \\ \frac{1}{1+k_{\text{decay}}(d-d_{\text{threshold}})^2} & \text{if } d > d_{\text{threshold}} \end{cases}$$

where $k_{\text{decay}} \in \mathbb{R}_{>0}$ and $d_{\text{threshold}} \in \mathbb{R}_{\geq 0}$ are parameters that exactly how the probability decreases with distance. These parameters can be estimated experimentally [141]. Alternatively, $C(\cdot, \cdot)$ can be modelled through some other method such as using Gaussian processes to fit experimental signal strength data [119]. For the remainder of this chapter, I will assume that $C(\cdot, \cdot)$ is known—or at least can be estimated—but its exact form is not particularly important.

5.2.2 Uncertain target location

When planning a search path, the searching robot needs to answer the question "If I go over there, what is the probability that I will be able to communicate with my target?" Suppose that the searcher has a belief of the target's location which

can be described by the probability measure $\mu : \Sigma \to [0, 1]$ where Σ is a σ-algebra on $\mathcal{Q}$. Based on this belief, the probability that the two robots could communicate if the searcher was at q_{sea} is

$$\mathbb{P}(\text{Communication is possible at } q_{\text{sea}}) = \int_{\mathcal{Q}} C(q_{\text{sea}}, q_{\text{tar}}) \mathrm{d}\mu(q_{\text{tar}}). \tag{5.1}$$

Although this expression quite elegantly expresses the communication probability, it is useless for robotic purposes as it is defined over continuous space. Instead it will be approximated.

How (5.1) gets approximated, depends on what model the searcher uses for the target belief, and what communication model it has. In this chapter, I will use an HMM for maintaining a belief and the resulting belief, $\boldsymbol{b}_{\text{tar}} : \mathcal{Y} \to [0, 1]$, is a probability vector (Section 5.3). The belief is based on a discretization of the environment into a finite set of cells, $\mathcal{Y}$, and the i^{th} element of $\boldsymbol{b}_{\text{tar}}$ is the belief that the target is in cell y_i. We can also express q_{sea} as a probability vector $\boldsymbol{b}_{\text{sea}} : \mathcal{Y} \to [0, 1]$ where all of its elements are non-zero except for the element corresponding to the cell that contains q_{sea}. Using $\mathcal{Y}$ as a discrete approximation of $\mathcal{Q}$, the integral in (5.1) becomes a sum, which can be expressed in matrix notation as

$$\mathbb{P}(\text{communication}) = \boldsymbol{b}_{\text{sea}}^{\top} \boldsymbol{C} \boldsymbol{b}_{\text{tar}} \tag{5.2}$$

where $\boldsymbol{C}$ is the *communication matrix*. The $(i, j)^{\text{th}}$ element of $\boldsymbol{C}$ is the probability that a robot at the centroid of cell y_i can communicate with a robot at the centroid cell y_j.

5.2.3 Environment decomposition

The probability of communication between robots in two cells varies depending on their exact locations in the cells, but the discrete approximation in (5.2) only uses the probabilities at the centroids of the cells. Its accuracy therefore depends on

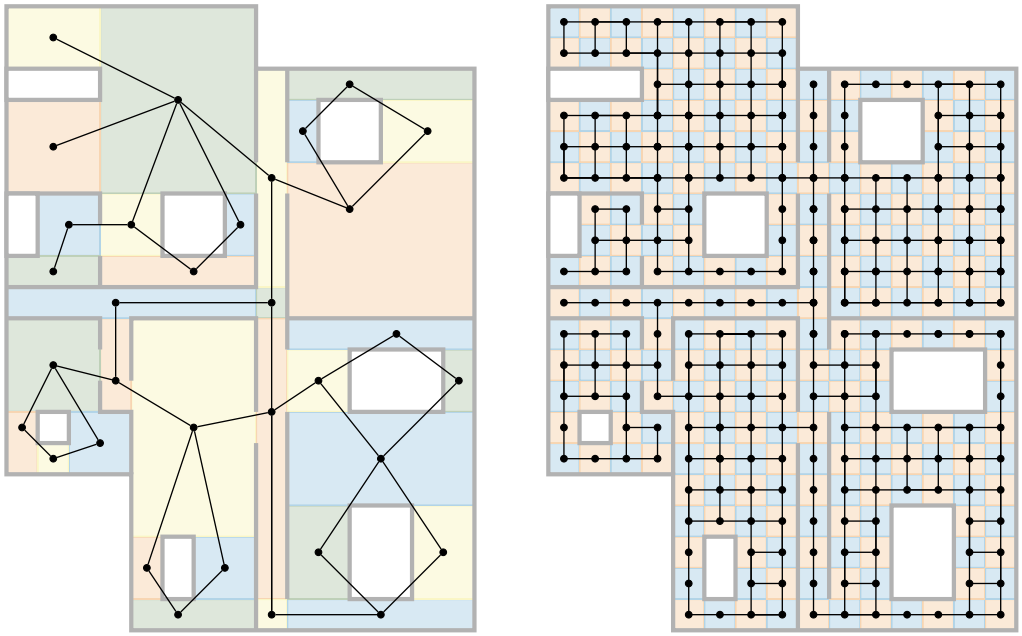


Figure 5.2: Exact (left) and approximate (right) cellular decompositions can both be used to convert a continuous environment into a graph. This graph is a simplified model that is used to define the belief vector which is used in search.

how much the communication probability varies within a cell. We therefore need to choose a decomposition $\mathcal{Y}$ which has little variance in communication probability within each cell.

As in coverage planning (Subsection 2.4.1), the two main kinds of decompositions are exact and approximate [37] (Figure 5.2). As probability of communication changes with distance, (5.2) is a better approximation if $\mathcal{Y}$ has small compact cells than for large or oblong cells. Exact decompositions have large, irregular shaped cells; whereas approximate decompositions have the desired small compact cells. Therefore we will use an approximate decomposition based on a polygonal lattice—either triangular, square, or hexagonal (Figure 5.3). A lattice with smaller cells has a higher resolutions and will approximate the belief and communication probability better at the expense of increased computational effort. The best lattice which balances of speed and resolution depends on the exact behavior of the robots and how much computing power is available.

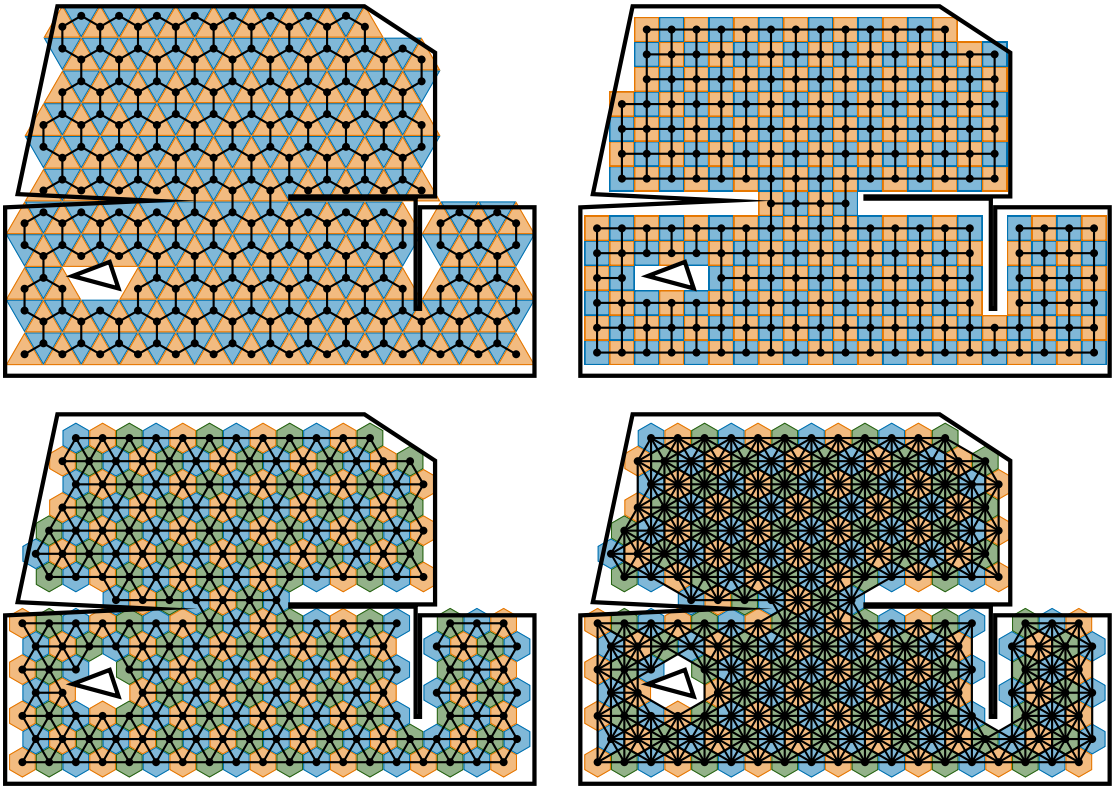


Figure 5.3: Examples of four regular polygonal lattices. All polygonal lattices are based on triangular, square, or hexagonal cells; however the number of neighbors each cell has can vary depending on whether or cells are only connected across shared edges or also along shared corners.

5.3 Tracking an unseen target

A robot searches for a disconnected teammate by following the path which maximizes the probability of finding it. This probability is computed from the belief of the target robot's position which is based on a model of its motion. The belief is a probability distribution over a discretization of the environment based on a polygonal lattice (Figure 5.4). Computationally, this belief is stored as a vector $\boldsymbol{b} \in \mathbb{R}^{|\mathcal{Y}|}$ where each element is

$$b_i = \mathbb{P}\left(q_{\text{tar}} \in y_i \mid \text{information known by searcher}\right) \tag{5.3}$$

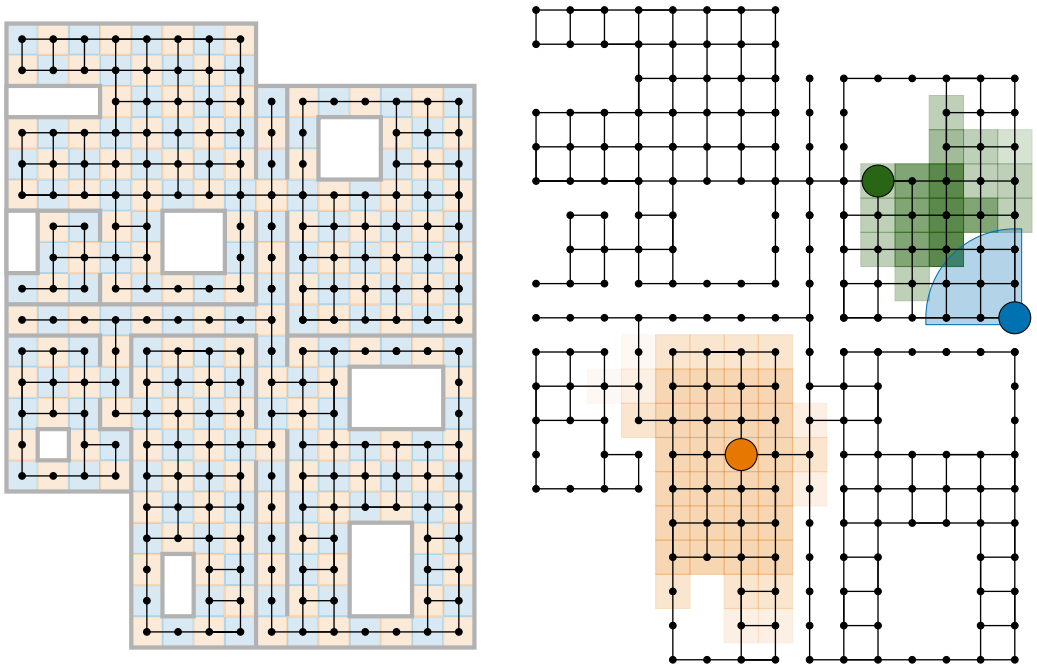


Figure 5.4: A searcher (blue) defines its beliefs using a discretization of the environment (left). The belief of each target's location is a probability distribution over this discretization.

If the target and searcher can communicate, then this vector is simply

$$b_i = \begin{cases} 1 & \text{if } q_{\text{tar}} \in y_i \\ 0 & \text{otherwise} \end{cases}$$

Once the target and searcher get disconnected, the belief must be updated and without full information, $\boldsymbol{b}$ will have many non-zero entries.

5.3.1 Basic Markov motion model

A Markov model is a simple way to update a belief vector. It is based on the somewhat unrealistic assumption that the target's behavior only depends on its current location. In any time step, the robot can move directly to a neighbor cell and the probability that it moves to this cell is known. Under this assumption,

the probability that a robot will be in cell y_j at time τ is

$$\begin{aligned}\mathbb{P}(q[\tau] \in y_j) &= \sum_{y_i \in \mathcal{Y}} \mathbb{P}(q[\tau] \in y_j \mid q[\tau-1] \in y_i)\mathbb{P}(q[\tau-1] \in y_i) \\ &= \sum_{y_i \in \mathcal{Y}} a_j^i \mathbb{P}(q[\tau-1] \in y_i) \end{aligned} \tag{5.4}$$

where $a_j^i = \mathbb{P}(q[\tau] \in y_j \mid q[\tau-1] \in y_i)$ is the probability that a robot in cell y_i will move to cell y_j in the next time step. The assumption of a Markov model is that the behavior of the target is completely determined by a_j^i which does not depend on time or any additional information. Depending on the shape of the lattice, and whether corner-neighbors or only edge-neighbors are allowed, a cell can have between 3–12 neighbors (Figure 5.5). Using (5.3) and (5.4), we can update the belief by

$$b_j[\tau] = \sum_{y_i \in \mathcal{Y}} a_j^i b_i[\tau-1] = \boldsymbol{a}_j \boldsymbol{b}[\tau-1]$$

where $\boldsymbol{a}_j \in (\mathbb{R}^n)^*$ is the covector obtained by combining all a_j^i into a single row vector. By aggregating this expression for all b_j, the overall update rule is simply

$$\boldsymbol{b}[\tau] = \boldsymbol{A}\boldsymbol{b}[\tau-1] \tag{5.5}$$

where $\boldsymbol{A} : \mathbb{R}^n \to \mathbb{R}^n$ is the *Markov transition matrix* whose rows are the $\boldsymbol{a}_j$'s. $\boldsymbol{A}$ is the adjacency matrix of a weighted directed graph $\mathcal{G}_{\text{lat}} = (\mathcal{Y}, \mathcal{E}, w)$ which contains the edge (y_i, y_j) if the two cells are neighbors in the lattice. As the probability, a_j^i, is only non-zero if y_i and y_j are neighboring cells, $\boldsymbol{A}$ is sparse.

5.3.2 What about momentum?

One of the most unrealistic features of the basic Markov model, (5.5), is that it assumes a robot is equally likely to move back to the cell it just came from as it is to move on to a third cell. Real robots, like all mechanical systems, have second

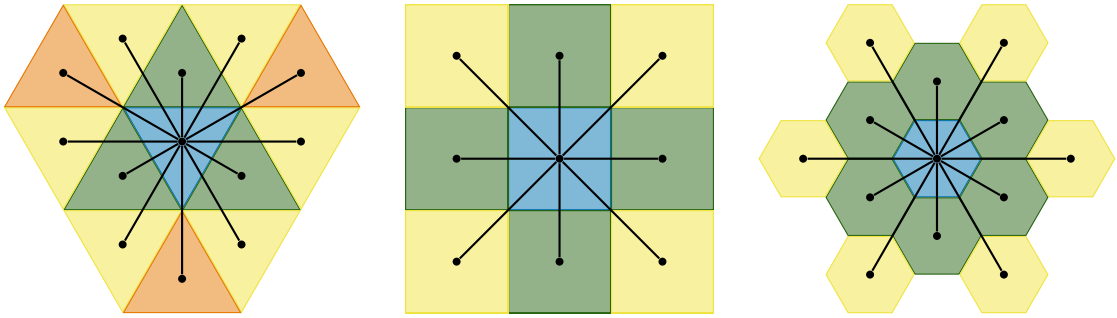


Figure 5.5: The number of possible neighbors in a lattice depends on whether corner-neighbors or only edge-neighbors are allowed. Triangular cells can have 3, 9, or 12 neighbors (left); square cells can have 4 or 8 neighbors (center); hexagonal cells can have 6 or 12 neighbors (right). For triangular latices, some corner neighbors are farther away than others so both 9- and 12-connected lattices are possible.

order dynamics so its momentum affects where it will go next. Indeed, the vast majority of robotic tasks—exploration, coverage, search, delivery—all involve the robot moving in a straight line much more often than it goes back-and-forth over the same location.

Ordinary Markov models simply cannot account for this tendency of robots to move in straight lines. If we relax the requirement that the transition probabilities only depend on the previous location, we could use the previous two locations to update the belief

$$\mathbb{P}(q[\tau] \in y_j) = \sum_{y_i \in \mathcal{Y}} \sum_{y_k \in \mathcal{Y}} a_j^{i,k} \mathbb{P}(q[\tau - 1] \in y_i) \mathbb{P}(q[\tau - 2] \in y_k) \tag{5.6}$$

where the new second-order transition probabilities are

$$a_j^{i,k} = \mathbb{P}(q[\tau] \in y_j \mid q[\tau - 1] \in y_i \text{ and } q[\tau - 2] \in y_k)$$

Although (5.6) could be used to update beliefs using the current and previous location, it cannot be converted into a matrix equation (it could be converted into a $(1, 2)$-tensor, however) which makes it somewhat more complicated, conceptually.

Rather than express the second-order Markov model update using a tensor product, I will instead express it as a hidden Markov model (HMM). An HMM augments the cells, $\mathcal{Y}$, of the basic Markov model with an additional, larger, set of states, $\mathcal{X}$. The states of $\mathcal{X}$ are "hidden" in the sense that they can represent some internal states of the target robot—its current plan, its heading, or its velocity—and not just its position which could be measured from a snapshot taken by an observer. The HMM is formulated quite similarly to a Markov model. The transitions between its states are Markov and can be represented by matrix multiplication

$$\boldsymbol{w}[\tau] = \boldsymbol{A}\boldsymbol{w}[\tau - 1] \tag{5.7}$$

where $\boldsymbol{w} \in \mathbb{R}^{|\mathcal{X}|}$ is a vector describing the searcher's belief about the target's state. Its elements are

$$w_i[\tau] = \mathbb{P}(x[\tau] = x_i \mid \text{information known by searcher})$$

where $x[\tau] \in \mathcal{X}$ is the target's true state and $x_i \in \mathcal{X}$ is one of the possible states. Similar to (5.5), the state at time τ only depends on the state at τ. The actual observable location, which corresponds to the belief vector, is related to the full state by a projection operation

$$\boldsymbol{b}[\tau] = \boldsymbol{P}\boldsymbol{w}[\tau] \tag{5.8}$$

where $\boldsymbol{P} : \mathbb{R}^{|\mathcal{X}|} \to \mathbb{R}^{|\mathcal{Y}|}$ is a projection matrix that converts high-dimensional state beliefs to lower-dimensional location beliefs.

We can convert a second-order Markov model into an HMM by using states that contain information about the current and previous state. There are two ways to encode this information:

1. Use pairs of cells, (y_i, y_k), as states; or

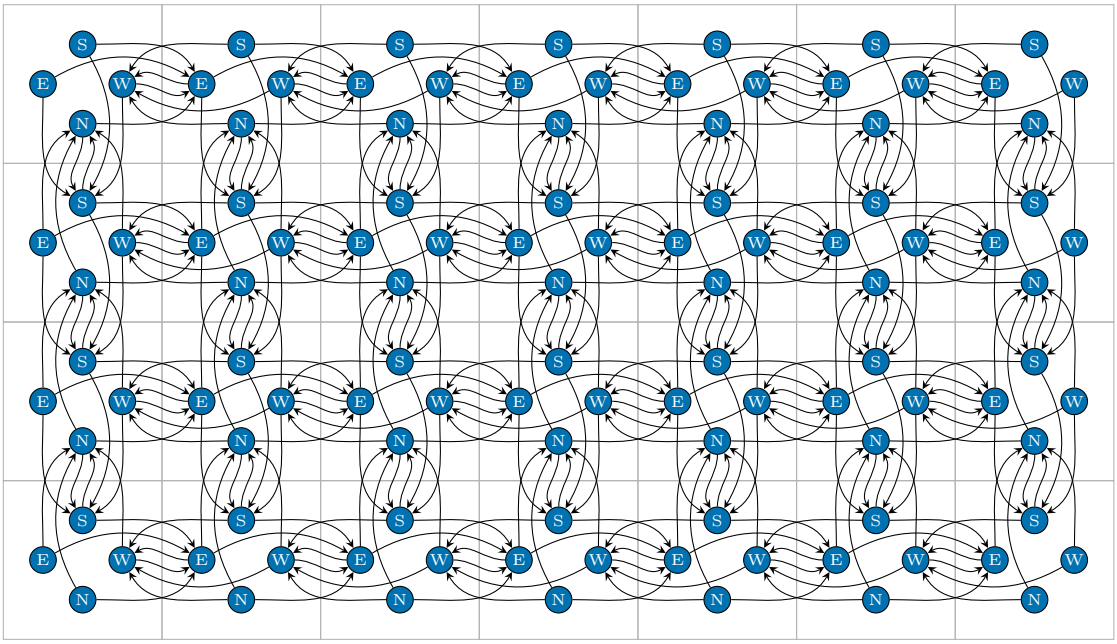


Figure 5.6: State transition graph of a second-order Markov model on a 4-connected square grid. Edges are between states of neighboring grid cells with a compatible direction of the end state.

2. Use a cell and orientation pair, (y_i, θ) as states. The set of possible orientations, Θ, only contains the directions of edges of $\mathcal{G}_{\text{lat}}$.

These two approaches are equivalent; however, I think the second approach is slightly more intuitive. If we were to use pairs of cells, we'd have to check which pairs are infeasible as it only makes sense to include (y_i, y_k) if y_i and y_k are neighbors. Using a single cell and orientation, we don't need to check for feasibility, as the robot can have any orientation in any cell.

Similar to the Markov model, the state transition matrix $\boldsymbol{A}$ in (5.7) is the adjacency matrix of a graph with $\mathcal{X}$ as its vertices (Figure 5.6). This graph contains the directed edge from (y_i, θ_i) to (y_j, θ_j) if and only if y_j is a neighbor of y_i and θ_j is the direction from y_i to y_j. Note that this definition does not depend on θ_i which means that every edge from states of y_i have out-edges to the same state of y_j.

The projection matrix, $\boldsymbol{P}$, is used to convert a state-belief, $\boldsymbol{w}$, to a cell-belief,

$\boldsymbol{b}$. It is a sparse matrix whose elements are

$$p_j^i = \begin{cases} 1 & \text{if state } x_i \text{ is in cell } y_j \\ 0 & \text{otherwise.} \end{cases} \tag{5.9}$$

This definition results in $\boldsymbol{P}$ being a left-stochastic matrix with a single 1 in each column.

5.3.3 Variable speed target

The models I've described so far also make the unrealistic assumption that the robot is always able to move by one cell per time step. This assumption is wrong. On a 9- or 12-connected triangular grid, 8-connected square grid, or 12-connected hexagonal grid, some of the edges have different lengths, so it should take longer for the robot to travel between those neighboring cells. When using a second-order graph, we should also expect it to take longer for a robot to travel between two cells if it needs to turn around to get to the destination cell than if it is already facing the correct direction. Furthermore, most robots do not reliably move at a constant speed. All of these facts mean that a realistic model of a robot's motion should allow for variable speed transitions.

Semi-Markov models are an extension of the basic Markov model to the scenario where the state transitions do not necessarily happen at regular intervals. In a semi-Markov model, there is a time distribution associated with each state and the process remains in that state for a duration determined by this distribution before transitioning to a new state. When the process transitions, the next state only depends on the previous state. A hidden semi-Markov model [204] is a variant of an HMM where the semi-Markov process is hidden and related to an observable state by a projection matrix like in (5.8).

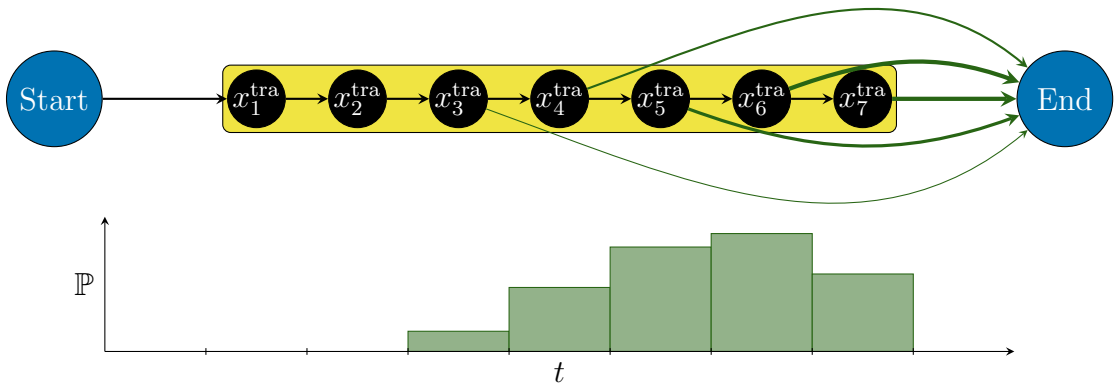


Figure 5.7: A semi-Markov model with a discrete time distribution can be expressed as an HMM by adding a chain of transit states. The probability of the transition from each transit state to the end state is determined by the time distribution of the semi-Markov model.

In its most general form a semi-Markov model's time distributions are continuous. For computational purposes, it is much more convenient to use a discrete approximation of the time distribution. In this case, we can add a chain of *transit states* between two states of the original Markov model (Figure 5.7). The length of this chain is equal to the number of time steps in the time distribution. Each transit state can either transition to the next transit state in the chain or to the end state of the original Markov model. The probability of transitioning to the end state is

$$\mathbb{P}(x[\tau] = \text{end} \mid x[\tau - 1] = x_i^{\text{tra}}) = \frac{\mathbb{P}(\text{transition takes } (i+1)\Delta t)}{\mathbb{P}(\text{transition takes longer than } (i+1)\Delta t)}$$

If the transition can happen in a single time step, we add an edge directly from the start vertex to the end vertex. An HMM with transit vertices with these probabilities has identical behaviour to a semi-Markov model with the discrete time distribution.

Chains of transit states can also be added along the edges of any existing HMM. In particular, we can add them between the HMM with direction states

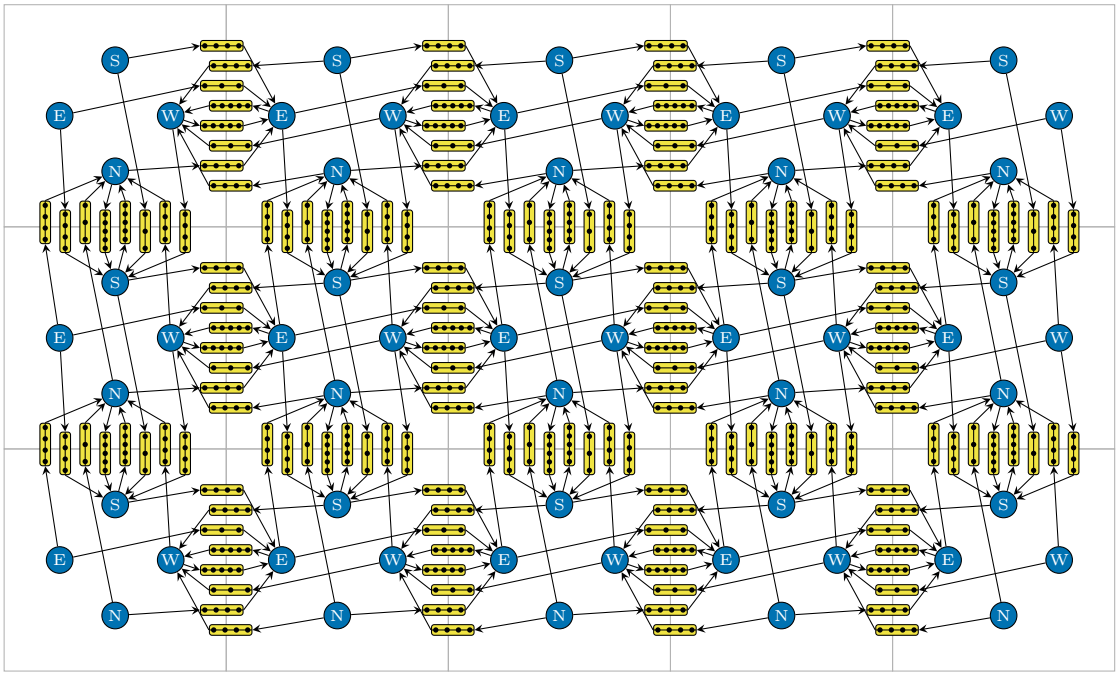


Figure 5.8: The overall search graph has direction states in each cell and chains of transit states connecting direction states in neighboring cells.

from Subsection 5.3.2. The resulting HMM (Figure 5.8) is equivalent to a second-order semi-Markov model. The transit states model the distribution of times that it actually takes to move between two different states of the second-order Markov model, including the time needed to turn. The time distributions therefore depend on the distance and angle between the start and end states as well as the target robot's linear and turning speeds.

When we add transit states to the HMM, the projection matrix, $\boldsymbol{P}$, will also need to contain one additional column per transit state. Suppose x^{tra} is a transit state between $x_i = (y_i, \theta_i)$ and $x_j = (y_j, \theta_j)$. As the transit state represents motion in between cells y_i and y_j, its physical location could be in either of these cells. Therefore the column of $\boldsymbol{P}$ corresponding to x^{tra} should have non-zero entries in the i^{th} and j^{th} positions and be zero everywhere else. Transit vertices at the beginning of the chain should have a larger i^{th} value; whereas transit vertices at the end of the chain should have a larger j^{th}. In this way, if the state moves along

the chain, the cell will transition monotonically from y_i to y_j.

5.3.4 A model built from historic data

A robot's target usually behaves in a somewhat repetitive and therefore predictable way. For example, the target may have certain locations that it visits regularly—such as a charging station—and other locations that it rarely visits. The searcher can use this information about the target's behavior to make a more accurate HMM. Ideally, the searcher will have access to historic data of the target's actual paths. However, if it only has a description of what the target is doing or how it behaves, it can simulate the target's behavior and use simulated path data in lieu of historic data.

Target paths can be used to determine the transition probabilities between adjacent direction states by simply counting the number of times each transition happens in the path data (Algorithm 5.1). As historic paths are continuous curves, they should first be discretized into a polygonal path consisting of a finite set of poses. The discretization interval should be similar to the size of the lattice cells so that adjacent points on the path are in adjacent lattice cells. Each pose along a discretized path consists of a location in $\mathcal{Q}$ and direction in $\mathbb{S}^1$. Each direction state also consists of a location and direction; however their values are chosen from the finite sets of lattice points and lattice directions. If the poses corresponded exactly to direction states in $\mathcal{X}$, we would increase the value of the corresponding element of $\boldsymbol{A}$ by 1. In reality, the poses are not necessarily located at lattice points and do not necessarily have lattice directions, so we first convert each pose to a convex combination of nearby direction states (Figure 5.9). For each pose of the path, we compute a convex coefficient $\gamma_i \in (0, 1]$ for each nearby direction state, x_i. For two consecutive poses (q_0, θ_0) and (q_1, θ_1), we use the convex coefficients for both poses and increase several elements of $\boldsymbol{A}$ by the product of two convex coefficients—one relating to the start pose and another to the end pose. After

modifying $\boldsymbol{A}$ for every pair of poses in the historic paths, we need to normalize $\boldsymbol{A}$ so that it is left-stochastic and can be used as a transition matrix. If only a few historic paths are known, we can add the transition matrix obtained from the historic data with one based on generic, random motion so that there is some small probability of transitions not represented in the data that are nevertheless still physically possible.

Algorithm 5.1: Historic transition probabilities

Input: Set of historic paths, $\mathcal{C}_{\text{hist}}$; and set of direction states $\mathcal{X}$
Output: Transition matrix, $\boldsymbol{A}$

```
A ← square zero matrix with dimension |X|
for historic path p ∈ C_hist do
    pdisc ← discretized version of p
    for consecutive poses (q_0, θ_0), (q_1, θ_1) of pdisc do
        X_0 ← nearby direction states of (q_0, θ_0)
        X_1 ← nearby direction states of (q_1, θ_1)
        for direction state x_i ∈ X_0 do
            γ_i ← convex coefficient relating (q_0, θ_0) and x_i
            for direction state x_j ∈ X_1 do
                γ_j ← convex coefficient relating (q_1, θ_1) and x_j
                a^i_j ← a^i_j + γ_i γ_j        /* (i, j)^th element of A */
Normalize A so each column sums to 1
return A
```

To verify a model based on historic data, we can compare the density of historic paths to the *stationary distribution* of the HMM based on those paths (Figure 5.10). A stationary distribution is a distribution which does not change when updated using the transition matrix [147]. A Markov model has a stationary distribution if it is

- **Irreducible**: any state can be reached from any other state;
- **Positive recurrent**: the expected time to return to any state after leaving it is finite; and
- **Aperiodic**: For any state, the greatest common divisor of the possible times

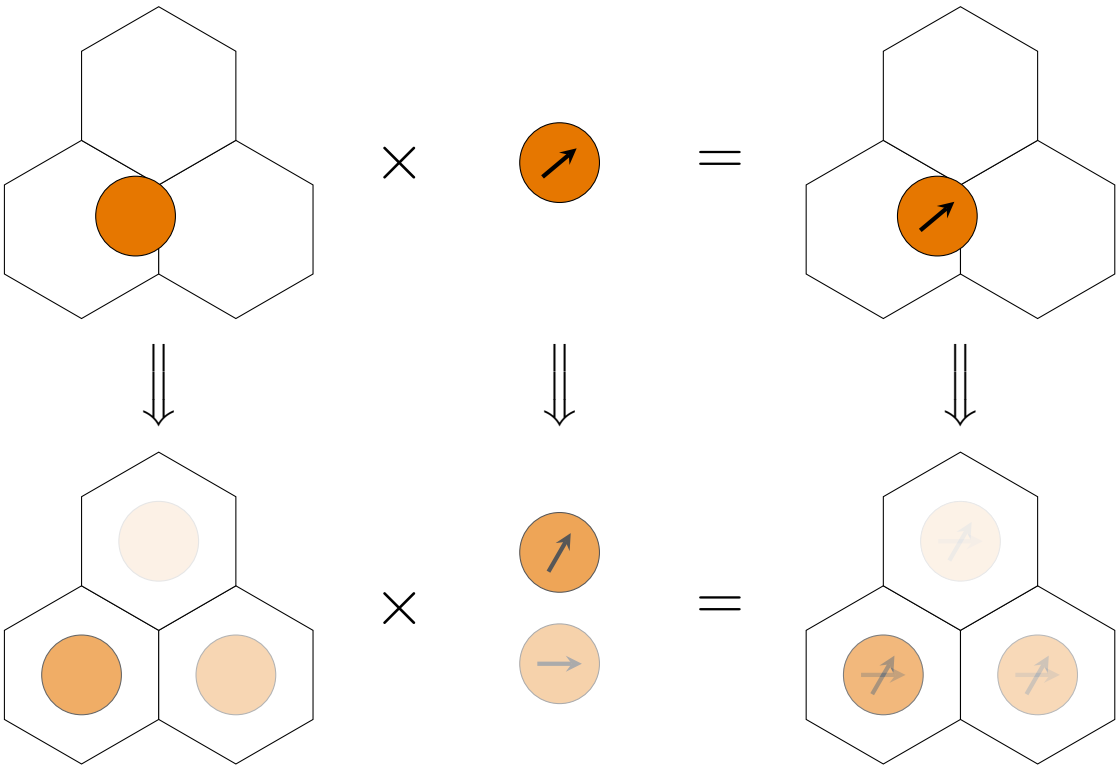


Figure 5.9: An arbitrary location can be converted into a convex combination of nearby lattice cells (left). The maximum number of cells needed in this combination is 3 for a hexagonal lattice, 4 for a square lattice, and 6 for a triangular lattice. Similarly, an arbitrary direction is a convex combination of at most 2 lattice directions (center). As poses are Cartesian products of locations and directions, a convex representation of any pose can be obtained from the Cartesian products of the convex representations of its location and direction (right).

it takes to return to that state is 1.

Except for specially constructed pathological environments, these properties will all hold for the HMMs that I consider. For an HMM with a unique stationary state distribution, it can be computed by iteratively applying the transition matrix to any initial state distribution, or by computing the eigenvectors of the transition matrix. The stationary cell distribution can be computed by multiplying the stationary state distribution by the projection matrix. If we use a large amount of accurate historic or simulated data, the stationary cell distribution will closely reflect the density of paths through each cell, indicating that the HMM accurately

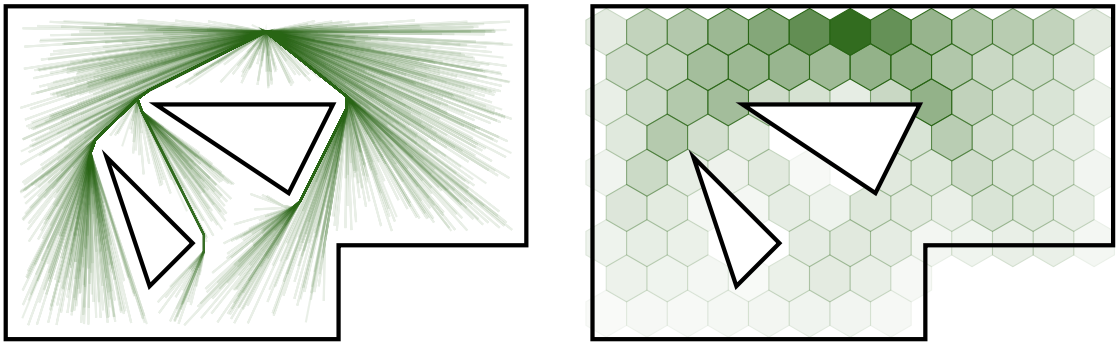


Figure 5.10: Simulated paths for a robot that moves from one fixed location to a random location in the environment and then returns to the fixed location (left). The stationary distribution of an HMM based on these paths (right) closely resembles the density of paths through each lattice cell.

captures the target's behavior.

5.4 Effects of observations

When searching, a robot can use an HMM model, (5.7)–(5.8), to predict how its target is behaving and where its target will be. However, it also has another useful source of information: its own observations. These observations fall under two categories:

- **Positive observations** where the searcher either communicates with the target or senses it in its field of view; and
- **Negative observations** where the searcher observes some part of the environment but does not see or establish communication with its target.

Both kinds of observations can be used by the searcher—albeit in slightly different ways—to get a more precise belief.

The general problem of using a sequence of observations to update a belief is called Bayes filtering [183] and its many forms are important throughout robotics. Using all available information—current as well as any older observations—the

belief that the target is in state $x_i \in \mathcal{X}$ is

$$w_i[\tau] = \mathbb{P}(x[\tau] = x_i \mid \text{current observation, older observations}).$$

Applying Bayes' law, the belief equals

$$w_i[\tau] = \frac{\mathbb{P}(\text{current} \mid x[\tau] = x_i, \text{older})\mathbb{P}(x[\tau] = x_i \mid \text{older})}{\mathbb{P}(\text{current} \mid \text{older})}$$

The term in the denominator does not depend on i so it will be constant for every state $x_i \in \mathcal{X}$ and is simply a normalization factor. In the first term in the numerator, the current observation only depends on the current state, and is independent of the older information. The second term in the numerator is the probability that the state is x_i given older information, not including the recent observation. We already have a formula for computing that as (5.7) uses all the older information to predict the current state from the previous state belief. Therefore this second term is simply the j^{th} element of (5.7) which is $\boldsymbol{a}_j \boldsymbol{w}[\tau - 1]$, the inner product of $\boldsymbol{A}$'s j^{th} row with the previous belief. Using these three simplifications, the belief can be updated by the Bayes filter

$$w_i[\tau] \propto \mathbb{P}(\text{current observation} \mid x[\tau] = x_i)\big(\boldsymbol{a}_i \boldsymbol{w}[\tau - 1]\big). \tag{5.10}$$

This equation can be used to update a belief using any kind of observation; however the first term will be computed differently for different kinds of observations.

5.4.1 Positive observations

For cooperative robots, a positive observation often means that the two robots are close enough to each other that they can communicate. These robots can share lots of information, including their current location and immediate plans. With this information, the searcher knows its target's state exactly. When these two

robots inevitably separate again, they will again need to maintain a belief of the other's state. This belief will be updated using a new HMM whose path states are created from the target's most recent plan.

It is also possible that the searcher observes its target but isn't able to communicate with it. In this scenario, the searcher now has an observation $\boldsymbol{z}[\tau] \in \mathbb{R}^{|\mathcal{Y}|}$ of the target's physical location. Depending on the quality of the searcher's sensors, this observation may be very precise, resulting in $\boldsymbol{z}$ only containing a single non-zero element corresponding to the target's actual location and heading, or it may have several non-zero elements corresponding to several nearby cells. The elements of this vector are

$$z_j[\tau] \propto \mathbb{P}(\text{observation} \mid y[\tau] = y_j).$$

These elements are not the probability that the target is located in cell y_j—the target is only located in a single location. Instead, they are the probabilities that the searcher would make its current observation (a set of signals recorded by its sensors) if the target was in cell y_j. For positive observations, we can express (5.10) as

$$w_i[\tau] \propto \left(\sum_{y_j \in \mathcal{Y}} \mathbb{P}(\text{observation} \mid y[\tau] = y_j) \mathbb{P}(y[\tau] = y_j | x[\tau] = x_i) \right) \big(\boldsymbol{a}_i \boldsymbol{w}[\tau - 1] \big)$$

The two terms inside the sum are both things we've already seen! The first term is the $z_j[\tau]$ I just described; the second term corresponds to an element, p^i_j of the projection matrix, $\boldsymbol{P}$, as defined in (5.9). The sum is therefore the inner product of $\boldsymbol{z}[\tau]$ with the i^{th} column of $\boldsymbol{P}$ and so the belief is

$$w_i[\tau] \propto \big((\boldsymbol{p}^i)^\top \boldsymbol{z}[\tau] \big) \big(\boldsymbol{a}_i \boldsymbol{w}[\tau - 1] \big).$$

Aggregating this expression for all i, we get the update rule for the entire vector,

$$\boldsymbol{w}[\tau] \propto \big(\boldsymbol{P}^\top \boldsymbol{z}[\tau]\big) \odot \big(\boldsymbol{A}\boldsymbol{w}[\tau-1]\big) \tag{5.11}$$

where $\odot : \mathbb{R}^{|\mathcal{X}|} \times \mathbb{R}^{|\mathcal{X}|} \to \mathbb{R}^{|\mathcal{X}|}$ denotes the elementwise product of two vectors. We need to use the elementwise product as each element of $\boldsymbol{w}$ is the product of two scalar expressions (which each aggregate to become a vector). This belief update rule, (5.11), takes the update rule without an observation, (5.7), and then multiplies each element by a corresponding element of $\boldsymbol{P}^\top \boldsymbol{z}[\tau]$ which is the probability of making the observation if the target is in each state.

5.4.2 Negative observations

A negative observation is when the searcher is not able to communicate with its target. Although negative observations contain relatively little information—the target could be anywhere that the searcher can't see—they still help improve the searcher's belief. By incorporating its negative observations into a belief, the searcher essentially records where it has already searched so that it is less likely to plan a search path that rechecks recently checked locations. To use a negative observation, the searcher must be aware of its own location, q_{sea}, and which cell of $\mathcal{Y}$ that location corresponds to. We assume that each robot is equipped with a localization system that provides a sufficiently accurate estimate of q_{sea} at any time. The robot can then use this estimate to compute its current cell. Since $\mathcal{Y}$ is a lattice, it can check if a given point is in a given cell in constant time and compute its current cell in $\mathcal{O}(|\mathcal{Y}|)$ using a brute force approach.

When a negative observation is made, (5.10), is equivalent to

$$w_i[\tau] \propto \mathbb{P}(\text{no communication} \mid x[\tau] = x_i)\big(\boldsymbol{a}_i \boldsymbol{w}[\tau-1]\big) \tag{5.12}$$

The first term can be expressed as a sum over all the cells as

$$\begin{aligned}
&\mathbb{P}(\text{no communication} \mid x[\tau] = x_i) \\
&\quad = \sum_{y_j \in \mathcal{Y}} \mathbb{P}(\text{no communication} \mid y[\tau] = y_j)\mathbb{P}(y[\tau] = y_j \mid x[\tau] = x_i) \\
&\quad = 1 - \sum_{y_j \in \mathcal{Y}} \mathbb{P}(\text{communication} \mid y[\tau] = y_j)\mathbb{P}(y[\tau] = y_j \mid x[\tau] = x_i) \qquad (5.13)
\end{aligned}$$

In Section 5.2, I defined the communication matrix, $\boldsymbol{C}$, which is used to compute the probability of communication via (5.2). If the target is in cell y_j, this probability is simply $\boldsymbol{c}_j \boldsymbol{b}_{\text{sea}}$ where $\boldsymbol{c}_j$ is the row of $\boldsymbol{c}$ corresponding to cell y_j. With this equation and the fact that the second term in its sum is p_j^i, we can write (5.13) as

$$\mathbb{P}(\text{no communication} \mid x[\tau] = x_i) = 1 - \sum_{y_j \in \mathcal{Y}} p_j^i \boldsymbol{c}_j \boldsymbol{b}_{\text{sea}} = 1 - (\boldsymbol{p}^i)^\top \boldsymbol{C} \boldsymbol{b}_{\text{sea}}$$

Substituting this expression back into (5.12), we obtain

$$w_i[\tau] \propto \big(1 - (\boldsymbol{p}^i)^\top \boldsymbol{C} \boldsymbol{b}_{\text{sea}}\big)\big(\boldsymbol{a}_i \boldsymbol{w}[\tau - 1]\big)$$

Aggregating this expression for all i, we get the update rule for the entire vector,

$$\boldsymbol{w}[\tau] \propto \big(\mathbf{1} - \boldsymbol{P}^\top \boldsymbol{C} \boldsymbol{b}_{\text{sea}}[\tau]\big) \odot \big(\boldsymbol{A}\boldsymbol{w}[\tau - 1]\big) \qquad (5.14)$$

where $\mathbf{1} \in \mathbb{R}^{|\mathcal{X}|}$ is the vector consisting of all ones.

The overall update rule for negative observations, (5.14), is quite similar to the update rules for no observation, (5.7), and positive observations, (5.11). The difference is that it multiplies the belief from the HMM, $\boldsymbol{A}\boldsymbol{w}[\tau - 1]$ by a communication term. Each element of this term is the probability that a target in that state can communicate with the searcher. All the states in cells near the searcher will have their probabilities set to 0 resulting in a more precise belief (Figure 5.11).

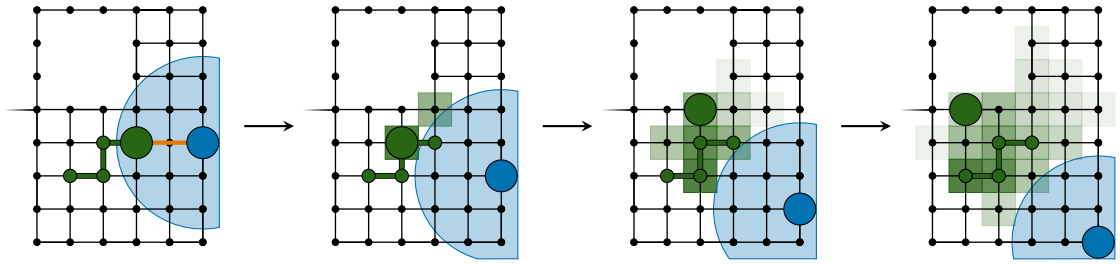


Figure 5.11: When two robots disconnect, the searcher (blue) maintains a belief of where the target (green) is located. It updates this belief using an HMM that is biased towards the targets planned path but also includes a small probability that the target will abandon its path. The searcher also incorporates negative observations into the belief by setting the probability of anywhere it can see equal to zero.

5.5 Combining beliefs

When two robots connect, they can combine their belief of a third robot's state. As this merged belief is based on observations made by both robots it will be more accurate than either robots' individual belief. The main factor which determines how the searchers merge their beliefs is whether or not they are using the same HMM for the target. In an ideal situation, cooperative searchers have a common map of the environment and can share all the parameters used to construct the HMM. However, in reality, the searchers may have never met before so they could have different maps of the environment and may be updating their beliefs with very different HMMs.

5.5.1 Searchers with a shared model

Suppose both searchers have a shared HMM. In this case, their beliefs are defined over a common set of states, $\mathcal{X}$, and so their belief vectors $\boldsymbol{w}^0, \boldsymbol{w}^1 \in \mathbb{R}^{|\mathcal{X}|}$ consist of elements corresponding to equivalent probabilities:

$$w_i^0 = \mathbb{P}(x = x_i \mid \text{robot 0's observations})$$

$$w_i^1 = \mathbb{P}(x = x_i \mid \text{robot 1's observations}).$$

The goal of merging beliefs is to compute

$$w_i^{0,1} = \mathbb{P}(x = x_i \mid \text{both robots' observations}) \tag{5.15}$$

which is an element of the merged belief $\boldsymbol{w}^{0,1}$. A conservative method of combining multiple distributions into a more accurate one is to elementwise minimum of the distributions [83]. This method sets $w_i^{0,1}$ equal to $\min\{w_i^0, w_i^1\}$ which is not strictly correct because $\min\{w_i^0, w_i^1\}$ only depends on one robot's observations but $w_i^{0,1}$ should depend on both robots' observations. To determine a better method for merging beliefs, we apply Bayes' law to (5.15):

$$\begin{aligned} w_i^{0,1} &= \mathbb{P}(x = x_i \mid \mathcal{I}_0, \mathcal{I}_1) && (5.16)\\ &= \frac{\mathbb{P}(\mathcal{I}_1 \mid x = x_i, \mathcal{I}_0)\mathbb{P}(x = x_i \mid \mathcal{I}_0)}{\mathbb{P}(\mathcal{I}_1 \mid \mathcal{I}_0)} && (5.17)\\ &= \frac{\mathbb{P}(\mathcal{I}_0 \mid x = x_i, \mathcal{I}_1)\mathbb{P}(x = x_i \mid \mathcal{I}_1)}{\mathbb{P}(\mathcal{I}_0 \mid \mathcal{I}_1)} && (5.18) \end{aligned}$$

where $\mathcal{I}_0$ and $\mathcal{I}_1$ refer to all the observations made by robots 0 and 1, respectively. Multiplying (5.17) by (5.18) and taking the square root yields an expression that is symmetric in $\mathcal{I}_0, \mathcal{I}_1$:

$$w_i^{0,1} = \eta_i \sqrt{\mathbb{P}(x = x_i \mid \mathcal{I}_0)\mathbb{P}(x = x_i \mid \mathcal{I}_1)} = \eta_i \sqrt{w_i^0 w_i^1} \tag{5.19}$$

where the normalization term is

$$\eta_i = \sqrt{\frac{\mathbb{P}(\mathcal{I}_1 \mid x = x_i, \mathcal{I}_0)\mathbb{P}(\mathcal{I}_0 \mid x = x_i, \mathcal{I}_1)}{\mathbb{P}(\mathcal{I}_1 \mid \mathcal{I}_0)\mathbb{P}(\mathcal{I}_0 \mid \mathcal{I}_1)}}.$$

The denominator of η_i is a normalization term because it does not depend on i. The terms in the numerator of η_i are the probabilities that one robot would

make its observations if the target is in state x_i and the other robot's made its observations. We will assume that these probabilities are similar for different states (i.e. regardless of where the target is, both searchers are likely to make similar observations) and so η_i is approximately constant for any i. Under this assumption, we can use (5.19) with the same η for any element of $\boldsymbol{w}^{0,1}$ and therefore the merged distribution is

$$\boldsymbol{w}^{0,1} \propto \sqrt{\boldsymbol{w}^0 \boldsymbol{w}^1} \tag{5.20}$$

which is simply the geometric mean of the two distributions (Figure 5.12). This operation can be generalized to merging $m \geq 2$ beliefs by multiplying them all together elementwise and then taking the m^{th} root.

The geometric mean has two properties that make it an attractive method of merging beliefs:

1. If any robot knows that the target cannot be in state x_i (i.e. $w_i^0 = 0$), then the merged belief will also have $w_i^{0,1} = 0$; and

2. If two robots have the exact same belief, merging them will not change the belief.

Although using the geometric mean is an approximation—η_i does depend somewhat on i—it satisfies both these two properties making it a reasonable approximation. Merging by taking the minimum of the two beliefs [83] also satisfies both properties, however it does not use information from all the robots or have a statistical justification like the geometric mean has.

5.5.2 Searchers with incompatible models

A searcher's belief about a target is based on its model of the environment and the target's behavior. The robot's map—its model of the environment—is typically constructed from that robot's (and possibly its teammates') noisy observations.

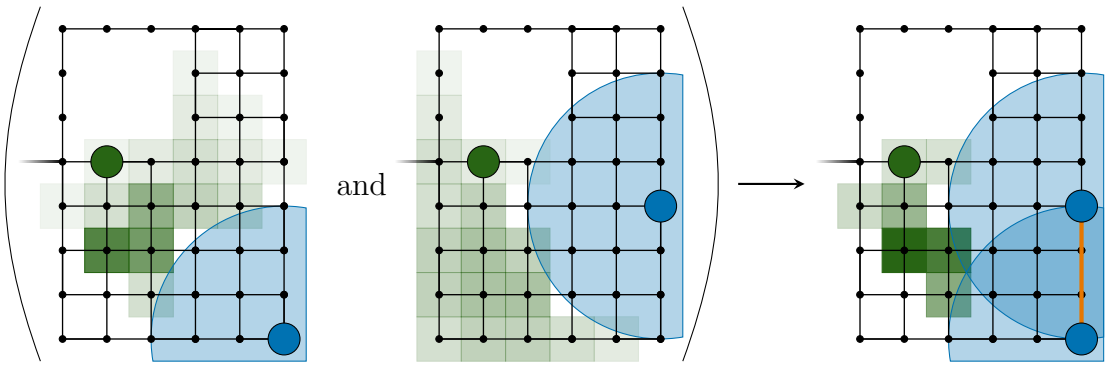


Figure 5.12: When two robots meet, they can merge their beliefs about a third robot by taking the geometric mean of their individual beliefs. The merged belief will be a narrow distribution which combines observations made by both robots.

Two robots who have never met will usually have similar, but not identical, maps. Different maps result in different lattices, and different states in the HMMs. Furthermore, they may have different models of a common target's behavior if they have different beliefs about the target's velocity, or different historic data about its past behavior.

These differences between the searchers' models mean that their HMMs use different sets of states, $\mathcal{X}_0$ and $\mathcal{X}_1$. Their belief vectors $\boldsymbol{w}^0 \in \mathbb{R}^{|\mathcal{X}_0|}$ and $\boldsymbol{w}^1 \in \mathbb{R}^{|\mathcal{X}_1|}$ are elements of different spaces and so it does not make any sense to try merging them by taking their geometric mean as was done in (5.20). To exploit information from the other searcher, they can either

(a) Treat the other robot's knowledge as an observation and continue to use different HMMs; or

(b) Combine their models into a single model and initialize an HMM based on that model with a state belief that combines information from both robots.

Although either approach could be used the first is much simpler as the robots do not need to merge their maps or change all their beliefs to different HMMs.

When a searcher treats another searcher's belief as an observation, it can use

it to update its own state belief in an equation similar to (5.11). Suppose robot 0 receives the cell belief $\boldsymbol{z}_1 \in \mathbb{R}^{|\mathcal{Y}_1|}$ from robot 1. As the robots have different maps, their lattices, $\mathcal{Y}_0$ and $\mathcal{Y}_1$ are not necessarily the same. Instead they can be related by a transformation matrix $\boldsymbol{T}_0^1 : \mathbb{R}^{|\mathcal{Y}_1|} \to \mathbb{R}^{|\mathcal{Y}_0|}$ where the $(i,j)^{\text{th}}$ element is

$$t_i^j = \frac{\text{Area of overlap between } y_i \in \mathcal{Y}_0 \text{ and the } y_j \in \mathcal{Y}_1}{\text{Area of overlap between all cells of } \mathcal{Y}_0 \text{ and } y_j \in \mathcal{Y}_1}$$

To compute t_i^j, the robots must share their lattices, $\mathcal{Y}_0$ and $\mathcal{Y}_1$, with each other. Once these lattices have been shared, both robots can compute $\boldsymbol{T}_0^1$ ins $\mathcal{O}(|\mathcal{Y}_0||\mathcal{Y}_1|)$ since the area of overlap of two lattice cells can be computed in constant time. If a cell of $\mathcal{Y}_1$ overlaps with any cell of $\mathcal{Y}_0$, its corresponding column of $\boldsymbol{T}_0^1$ will sum to 1, and so $\boldsymbol{T}_0^1$ is left-stochastic if every cell of $\mathcal{Y}_1$ overlaps with one or more cell of $\mathcal{Y}_0$. Furthermore, if $\mathcal{Y}_1$ and $\mathcal{Y}_0$ cover the exact same region and have the same number of cells then $\boldsymbol{T}_0^1$ is doubly-stochastic and $\left(\boldsymbol{T}_0^1\right)^\top = \boldsymbol{T}_1^0$. This transformation converts a belief over $\mathcal{Y}_1$ to a belief over $\mathcal{Y}_0$ by

$$\widetilde{\boldsymbol{z}}_0 = \boldsymbol{T}_0^1 \boldsymbol{z}_1.$$

The resulting belief is only an approximation as some information is lost when representing a belief over a specific lattice and $\boldsymbol{T}_0^1 \boldsymbol{T}_1^0 \neq \boldsymbol{I}$ unless the two lattices are identical. Treating the transformed version of robot 1's cell belief as a new observation, robot 0 can update its own state belief by

$$\boldsymbol{w}_0^{0,1}[\tau] = \left(\boldsymbol{P}_0^\top \boldsymbol{T}_0^1 \boldsymbol{z}_1[\tau]\right) \odot \boldsymbol{w}_0[\tau]$$

which is analogous to the effect of the observation in (5.11). By updating their beliefs in this way, the two robots can share information without needing to combine their maps and HMMs. However, the resulting merged beliefs are not as precise as would be obtained by merging compatible beliefs using (5.20).

5.6 Evaluating search paths

When a robot decides it needs to find another robot, it can use its belief vector, $\boldsymbol{b}$, to plan a path for reconnection. Although we would like to minimize the expected time required to find the target, this objective is only feasible if we can guarantee that the target is found eventually. For stationary targets in a finite environment, it is possible to guarantee success in finite time so this objective is used [135, 167]. However, for mobile targets, it is not in general possible to guarantee success in a finite time, and the problem of determining the number of searchers needed to guarantee success is NP-hard [136]. As there will usually be a small chance of search for a moving target failing, the expected time to success is not necessarily well-defined and is therefore not a useful optimization criterion.

Instead of trying to minimize the expected time until success, we could try to maximize the probability of success [69, 115, 162, 203]. Although this criterion will result in paths that are very likely to be successful, it does not distinguish between paths that find the target quickly versus paths that find it slowly. To prioritize paths based on how soon the target is found, we use a discounted reward function [81]. For an infinitely long discrete path, $p = (q[0], q[1], \dots)$, the discounted reward is

$$J_{\text{inf}}(p) = \sum_{\tau=1}^{\infty} \beta^{\tau-1} \Delta\phi[\tau] \tag{5.21}$$

where $\beta \in [0, 1]$ is the discount factor and $\Delta\phi[\tau]$ is the probability of finding the target for the first time at time step τ when following p given some initial target belief $\boldsymbol{w}[0]$. This reward function uses the discount factor β to scale the reward so that there is a lower reward for finding the target slowly. Choosing β close to 0 results in greedy behavior that prioritizes a high probability of finding the target soon. Choosing β close to 1 results in a more conservative search that prefers paths that have a high probability of finding the target eventually.

The reward of an infinite length path is based on the probabilities of finding the target for the first time at each time step. These probabilities are

$$\begin{aligned}\Delta\phi[\tau] &= \mathbb{P}\left(\text{First connected to the target at time } \tau\right) \\ &= \mathbb{P}\left(\text{Connected by time } \tau\right) - \mathbb{P}\left(\text{Connected by time } \tau - 1\right) \\ &= \phi[\tau] - \phi[\tau - 1] \end{aligned} \tag{5.22}$$

where $\phi[\tau]$ is the probability that the two robots have been connected at least once by time τ. By using ϕ, we can compute $\Delta\phi$ for the first several locations along the path iteratively by keeping track of the cumulative probability of having found the target so far. To compute $\phi[\tau]$, we will define a new vector $\widehat{\boldsymbol{w}}[\tau] \in \mathbb{R}^{|\mathcal{X}|}$ whose elements are the probability that the target is in each state and has never been connected to the searcher between steps 0 and τ. This vector is defined so that

$$\phi[\tau] = 1 - \mathbf{1}^\top \widehat{\boldsymbol{w}}[\tau]. \tag{5.23}$$

As the searcher and target are never connected at time 0, $\widehat{\boldsymbol{w}}[0] = \boldsymbol{w}[0]$, the initial belief for the search problem. When computing the state update law (5.14), the normalization factor is the inverse of the probability that the robots were not connected in the previous time step. The state belief at time step τ could be computed from $\boldsymbol{w}[0]$ by applying (5.14) τ times which would involve τ of these normalization factors. The product of all of these normalization factors is the inverse of the probability that the robots were not connected between time steps 0 and τ. Therefore we can update $\widehat{\boldsymbol{w}}[\tau]$ by simply applying (5.14) without the normalization factor:

$$\widehat{\boldsymbol{w}}[\tau] = \left(\mathbf{1} - \boldsymbol{P}^\top \boldsymbol{C} \boldsymbol{b}_{\text{sea}}[\tau]\right) \odot \left(\boldsymbol{A}\widehat{\boldsymbol{w}}[\tau - 1]\right) \tag{5.24}$$

Once $\boldsymbol{w}[\tau]$ has been computed, we can compute $\Delta\phi[\tau]$ using (5.22)–(5.23). As

these computations only rely on $\widehat{\boldsymbol{w}}[\tau]$ and $\phi[\tau]$, it is possible to compute the cost of a path by iterating through the path's locations and using the data about the previous location to compute the data at the next location. For the first location, we use the initial condition $\phi[0] = 0$ and for all subsequent locations, $\phi[\tau - 1]$ will have been computed at the previous location.

5.6.1 Reward of finite length paths

When planning a search path, we will plan finite length paths by adding one location at a time to existing paths which results in a planning tree. As (5.21) is for infinite length paths, we must use a slightly different reward function to evaluate the paths that we will plan. The obvious modification is to simply truncate the sum of (5.21) to obtain a finite path reward

$$J_{\min}(p) = \sum_{\tau=1}^{T} \beta^{\tau-1} \Delta\phi[\tau] \tag{5.25}$$

where T is the length of the path p. This reward is a lower bound for the reward, $J_{\inf}$, of any infinite length path which starts with p and continues after reaching the end of p. Unfortunately, this naïve approach is not particularly useful as it tends to give higher rewards to longer paths as there are more non-negative terms in the sum. To avoid a bias towards long paths, I will use the reward function

$$J(p) = \max_{p' \in \{\text{paths starting with } p\}} \{J_{\inf}(p')\} \tag{5.26}$$

which rewards short paths for the potential future value of the path that the robot could follow after reaching the end of the path. This reward function results in paths which satisfy Bellman's principal of optimality: "An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from

the first decisions." [27]. This property is very useful as it means that the searcher can plan a single path and follow it to its end without needing to replan every time it reaches a new location.

Lemma 5.1. *For some finite length path p, let p' be the infinite length path which maximizes J_{inf} subject to the constraint of starting with p. Then p maximizes J if and only p' maximizes J_{inf} among all possible infinite length paths.*

Proof. By the definition of p', we have that $J(p) = J_{\text{inf}}(p')$. First, suppose p' maximizes J_{inf}. For any $\widehat{p}$, let $\widehat{p}'$ be such that $J(\widehat{p}) = J_{\text{inf}}(\widehat{p}')$. Then since p' maximizes J_{inf}, it is also true that

$$J(\widehat{p}) = J_{\text{inf}}(\widehat{p}') \leq J_{\text{inf}}(p') = J(p).$$

This expression holds for any $\widehat{p}$ and so p maximizes J.

Now suppose that p maximizes J. Let $\widehat{p}'$ be some other infinite length path and $\widehat{p}$ be a path consisting of the first several locations of $\widehat{p}'$. Then by the definition of J, $J_{\text{inf}}(\widehat{p}') \leq J(\widehat{p})$ and since p maximizes J, $J(\widehat{p}) \leq J(p)$. Combining these inequalities with the fact that $J(p) = J_{\text{inf}}(p')$, we have

$$J_{\text{inf}}(\widehat{p}') \leq J(\widehat{p}) \leq J(p) = J_{\text{inf}}(p')$$

This expression is true for any $\widehat{p}'$ and so p' maximizes J_{inf}. □

Theorem 5.1. *Let $p^* = (q[0], q[1], \ldots, q[T])$ be the optimal path when the searcher starts at $q[0]$ with target belief $\boldsymbol{w}[0]$. Then $\widetilde{p}^* = (q[1], \ldots, q[T])$ is the optimal path when the searcher starts at $q[1]$ with target belief $\boldsymbol{w}[1]$, the belief obtained by updating $\boldsymbol{w}[0]$ using (5.14) with a negative observation at $q[1]$.*

Proof. We will refer to the rewards for paths starting at $q[0]$ with belief $\boldsymbol{w}[0]$ by J and J_{inf} and the rewards for paths starting at $q[1]$ with belief $\boldsymbol{w}[1]$ by $\widetilde{J}$ and

$\widetilde{J}_{\text{inf}}$. Let $p' = (q[0], q[1], \dots)$ be some infinite length path and $\widetilde{p}' = (q[1], \dots)$ be the same path starting at $q[1]$ instead of $q[0]$. We will relate these two paths' respective rewards. By relabelling (5.21), the reward for $\widetilde{p}$ is equal to

$$\widetilde{J}_{\text{inf}}(\widetilde{p}') = \sum_{\tau=2}^{\infty} \beta^{\tau-2} \Delta\widetilde{\phi}[\tau]$$

where $\Delta\widetilde{\phi}$ is the probability of finding the target for the first time at a certain time step given that it was not found at time step 1. It can be computed from $\widetilde{\phi}$, the cumulative probability of finding it by that time step given that it was not found at time step 1. This cumulative probability is related to the cumulative probability ϕ (without the condition of not finding it at time step 1) by

$$\begin{aligned}
\widetilde{\phi}[\tau] &= \mathbb{P}\left(\text{Connected by } \tau \mid \text{ Not connected by 1}\right) \\
&= \frac{\mathbb{P}\left(\text{Connected by } \tau \text{ and not connected by 1}\right)}{\mathbb{P}\left(\text{Not connected by } \tau\right)} \\
&= \frac{\phi[\tau] - \phi[1]}{1 - \phi[1]}.
\end{aligned}$$

Similarly, using this relationship, we can relate the marginal probabilities by

$$\Delta\widetilde{\phi}[\tau] = \widetilde{\phi}[\tau] - \widetilde{\phi}[\tau - 1] = \frac{\phi[\tau] - \phi[\tau - 1]}{1 - \phi[1]} = \frac{\Delta\phi[\tau]}{1 - \phi[1]}$$

and the infinite rewards are related by

$$\begin{aligned}
\widetilde{J}_{\text{inf}}(\widetilde{p}') &= \frac{1}{\beta} \sum_{\tau=2}^{\infty} \beta^{\tau-1} \frac{\Delta\phi[\tau]}{1 - \phi[1]} \\
&= \frac{1}{\beta(1 - \phi[1])} \sum_{\tau=2}^{\infty} \beta^{\tau-1} \Delta\phi[\tau] \\
&= \frac{1}{\beta(1 - \phi[1])} \left(-\Delta\phi[1] + \sum_{\tau=1}^{\infty} \beta^{\tau-1} \Delta\phi[\tau] \right) \\
&= \frac{1}{\beta(1 - \phi[1])} \left(J_{\text{inf}}(p') - \Delta\phi[1] \right)
\end{aligned}$$

This transformation consists of adding a constant term, $-\Delta\phi[1]$, and multiplying by a positive term, $\frac{1}{\beta(1-\phi[1])}$. As both of these operations preserve inequalities, if some path maximizes J_{inf} then the same path with the first location removed maximizes $\widetilde{J}_{\text{inf}}$.

As stated in the theorem, p^* is a finite length path which maximizes J so by Lemma 5.1, there exists an infinite length path starting with p^* which maximizes J_{inf}. Then this path, with its first vertex removed is an infinite length path which maximizes $\widetilde{J}_{\text{inf}}$. Applying the other direction of Lemma 5.1, any finite truncation of that path also maximizes its corresponding finite length reward function. Truncating to paths with $T-1$ vertices, the path $(q[1],\ldots,q[T]) = \widetilde{p}^*$ must maximize $\widetilde{J}_{\text{inf}}$. □

5.6.2 Comparison through bounds

To compare two paths, we would ideally compare the true reward $J(p)$ of each path, which is based on the best path starting with p. As this reward does not depend on a path's length it can be used to fairly compare paths of different lengths. If $J(p_0) > J(p_1)$ then the best path starting with p_0 is better than the best path starting with p_1. Although, J is conceptually quite simple, it is impractical to compute a maximum over a set of infinite length paths. Fortunately, it is easy to bound J as all of the $\Delta\phi[\tau]$ in (5.26) are non-negative and they sum to 1. We already saw a lower bound for J in (5.25). (This bound is met for an infinite length path which never has any chance of finding the target after reaching the end of p.) When extending a path by adding an additional vertex, we can express this lower bound recursively. Let $p_1 = (q[0],\ldots,q[T],q[T+1])$ be a path that begins with $p_0 = (q[0],\ldots,q[T])$ and has one additional location. Its minimum possible reward is

$$J_{\min}(p_1) = \sum_{\tau=1}^{T+1} \beta^{\tau-1}\Delta\phi[\tau]$$

$$= \sum_{\tau=1}^{T} \beta^{\tau-1} \Delta\phi[\tau] + \beta^T \Delta\phi[T+1]$$

$$= J_{\min}(p_0) + \beta^T \Delta\phi[T+1]. \tag{5.27}$$

This expression can be used to quickly update $J_{\min}$ when extending a path by a single location in constant time.

On the other hand, the maximum possible value for J occurs along a path which is guaranteed to find the target in the time step immediately after the end of p. We call the reward of this hypothetical path $J_{\max}(p)$, and as any other infinite length path starting with p is guaranteed to have a lower cost,

$$J(p) \leq J_{\max}(p) = J_{\min}(p) + \beta^T \left(1 - \phi[\tau]\right). \tag{5.28}$$

As this expression is based on $J_{\min}(p)$, which will typically already be computed for an existing path, we can also compute $J_{\max}(p)$ in constant time.

Using (5.27)–(5.28), we can compute an upper and lower bound for any path p, and we can update the lower bound whenever we consider a new path which begins with p. To compare two paths, p_0 and p_1, we would like to compare their rewards $J(p_0)$, and $J(p_1)$, which we don't actually know. However, we do know that $J(p_i) \in [J_{\min}(p_i), J_{\max}(p_i)]$ where upper and lower bounds are known for any paths we are considering. If $J_{\min}(p_0) > J_{\max}(p_1)$, then it is guaranteed that $J(p_0) > J(p_1)$ and so we can be sure that p_0 is a better path even though we don't know the true reward of either path. On the other hand, if the two paths' reward ranges overlap, then we cannot say conclusively which path is better. In Algorithm 5.6, this criterion will be used to remove vertices—representing paths which are guaranteed to be sub-optimal—from a planning tree to reduce its size and decrease the computational burden of planning a search path.

5.7 Sampling based planner

Potential search paths can be compared using their reward bounds, $J_{\min}(p)$ and $J_{\max}(p)$, which are based on $\Delta\phi[\tau]$ for each location along a path. These marginal probabilities are computed recursively, with $\Delta\phi[T]$ based on $\phi[T]$ and $\Delta\phi[T-1]$. Due to this recursive relationship between the reward bounds, an effective way of constructing several candidate paths and evaluating their rewards is to iteratively add single vertices to existing candidate paths. Each vertex has a predecessor vertex—the former last vertex of the path it was added to—so the set of all these vertices naturally form a tree.

The planner (Algorithm 5.2) constructs a tree in a manner similar to LaValle's rapidly-exploring random trees (RRT) [116]. Each vertex of the tree corresponds to a finite length path, and it consists of a

- Location, q
- Belief, $\widehat{\boldsymbol{w}}$
- Total probability, ϕ
- Marginal probability, $\Delta\phi$
- Minimum reward, $J_{\min}$
- Maximum reward, $J_{\max}$

The location is the final location of the path and the other data are based on the unique path from the root vertex to the vertex. The root vertex is located at the searcher's current location and has the searcher's current belief of the target's state, but the remainder of its properties are initialized with default values (Algorithm 5.3). In each round of the algorithm, one or more vertices are added to the tree. Each new vertex is a child of an existing vertex and its properties are computed via (5.22)–(5.24), (5.27)–(5.28) using its location and its parent's properties. Similar to RRT, locations of new vertices are determined by travelling in the direction of randomly sampled locations. This sampling approach is biased to points far away from the existing tree, which allows it to grow towards unexplored

areas and quickly find a high quality search path. Once the algorithm reaches a stopping criterion—the number of vertices, number of rounds of the algorithm, or reward of the best path—the algorithm returns the tree.

Algorithm 5.2: Search tree

Input: Environment, $\mathcal{Q} \subset \mathbb{R}^2$; searcher's location $q_0 \in \mathcal{Q}$; and belief, $\widehat{\boldsymbol{w}}[0]$
Output: Search tree, $\mathcal{T}$

```
v_0 ← root vertex at q_0 with belief ŵ[0]          /* Algorithm 5.3 */
T ← planning tree consisting of v_0
v*(T) ← v_0                                        /* Best vertex in tree */
while stopping criterion is not met do
    q ← random location in Q
    Grow T by adding new vertices based on q       /* Algorithm 5.4 */
    if v*(T) has been updated then
        Prune T to remove low quality vertices
return T
```

Algorithm 5.3: Create root vertex

Input: Location q_0; and target belief, $\widehat{\boldsymbol{w}}[0]$
Output: Root vertex, v_0

```
q(v_0) ← q_0                                       /* Location */
τ(v_0) ← 0                                         /* Depth in tree */
ŵ(v_0) ← ŵ[0]                                      /* Belief vector (not normalized) */
φ(v_0) ← 0                                         /* Total probability of success */
Δφ(v_0) ← 0                                        /* Marginal probability of success */
J_min(v_0) ← 0                                     /* Reward lower bound */
J_max(v_0) ← 1                                     /* Reward upper bound */
return v_0
```

The best path is obtained from the planning tree by following the unique path from the root vertex to the best vertex, v^*. The best vertex is defined as the vertex which maximizes $J_{\min}$, the guaranteed reward when following that path not including potential future rewards. Throughout Algorithm 5.2, the planner keeps track of the best vertex—initially the root vertex. When new vertices are added, if a new vertex has a better $J_{\min}$ than the existing best vertex, the best vertex is updated. When the best vertex changes, $J_{\min}(v^*)$ increases and the planner prunes

the tree by removing any vertices which are guaranteed to have worse rewards than this new $J_{\min}(v^*)$.

5.7.1 Growing the tree

In each round of Algorithm 5.2, the planning tree grows as new vertices are added to it (Algorithm 5.4). Ideally the best vertex would be selected using the actual reward J, but as it is not computable, one of the bounds, $J_{\min}$ or $J_{\max}$, or a convex combination of them is used instead. For each existing vertex in the tree, the algorithm generates a candidate vertex v_{new} which is a child of that existing vertex. The location of v_{new} is located along the shortest path from its parent to a randomly sampled location. As in RRT, the same randomly sampled location is used to generate every candidate vertex in a given round. I chose to use the shortest path fully contained in the environment instead of the shortest direct path. This choice biases the directions examined by the planner based on the topology of the environment and prevents the planner from considering paths that would cause a collision with an obstacle. This bias is beneficial as the planner is more likely to consider directions which are highly connected topologically and these directions tend to be the most useful for search.

The new candidate vertex is a child of an existing vertex in the tree and its properties are based on this vertex and its location along the path to the sampled location (Algorithm 5.5). The child's belief is obtained by (5.24) which advances its parent's belief by one time step with a negative observation made at the child's location. This belief is not normalized, so its sum equals the probability of not finding the target when travelling from the root to the new child vertex. This fact is used to compute the total probability, ϕ, and the marginal probability is the difference between the child and parent's total probability. Once the total and marginal probabilities have been computed, they can be used to compute the reward bounds, $J_{\min}$ and $J_{\max}$ using inductive versions of (5.28)–(5.25).

Algorithm 5.4: Grow tree

Input: Environment, $\mathcal{Q} \subset \mathbb{R}^2$; planning tree, $\mathcal{T}$; and sampled location, $q \in \mathcal{Q}$
Output: Planning tree, $\mathcal{T}$, with additional vertices added

$\mathcal{V}_{\text{new}} \leftarrow \{\}$ `/* Set of vertices to add */`
for existing vertex $v \in \mathcal{T}$ **do**
  $p \leftarrow$ shortest path from v to q within $\mathcal{Q}$
  $q_{\text{new}} \leftarrow$ location reached by following p for one time step
  $v_{\text{new}} \leftarrow$ vertex at q_{new} with parent v `/* Algorithm 5.5 */`
  if $J_{\max}(v_{\text{new}}) \geq J_{\min}(v^*(\mathcal{T}))$ **then**
    if only adding one vertex **then**
      $v_{\text{old}} \leftarrow$ only vertex in $\mathcal{V}_{\text{new}}$
      if (v_{old} does not exist) or ($J(v_{\text{new}}) > J(v_{\text{old}})$) **then**
        $\mathcal{V}_{\text{new}} \leftarrow \{v_{\text{new}}\}$
    else if adding one vertex at each depth **then**
      $v_{\text{old}} \leftarrow$ only vertex in $\mathcal{V}_{\text{new}}$ with $\tau(v) = \tau(v_{\text{new}})$
      if (v_{old} does not exist) or ($J(v_{\text{new}}) > J(v_{\text{old}})$) **then**
        $\mathcal{V}_{\text{new}} \leftarrow \mathcal{V}_{\text{new}} \cup \{v_{\text{new}}\} \setminus \{v_{\text{old}}\}$
    else if adding as many vertices as possible **then**
      $\mathcal{V}_{\text{new}} \leftarrow \mathcal{V}_{\text{new}} \cup v_{\text{new}}$
for new vertex $v_{\text{new}} \in \mathcal{V}_{\text{new}}$ **do**
  Add v_{new} to $\mathcal{T}$
  if $J_{\min}(v_{\text{new}}) > J_{\min}(v^*(\mathcal{T}))$ **then**
    $v^*(\mathcal{T}) \leftarrow v_{\text{new}}$
return $\mathcal{T}$

Algorithm 5.5: Create child vertex

Input: Location q_1; and parent vertex, v_0
Output: Vertex, v_1

$q(v_1) \leftarrow q_1$ `/* Location */`
$\texttt{parent}(v_1) \leftarrow v_0$
$\tau(v_1) \leftarrow \tau(v_0) + 1$ `/* Depth in tree */`
$\widehat{\boldsymbol{w}}(v_1) \leftarrow \left(\mathbf{1} - \boldsymbol{P}^\top \boldsymbol{C}\boldsymbol{b}(q_1)\right) \odot (\boldsymbol{A}\widehat{\boldsymbol{w}}(v_0))$ `/* Belief (not normalized) */`
$\phi(v_1) \leftarrow 1 - \mathbf{1}^\top \widehat{\boldsymbol{w}}(v)$ `/* Total probability of success */`
$\Delta\phi(v_1) \leftarrow \phi(v_1) - \phi(v_0)$ `/* Marginal probability of success */`
$J_{\min}(v_1) \leftarrow J_{\min}(v_0) + \beta^{\tau(v_1)-1}\Delta\phi(v_1)$ `/* Reward lower bound */`
$J_{\max}(v_1) \leftarrow J_{\min}(v_1) + \beta^{\tau(v_1)}(1 - \phi(v_1))$ `/* Reward upper bound */`
return v_1

The new candidate vertex can be compared to other candidate vertices based on their reward bounds, $J_{\min}$ and $J_{\max}$, to determine which candidate vertices will be added to the tree. The total number of candidate vertices which will be added depends on the *growth strategy* used. I considered three growth strategies which determine how many vertices are added to the tree in each round (Figure 5.13):

1. **One new vertex per round**. The best vertex is the one which maximizes $J_{\min}$, $J_{\max}$, or some linear combination of them. After n rounds the tree contains $\mathcal{O}(n)$ vertices.

2. **One new vertex per layer**. Each vertex of the tree has a depth which is its distance from the root and a layer of the tree is a set of vertices which all have the same depth. The best new vertex in each layer is again chosen as the one that maximizes some combination of $J_{\min}$ and $J_{\max}$. As the tree contains $n + 1$ layers after n rounds, the number of vertices added in each round is $\mathcal{O}(n)$ and the total number of vertices after round n is $\mathcal{O}(n^2)$.

3. **One new vertex per existing vertex**. In this brute force strategy, all candidate vertices are added. As the number of vertices after round n is $\mathcal{O}(\exp(n))$, this strategy is only used to validate the quality of paths produced by the other strategies.

While Algorithm 5.4 considers each candidate vertex, it stores the best candidate vertices it has considered so far in a set, $\mathcal{V}_{\text{new}}$. When a new candidate vertex is considered, the algorithm adds it to $\mathcal{V}_{\text{new}}$ depending on the growth strategy and how it compares to other candidate vertices already in $\mathcal{V}_{\text{new}}$. For the strategies which do not add all vertices, when new vertices are added to $\mathcal{V}_{\text{new}}$, they often replace existing ones. Once all candidate vertices (one for each existing vertex) have been considered, every vertex in $\mathcal{V}_{\text{new}}$ gets added to the tree.

Which vertices get added in a given round depends heavily on which reward bound is used to rank candidate vertices. By the definitions of the two bounds, if

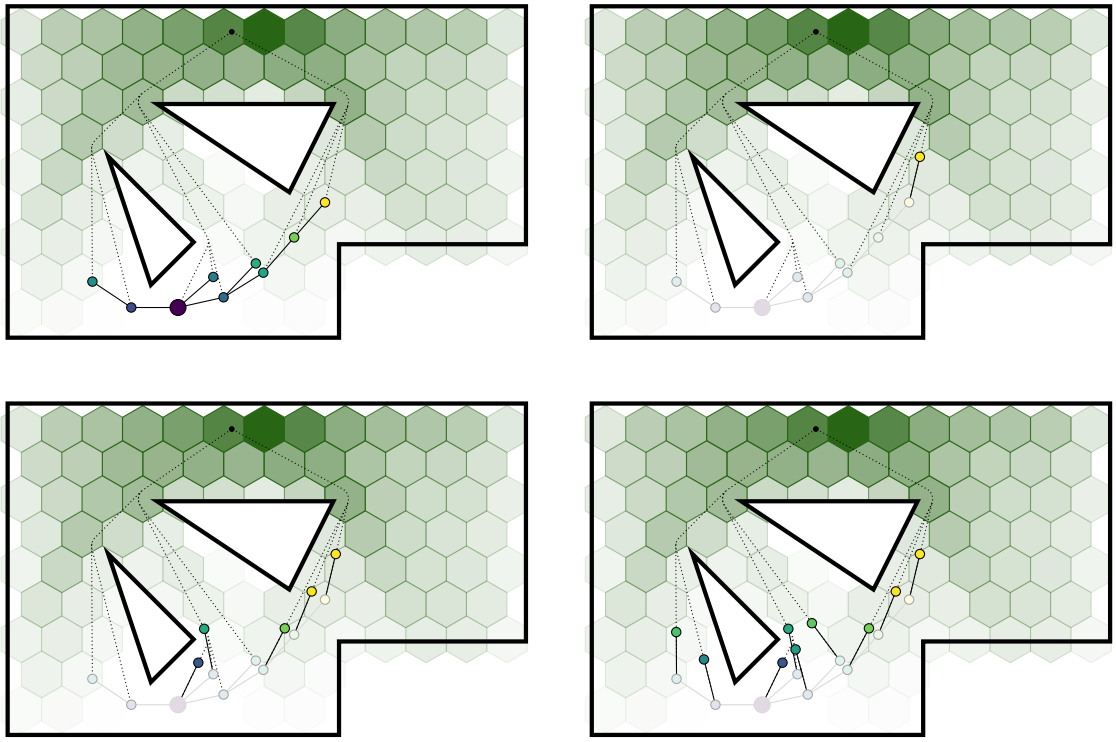


Figure 5.13: In each round of Algorithm 5.2, a randomly sampled location is used to add one or more vertices to the planning tree. This location is connected to each existing vertex of the tree by the shortest paths to it (top left) and new vertices are added by following these paths by a fixed distance. The three strategies for deciding which vertices to add to the tree are: adding a single vertex per sampled location (top right); adding one vertex at each depth of the tree (bottom left); or adding one vertex for every existing vertex in the tree (bottom right).

v_{child} is the child of v_{parent}, then it is always true that

$$J_{\max}(v_{\text{child}}) \leq J_{\max}(v_{\text{parent}})$$
$$J_{\min}(v_{\text{child}}) \geq J_{\min}(v_{\text{parent}}).$$

These inequalities show that $J_{\min}$ is higher for vertices deep in the tree, where as $J_{\max}$ is higher for vertices close to the root. As a result, when adding a single vertex per round, if the best vertex is chosen based on $J_{\min}$, the algorithm will favor adding vertices to the longest branch. On the other hand, if the best vertex is the one that maximizes $J_{\max}$, the algorithm will add many vertices adjacent to

the root. To avoid either of these biases, we use the heuristic reward

$$J_{\text{heur}}(v) = (1-\gamma)J_{\min}(v) + \gamma J_{\max}(v) \tag{5.29}$$

which is a convex combination with convex coefficient $\gamma \in [0,1]$. Varying γ results in drastically different planning trees when only adding one vertex per round (Figure 5.14). Smaller values of γ give higher weight to $J_{\min}$, resulting in a tree that grows too quickly and does not explore multiple branches A larger γ gives higher weight to $J_{\max}$, resulting in a tree that branches too often and remains clustered around the root. An intermediate value of γ can better balance growth and branching, resulting in a small tree whose best search path is optimal with respect to random samples used to grow the tree. Unfortunately, the possible range for γ which balances both behaviors is quite narrow, and can change for different environments or target beliefs.

When comparing different vertices at the same depth of the tree, there is a strong positive correlation between $J_{\min}$ and $J_{\max}$ (Figure 5.15). As this correlation is positive, for vertices, v_0 and v_1, at the same depth, it is usually true that

$$J_{\min}(v_0) > J_{\min}(v_1) \qquad \Longleftrightarrow \qquad J_{\max}(v_0) > J_{\max}(v_1)$$

and so the vertex which maximizes one reward likely maximizes the other. Therefore, within a layer, the optimal vertex is not very sensitive to the value of γ. As a result, the one-vertex-per-layer growth strategy produces similar planning trees for any heuristic reward, and in many cases, the same search path is returned for any heuristic reward (Figure 5.16). This insensitivity to γ makes adding one vertex per layer much more effective in practice than adding one vertex per sample despite its higher computational complexity.

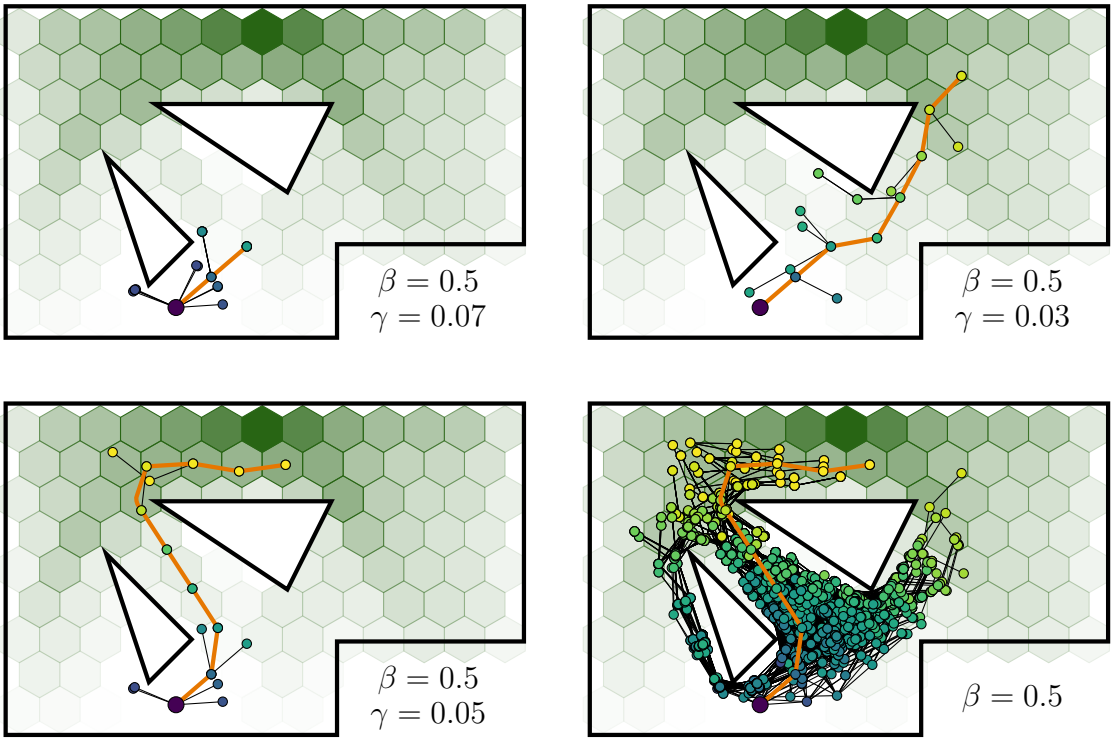


Figure 5.14: Different planning trees are obtained from the same set of sampled locations when different criteria are used to determine which vertices get added to the tree. When only a single vertex is added per sampled location, the vertex added is the one that maximizes the heuristic reward, (5.29). If γ is too small (top left), the heuristic favors adding vertices deep in the tree, resulting in a tree that expands away from the start location as fast as possible. If γ is too large (top right), the heuristic favors adding vertices shallow in the tree, and the tree remains clustered around the start location. For an intermediate value of γ (bottom left), these two tendencies are balanced. In this scenario, the optimal path in the tree is the same as the path obtained by a brute-force search (bottom right) where every round of Algorithm 5.2 adds one new vertex for each existing vertex.

5.7.2 Pruning the tree

When adding new vertices to the tree, one candidate vertex is considered for each candidate vertex in the tree. Therefore, the computational complexity of Algorithm 5.4 is directly proportional to the number of vertices in the tree, and so the planner can be made more efficient by keeping the number of vertices as small as possible. Rather than adding fewer vertices in each round, we can *prune* the

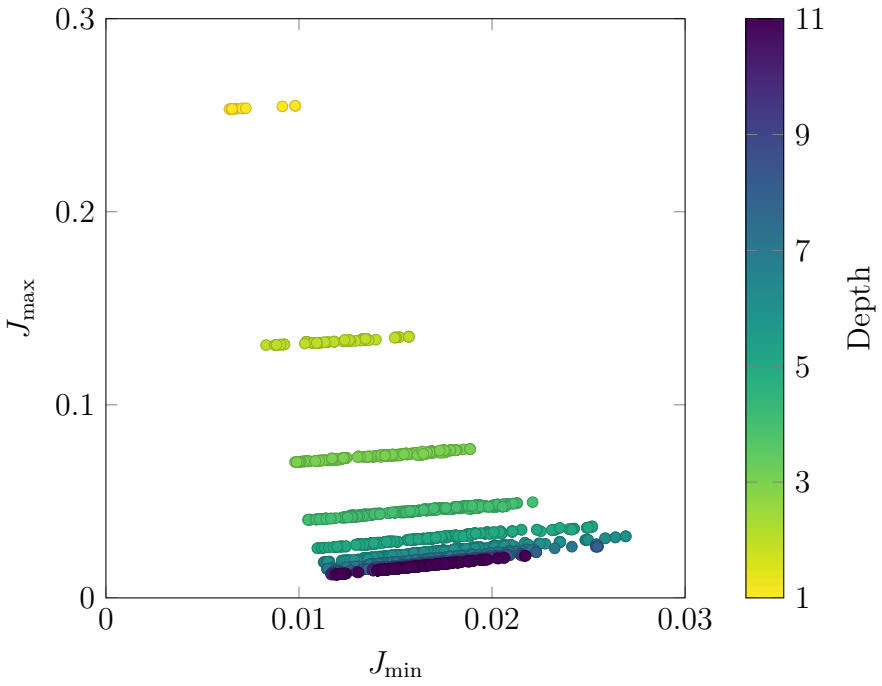


Figure 5.15: Upper and lower reward bounds for each candidate vertex considered when building the planning trees in Figure 5.14. At each depth of the tree, there is a strong positive correlation between the two rewards.

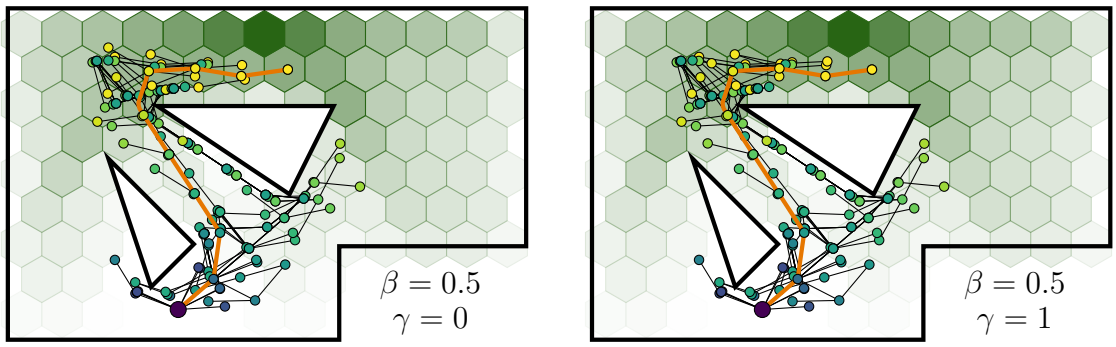


Figure 5.16: When each round of Algorithm 5.2 adds one vertex to each layer of the tree, the resulting tree is not very sensitive to γ. In this scenario, candidate vertices are only compared with other candidate vertices at the same depth and the rankings of candidate vertices in the same layer are similar if based on $J_{\min}$ or on $J_{\max}$. The tree obtained using $\gamma = 0$ which corresponds to $J_{\text{heur}} = J_{\min}$ (left) is almost identical to the tree obtained using $\gamma = 1$ which corresponds to $J_{\text{heur}} = J_{\max}$ (right).

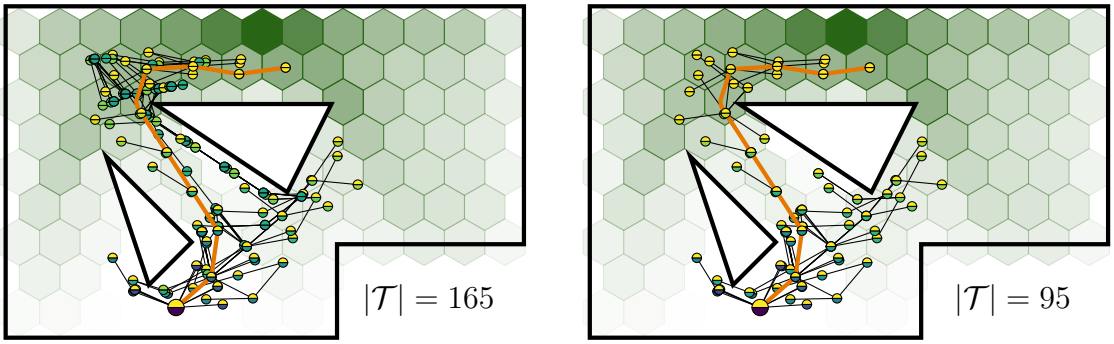


Figure 5.17: After several rounds of adding vertices, the tree (left) may be some vertices with $J_{\max}(v) < J_{\min}(v^*)$ for some other vertex v^*. Any path through these vertices is guaranteed to be worse than any path through v^* and so they can be removed from the tree. After removing these vertices, the pruned tree (right) is much smaller than the original tree. In this figure, the color of the top half of each vertex represents its $J_{\max}$ and the color of the bottom half represents its $J_{\min}$.

tree to remove poor quality vertices which were added in previous rounds. The two reward bounds, $J_{\min}(v)$ and $J_{\max}(v)$, of v are defined such that every path that begins by following the tree from its root to v has its reward between these two bounds. If a second vertex, v', has $J_{\max}(v') < J_{\min}(v)$ then every path through v' must be worse than any path through v. In this case, there is no point in considering paths through v'—none of them could possibly be the best path—so we might as well remove v from the tree. By pruning the tree to remove all such vertices, we can often remove close to half of the tree's vertices (Figure 5.17) which will make every future round of Algorithm 5.4 more efficient.

Pruning should be performed as often as necessary, but does not need to be performed every time a new vertex is added to the tree. In particular, it only needs to be performed when v^*—the vertex which maximizes $J_{\min}$—changes. When new vertices are added in Algorithm 5.4, they are compared with the existing v^* and it is updated if necessary. After a round of Algorithm 5.4, if v^* has been updated, then Algorithm 5.2 prunes the tree based on the new $J_{\min}(v^*)$. The actual process of pruning (Algorithm 5.6) is extremely simple. As every vertex of the tree has

$J_{\min} \leq J_{\min}(v^*)$, the algorithm simply compares each vertex with $J_{\min}(v^*)$ and removes every vertex whose reward is guaranteed to be less than this value.

Algorithm 5.6: Prune tree

Input: Planning tree, $\mathcal{T}$
Output: Planning tree, $\mathcal{T}$, with some vertices removed

```
for vertex v ∈ T do
    if J_max(v) < J_min(v*(T)) then
        Remove v from T
return T
```

5.7.3 Re-rooting the tree

As the tree grows, different numbers of vertices are considered at different depths. Algorithm 5.2 can only increase the maximum depth of the tree by 1 in each round, so it begins considering vertices at depth τ during the τ^{th} round. Therefore, at shallow depths, it will consider many different candidate vertices and so the first several vertices of the path are well optimized. The deep vertices, on the other hand, are not usually very optimized as fewer candidates are considered and the exponential decay of β^τ results in deep vertices contributing relatively little to the overall reward. As a result, the searcher's best strategy is to follow the optimized front of best path—from the search tree's root to v^*—and then replan instead of following the back of the best path, which is often not fully optimized.

Suppose the path from the root vertex, v_0, to the vertex with the highest reward, v^* is $p^* = (v_0, v_1, \ldots, v^*)$ and the searcher has now moved from its original original location, $q(v_0)$, to the first vertex, $q(v_1)$. At this point, if the searcher wants to replan, it could grow a new tree at $q(v_1)$ with initial belief equal to a normalized version of $\widehat{\boldsymbol{w}}(v_1)$. However, many vertices of the original tree are already children of v_1 and so it is more efficient to use these existing vertices in the new tree as they are still valid due to the recursive nature of the reward function J. The process of converting the existing tree into a new tree rooted at v_1 is called

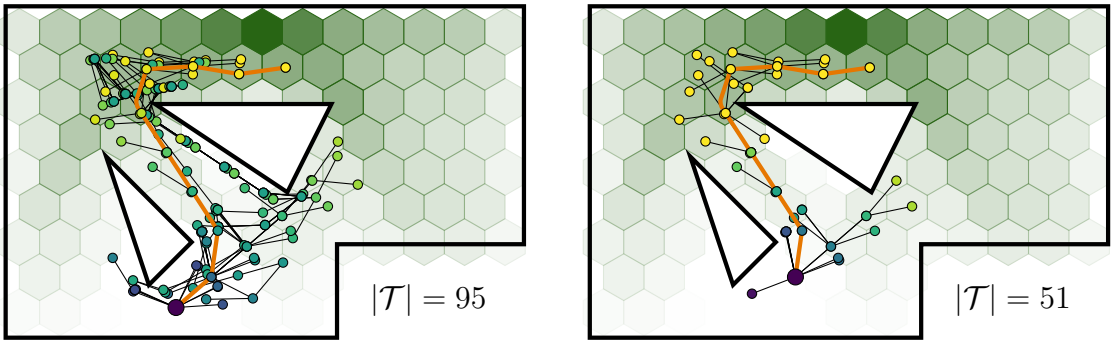


Figure 5.18: After a searcher has followed its search path to the first vertex of the tree, part of the original tree (left) can be used as a starting tree for planning a search path from the searcher's new location. This re-rooted tree (right) has the robot's new location as its root and contains all vertices of the original tree which are descendants of this new root vertex.

re-rooting. It can often be used to initialize a new tree with more than half of the original vertices (Figure 5.18). As any stopping criterion for Algorithm 5.2 is related to the number of vertices in the tree, this process can significant reduce the computation needed when replanning.

Each vertex's location and parent remains the same; however all their statistical properties will change as they must now conditioned on the fact that the searcher did not find the target when travelling from the old root, v_0, to the new root, v_0'. With this new condition, the vertex's cumulative probability becomes

$$\begin{aligned}\phi(v') &= \mathbb{P}\left(\text{Connected by } v \;\middle|\; \text{Not connected by } v_0'\right) \\ &= \frac{\mathbb{P}\left(\text{Connected by } v \text{ and not connected by } v_0'\right)}{\mathbb{P}\left(\text{Not connected by } v_0'\right)} \\ &= \frac{\phi(v) - \phi(v_0')}{1 - \phi(v_0')}\end{aligned}$$

where the v' denotes the vertex with properties computed for the re-rooted tree. Combining this transformation with (5.22), (5.25), and (5.28), the other statistical

properties get transformed according to

$$\begin{aligned}\Delta\phi(v') &= \frac{\Delta\phi(v)}{1-\phi(v'_0)} \\ J_{\min}(v') &= \frac{J_{\min}(v)-\phi(v'_0)}{\beta^{\tau(v'_0)}(1-\phi(v'_0)} \\ J_{\max}(v') &= \frac{J_{\max}(v)-\phi(v'_0)}{\beta^{\tau(v'_0)}(1-\phi(v'_0)}.\end{aligned}$$

Additionally, the vertex's new depth is $\tau(v') = \tau(v) - \tau(v'_0)$ and its belief should be renormalized so that it sums to $1 - \phi(v')$ instead of $1 - \phi(v)$. Using these rules, a vertex's properties can all be updated in constant time when the tree gets re-rooted (Algorithm 5.7).

Algorithm 5.7: Re-root vertex

Input: Vertex, v; and new root vertex, v'_0, for planning tree
Output: Vertex, v', with properties updated for re-rooted tree

1 $\tau(v') \leftarrow \tau(v) - \tau(v'_0)$ `/* Depth in tree */`
2 $\phi(v') \leftarrow \frac{\phi(v)-\phi(v'_0)}{1-\phi(v'_0)}$ `/* Total probability of success */`
3 $\Delta\phi(v') \leftarrow \frac{\Delta\phi(v)}{1-\phi(v'_0)}$ `/* Marginal probability of success */`
4 $J_{\min}(v') \leftarrow \frac{J_{\min}(v)-\phi(v'_0)}{\beta^{\tau(v'_0)}(1-\phi(v'_0)}$ `/* Reward lower bound */`
5 $J_{\max}(v') \leftarrow \frac{J_{\max}(v)-\phi(v'_0)}{\beta^{\tau(v'_0)}(1-\phi(v'_0)}$ `/* Reward upper bound */`
6 $\widehat{\boldsymbol{w}}(v') \leftarrow \frac{1-\phi(v)}{\mathbf{1}^\top\widehat{\boldsymbol{w}}(v)}\widehat{\boldsymbol{w}}(v)$ `/* Belief (not normalized) */`
7 **return** v'

When constructing the re-rooted tree (Algorithm 5.8), we first sort the vertices of the old tree by depth. By sorting them, every vertex is considered after its parent, so we can simply check if a vertex's parent is already in the re-rooted tree to determine if that vertex should be added. The new root vertex is always added first, and then the sorting guarantees that all of its descendants will also be added. When a vertex is added to the re-rooted tree, its properties are immediately updated to be correct based on the condition that the target was not found while travelling to the new root vertex. After every vertex has been added, the re-rooted tree resembles a tree that would be built by several rounds of Algorithm 5.4 without

needing to perform any difficult calculations.

Algorithm 5.8: Re-root tree

Input: Planning tree, $\mathcal{T}$; and new root vertex, v_0', already in $\mathcal{T}$
Output: Planning tree, $\mathcal{T}'$ rooted at v_0'

```
Create new tree T' with v_0' as root vertex          /* Algorithm 5.3 */
Sort vertices of T by depth
for vertex v ∈ T do
    if re-rooted version of parent(v) ∈ T' then
        v' ← re-rooted version of v with v_0 as root  /* Algorithm 5.7 */
        Add v' to T'
return T'
```

5.7.4 Planning trees for multiple searchers

Multiple searchers can improve their chances of finding a common target by coordinating their search paths. Typically this coordination requires the searchers to plan their paths so that they remain connected to each other at all times. Planning coordinated paths for multiple robots is challenging as the size of the search space scales exponentially with the number of robots. Sampling-based planning algorithms, however, are not hindered by this high dimensionality and are able to effectively plan coordinated paths [58].

Multiple coordinated search paths can also be planned using a search tree; however this tree's vertices consists of m locations—one for each searcher (Figure 5.19). In each round of adding candidate vertices, we sample one random location for each searcher and connect each of these m samples to the corresponding searcher's locations for each existing vertex of the tree. A new candidate vertex is obtained from one of the existing vertices by following the shortest paths from all of that vertex's locations towards their respective samples. This set of locations is a valid candidate vertex if they satisfy a communication constraint—either connectivity of the ad-hoc network formed by a team of searchers at those locations

or a sufficiently high probability of connectivity. The connectivity can be measured algebraically using the network's Fiedler eigenvalue—the second smallest eigenvalue of its Laplacian matrix—whose value is related to the overall strength of the possibly indirect communication between each pair of locations [59]. For an environment with a binary communication model, the network is connected if this eigenvalue is strictly positive; for a stochastic communication model, it has a high enough probability of connectivity if this eigenvalue is greater than a given threshold. Aside from vertices containing m locations which must satisfy this connectivity constraint, adding vertices works in exactly the same way as for a single searcher. Each vertex has a single value of ϕ which equals the probability that any of the searcher's will find the target when travelling from its root position to its position in that vertex. The single values of ϕ are used to define single values for $J_{\min}$ and $J_{\max}$ which are used when deciding which vertices to add to the tree.

Once sufficiently many vertices have been added to the tree, the best vertex is used to find the path from the root to the best vertex. This path maximizes the team's reward and is defined in terms of vertices of the tree which each consist of m locations. It can be converted into a set of m paths by connecting the corresponding locations in adjacent vertices of the planning tree. These paths are the optimal coordinated search strategy. They maximize the team's reward and ensure that the team of searchers maintains a connected network during the entire search. In general, the connectivity constraint forces the team to stay close together which limits the overall number of paths considered by the planner and reduces the burden of the high dimensional planning space.

5.8 Results

The search algorithm described in this chapter can be used to reconnect teams of robots who are performing a variety of tasks, such as search, coverage, surveillance,

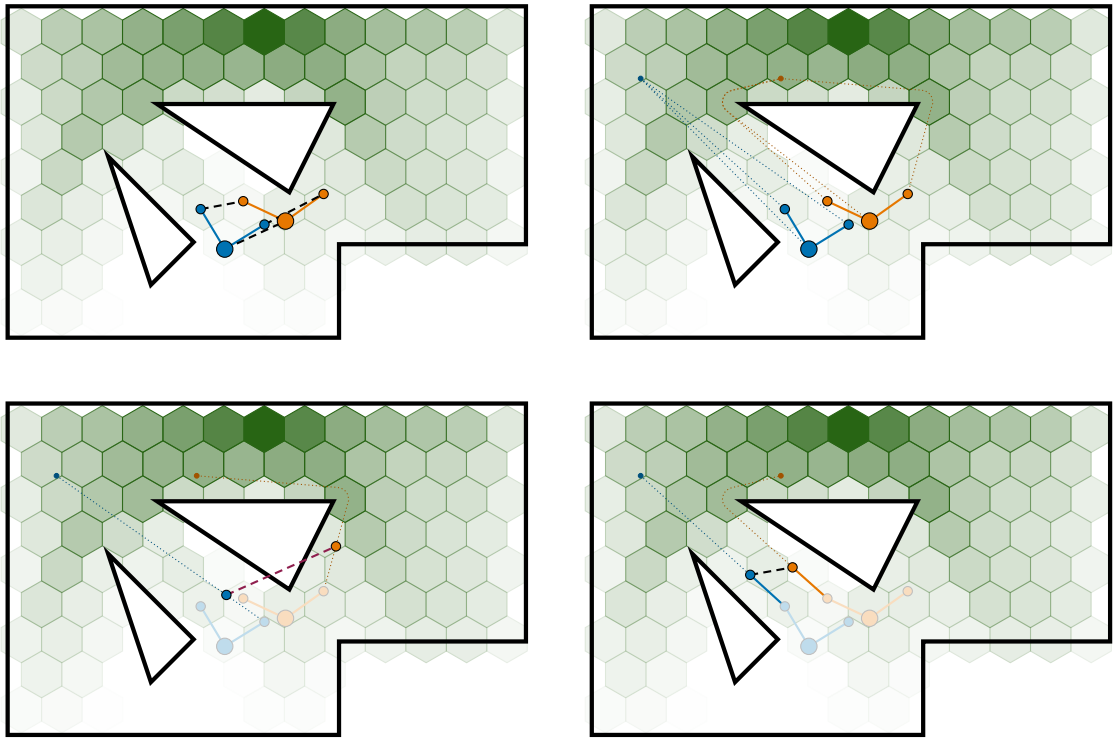


Figure 5.19: A planning tree for two cooperative searchers consists of pairs of connected searcher locations (top left). New candidate pairs are obtained by sampling the environment twice and connecting each location to all of one searcher's existing locations via shortest paths (top right). If the two locations of a candidate pair cannot communicate, or have too low of a communication probability, the pair cannot be added to the tree (bottom left). If the two locations of the pair can communicate, the pair can be added to the tree if it has a high enough reward (bottom right).

or delivery. The approach can be used in any application as long as the searcher has some knowledge—typically in the form of historic or simulated path data—of how its target behaves. To illustrate the effectiveness of this search algorithm, I applied it to a simple relay scenario consisting of two robots. These two robots in these simulations have different roles and thus different behavior.

The first robot is a wandering robot which travels throughout the environment by selecting a random target location and then following the shortest path from its current location to that location (Figure 5.20). Once it reaches its target location, it selects a new random target location and takes the shortest path to

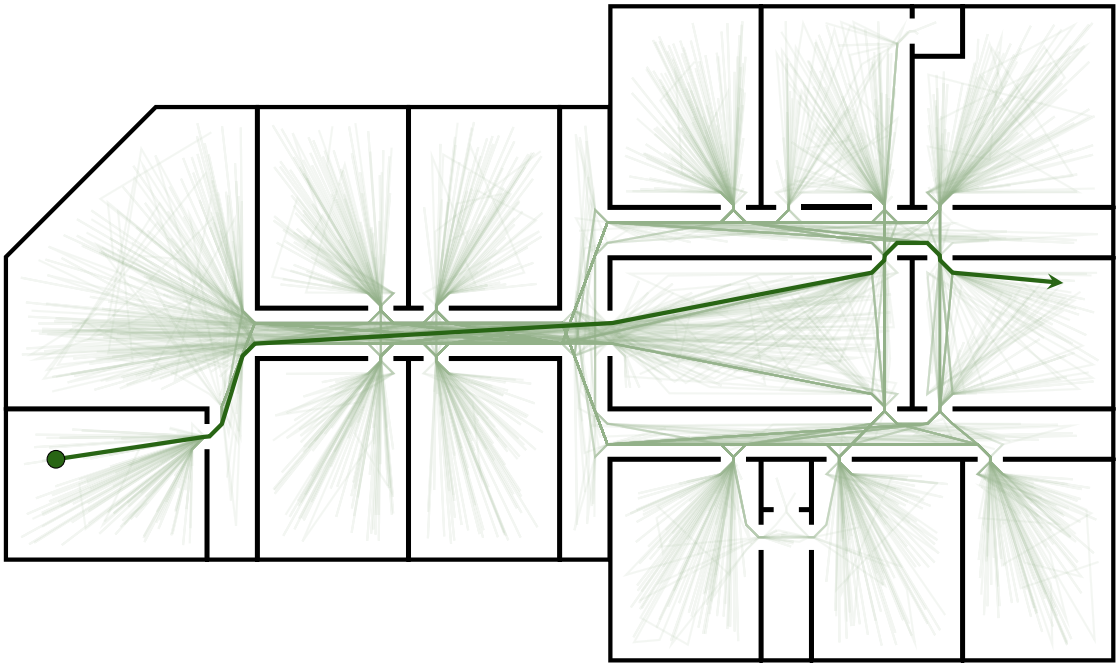


Figure 5.20: Example of a single simulated path for the wandering robot (dark green). It is overlayed on 2000 simulated paths (light green) which are semi-transparent to show which regions have high densities of paths and which regions have low densities.

it. Although this example may seem somewhat contrived, this type of behavior is actually quite common for a robot that has to perform many simple, spatially distributed tasks. If the tasks take relatively little time, such as in monitoring or delivery applications, the robot spends majority of its time travelling and likely takes the shortest route to its next task. The result of this kind of behaviour is that the robot is more likely to be found in some locations than others, even though the distribution of target locations is uniform. Depending on the environment, certain regions, such as hallways, will be part of many paths whereas remote locations, such as the corners of rooms, will only be visited rarely as the only paths through that location are paths starting or ending there.

The second robot serves as a relay between the wandering robot and a base station which does not move. Its main objective is to find the wandering robot so that it can receive some information from it. Once it receives this information, it

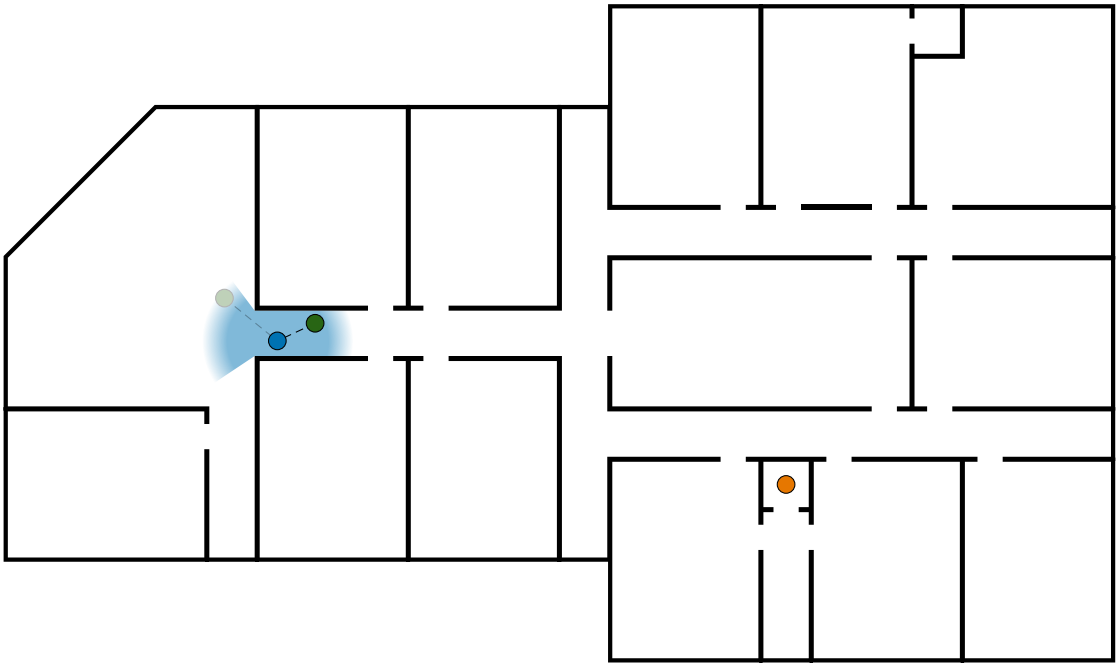


Figure 5.21: The search simulations are set in an environment where the searcher (blue) is guaranteed to find the wandering robot (green) if they are less 2 apart and have a clear line-of-sight. At distances between 2 and 3, their probability of communication decreases linearly from 1 to 0. The base station (orange) is located in a remote location which is rarely visited by the wandering robot.

travels back to the base station to transmit this information, after which it searches for the wandering robot again. Search for the wandering robot is successful when the two robots are within communication range. However when returning to the base station, the searcher is required to physically touch it before beginning a new search. For these simulations, I chose to put the base station in a remote location which the wandering robot rarely visits. This choice reduces the number of times that the search is trivial due to the wandering robot being visible from the base location. The communication model used in these simulations requires a line of sight. The probability of communication is 1 within a distance of 2 and then decreases linearly from 1 to 0 between the distances of 2 and 3. The location of the base station and communication range are shown in Figure 5.21.

As this scenario is cooperative, the searcher knows the other robot's general

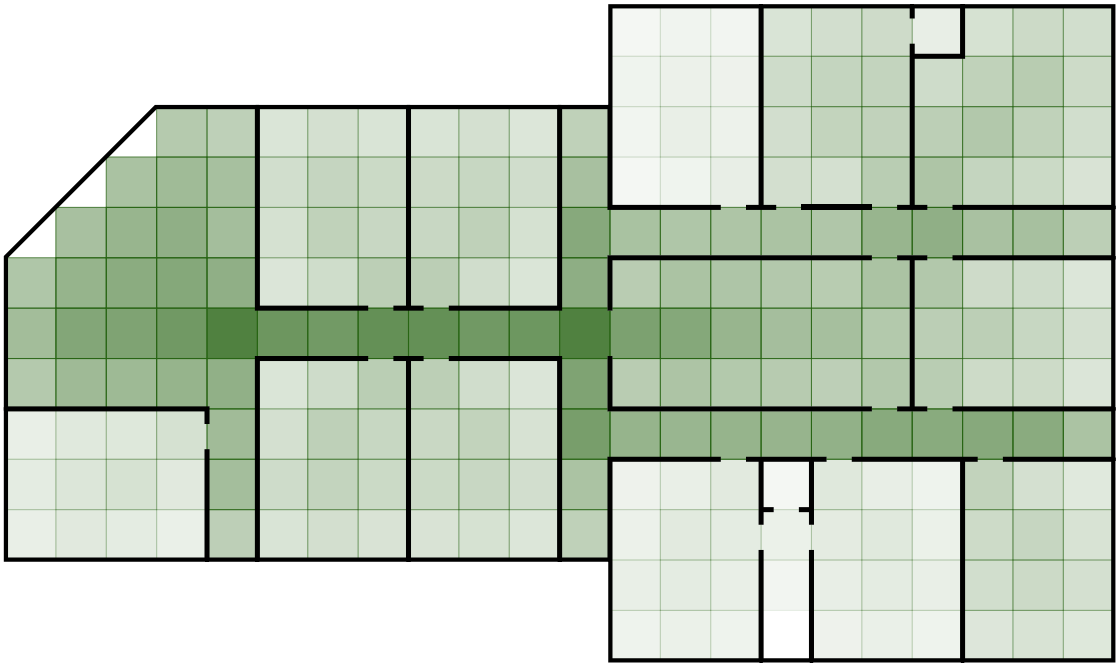


Figure 5.22: Stationary distribution of a HMM modeling the behavior of the wandering robot. Its transition probabilities were computed using 2000 simulated paths. The probability of each cell closely resembles the density of paths through that cell in Figure 5.20.

behavior—that it continually travels to randomly selected locations—however as the locations are selected randomly, it never knows the other robot's current target location. Using this knowledge, the searcher is able to simulate the target's behavior and use the simulated paths determine the transition probabilities in the HMM it uses to update its belief. The stationary distribution of this HMM (Figure 5.22) is quite similar to the actual density of target paths.

5.8.1 Comparison with other approaches

I compared my sampling-based search path planner with two reasonable baseline algorithms:

- **Random searcher**: This searcher chooses a random location and follows the shortest path to it. If it finds the target when following this path, search is successful; if not, it chooses a new random location and repeats until it

happens to find the target.

- **Go-to-mode searcher**: This searcher behaves greedily by following the shortest path from its current location to the location with the highest belief. As it takes very little effort to plan a path to the mode of the belief distribution, the searcher can replan its search path every time step.

For my algorithm and both of these benchmark algorithms, I ran a simulation where the searcher must find the target robot 500 times. The environment used for these simulations and its communication range and base station location are shown in Figure 5.21. The searcher and target both move at the same speed in all simulations. For each simulation, I recorded the number of time steps needed for the searcher to find the target after it leaves the base station and used these data to construct time-to-find distributions for each algorithm (Figure 5.23). My search algorithm had a mean time-to-find of 33.9 which was faster than both the go-to-mode (42.6) and random (48.1) searchers. I performed a Welch's t-test and found that the difference between all three means was statistically significant at the 95% confidence level.

Although the mean times-to-find indicate my algorithm performs better than either of the benchmarks, I noticed that the distributions are definitely not normal, so I decided to check some other statistics as well. I first checked the medians: 29 for my sampling-based searcher, 31.5 for the go-to-mode searcher and 33 for the random searcher. To my surprise, the medians were very close! Intrigued, I decided to check some other deciles and discovered that the first 4 deciles of all three algorithms are similar and my algorithm is only better for the higher deciles (Table 5.1). The same result can be observed in the percentile plots for each distribution (Figure 5.24) which show no significance between the algorithms below approximately the 45^{th} percentile with my algorithm being significantly better at higher percentiles.

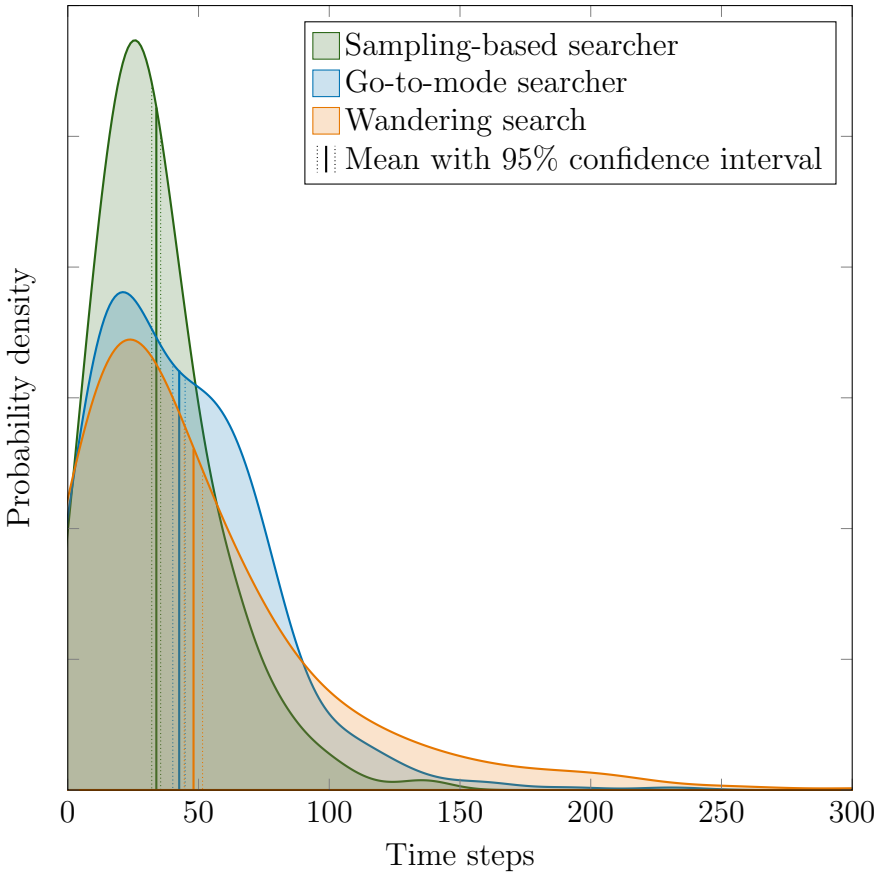


Figure 5.23: Time-to-find distributions for three search algorithms. Each distribution is based on 500 successful searches for a wandering robot in the environment shown in Figure 5.21 which moves at the same speed as the searcher. The smoothed distributions were obtained by kernel density estimation using a Gaussian kernel with bandwidth 0.4. The 95% confidence interval for the means of each distribution were computed by bootstrapping with 1000 resamplings of the data.

What can we conclude based on these quantiles? Is my algorithm *actually* better than the two simple benchmarks if the difference between them is not always significant? I was tempted to only report the distributions' means as they indicate my algorithm is better; however, doing so would be somewhat dishonest as the medians do not show nearly as significant of a difference and I had no justification as to why the mean would be a better statistic than the median for comparing the algorithms. After some reflection, I realized that the seemingly contradictory information displayed by the quantiles can be explained by dividing the data into

Table 5.1: Deciles, D_i, for the distributions shown in Figure 5.23. The probabilities are the p-values for a two-sided t-test that two different searcher's deciles are equal.

	Sampling-based	Go-to-mode		Random	
i	D_i^{sb}	D_i^{gtm}	$\mathbb{P}(D_i^{\text{sb}} = D_i^{\text{gtm}})$	D_i^{rand}	$\mathbb{P}(D_i^{\text{sb}} = D_i^{\text{rand}})$
1	8.47	7.19	0.418	7.41	0.457
2	15.46	14.26	0.542	13.68	0.172
3	20.14	21.31	0.468	17.68	0.061
4	25.20	27.56	0.334	26.09	0.688
5	29.74	37.34	**0.001**	33.42	**0.047**
6	34.12	48.18	**0.000**	43.14	**0.005**
7	41.34	56.35	**0.000**	56.33	**0.000**
8	48.56	66.69	**0.000**	72.37	**0.000**
9	63.54	78.26	**0.000**	107.32	**0.000**

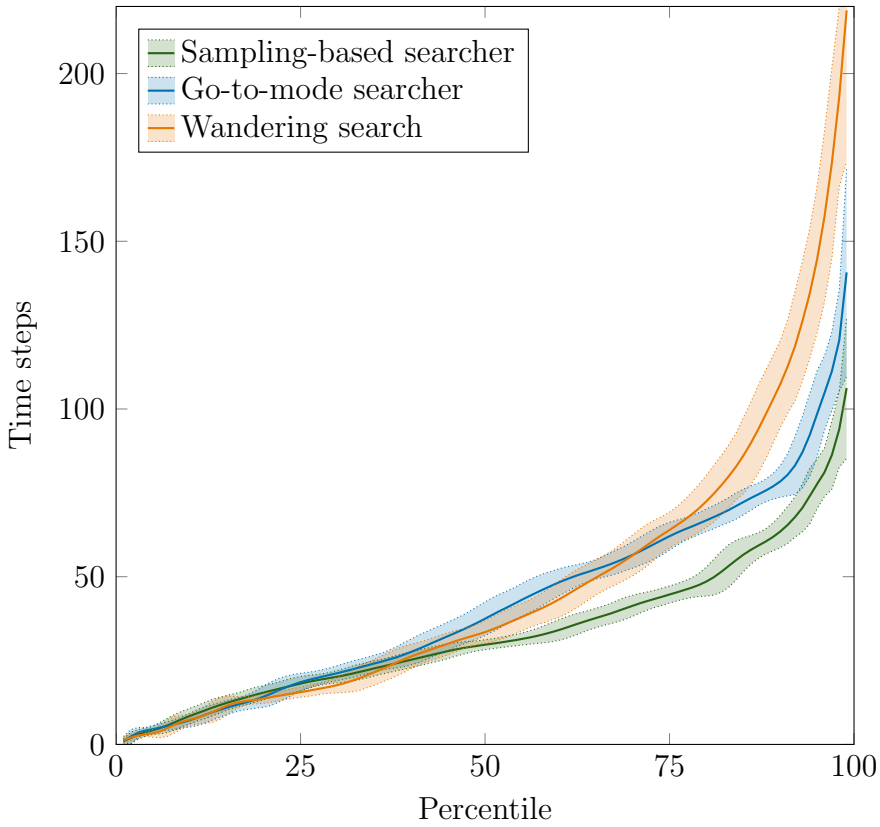


Figure 5.24: 95% confidence bands for the percentiles of the distributions shown in Figure 5.23. The percentiles and their confidence limits were computed by bootstrapping with 1000 resamplings of the data.

two categories:

- **Easy searches**: In these cases, the target robot happens to be relatively close to the base station. When the target is nearby, any searcher leaving the base station is likely to find it quite quickly, regardless of which path it takes, and so all searchers are likely to find the target in a similar amount of time.

- **Difficult searches**: In these cases, the target robot is far away from the base station. The searcher would need to follow a relatively long path, involving many turns, to reach the target even if it knew exactly where the target was and where it was going. As the searchers do not know where the target is (the belief is not necessarily accurate), there are many chances for the searcher to make a wrong turn increasing the time needed to find the target. To make matters worse, the longer it has been since the searcher last found the target, the less precise its belief is, making search more difficult.

Every individual search therefore has a difficulty which depends on the location of the target, the communication range of the searcher, and topology of the environment. For the environment used in these simulations, approximately 40% of the searches are easy. In any easy situation, any search algorithm can be used to quickly find the target and so we do not see a significant difference between any of the algorithms at the 4^{th} decile or below. The remaining 60% of searches are more difficult and these searches are really the scenarios that should be used when assessing a search algorithm. The fact that the 5^{th} and higher deciles are smaller for my algorithm means that it is indeed outperforming the other algorithms in difficult searches. The high percentiles can also be used as a statistical performance guarantee: there is a 95% chance that my algorithm will find the target within 77 time steps, whereas the go-to-mode and random searchers require 85 and 123 time steps to make the same guarantee. These results illustrate that the high

quantiles are actually better statistics for comparing search algorithms, and they show my sampling-based search algorithm performing significantly better than the other two algorithms.

5.8.2 Effect of discount factor

I also compared the effect of discount factors on the performance of a searcher using my algorithm. Recall that the discount factor, $\beta \in [0, 1]$, determines the relative reward of finding the target after different lengths of time according to (5.25). A small value of β prioritizes paths which have a high probability of finding the target immediately even if it is unlikely to find the target later on the path. A large value of β prioritizes paths which have a high probability of finding the target eventually but does not distinguish between paths which find it quickly and paths which find it slowly.

Intuitively, I expected some intermediate value of β—rewarding paths which have a high probability of finding the target eventually but are also likely to find it quickly—would give the best performance, measured based on the distributions of times needed to find the target. To test this hypothesis, and determine which value of β performs best, I ran simulations for 7 evenly spaced values of β ranging from 0 to 1 where the searcher was required to find the target 500 times (Figure 5.25). To my surprise, I found that the choice of β had very little effect on the distributions of search times! When comparing the means of these distributions, representing the average time needed to find the target, I found no significant difference between the means when $\beta \in [1/3, 1]$ at the 95% confidence level (Table 5.2). For $\beta < 1/3$, there was a statistically significant increase in the mean search time as β decreased, with the $\beta = 1/6$ searcher's performance similar to the go-to-mode searcher and the extreme case of the $\beta = 0$ searcher similar to the random searcher.

I also compared the quantiles of the search time distributions for different values of β (Figure 5.26). The quantiles show that the searcher with $\beta = 0$ performs

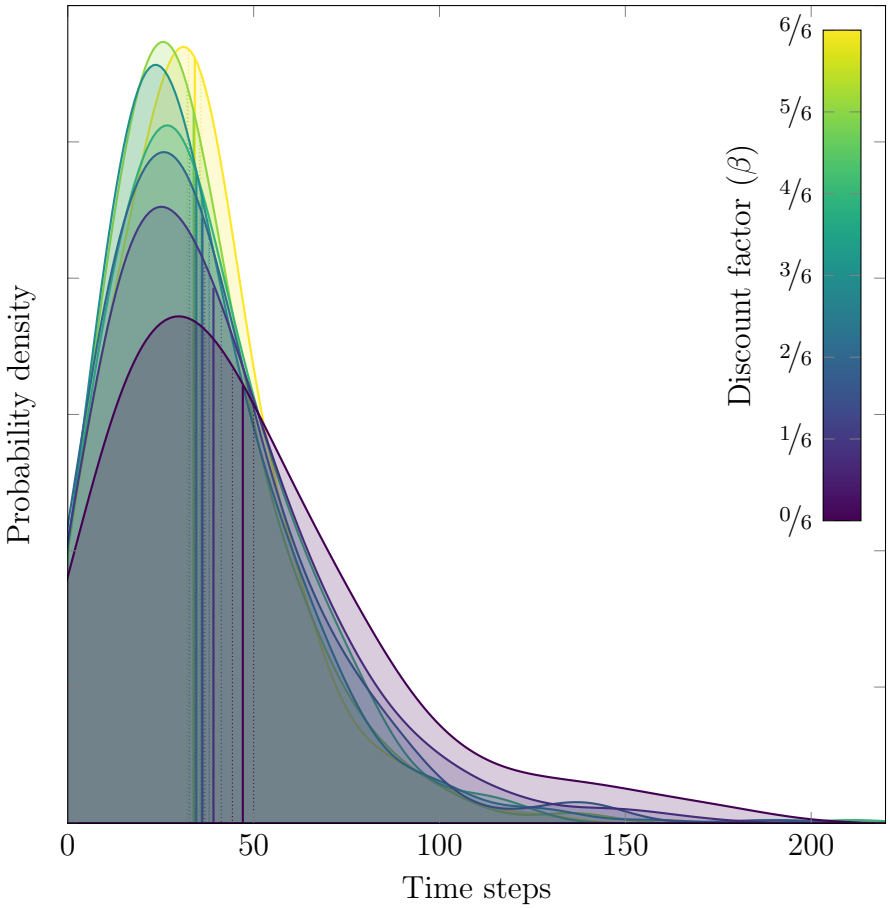


Figure 5.25: Time-to-find distributions for the sampling-based search algorithm with 7 different values of the discount factor, β. Each distribution is based on 500 successful searches for a wandering robot in the environment shown in Figure 5.21 which moves at the same speed as the searcher. The smoothed distributions were obtained by kernel density estimation using a Gaussian kernel with bandwidth 0.4. The 95% confidence interval for the means of each distribution were computed by bootstrapping with 1000 resamplings of the data.

much worse than the other searchers, whereas the performance of the searchers with $\beta \in [1/2, 1]$ are statistically indistinguishable as none of their quantiles are significantly different. This lack of sensitivity to the exact value of β indicates that this search algorithm will be quite easy to use in practice as any value of $\beta \geq 1/2$ gives essentially the same performance: they are all significantly better than the alternatives of the go-to-mode and random searchers.

For the simulations in this chapter, the simulated robots had average velocities

Table 5.2: Effect of the discount factor, β, on the means of the time-to-find distributions in Figure 5.25 for sampling-based searchers. These mean times, $\overline{\tau}^{\mathrm{sb}}_{\beta}$, are compared with the mean times for the go-to-mode searcher, $\overline{\tau}^{\mathrm{gtm}}$, and random searcher, $\overline{\tau}^{\mathrm{rand}}$. The p-values were computed using a two-sided t-test for the hypothesis that the mean is equal to the mean for one of the searchers in Figure 5.23.

β	$\overline{\tau}^{\mathrm{sb}}_{\beta}$	$\mathbb{P}(\overline{\tau}^{\mathrm{sb}}_{\beta} = \overline{\tau}^{\mathrm{sb}}_{5/6})$	$\mathbb{P}(\overline{\tau}^{\mathrm{sb}}_{\beta} = \overline{\tau}^{\mathrm{gtm}})$	$\mathbb{P}(\overline{\tau}^{\mathrm{sb}}_{\beta} = \overline{\tau}^{\mathrm{rand}})$
1	34.170	0.777	**0.000**	**0.000**
$5/6$	33.860	1.000	**0.000**	**0.000**
$4/6$	36.046	0.158	**0.000**	**0.000**
$3/6$	34.592	0.636	**0.000**	**0.000**
$2/6$	36.216	0.115	**0.001**	**0.000**
$1/6$	39.192	**0.002**	0.071	**0.000**
0	47.089	**0.000**	**0.042**	0.712

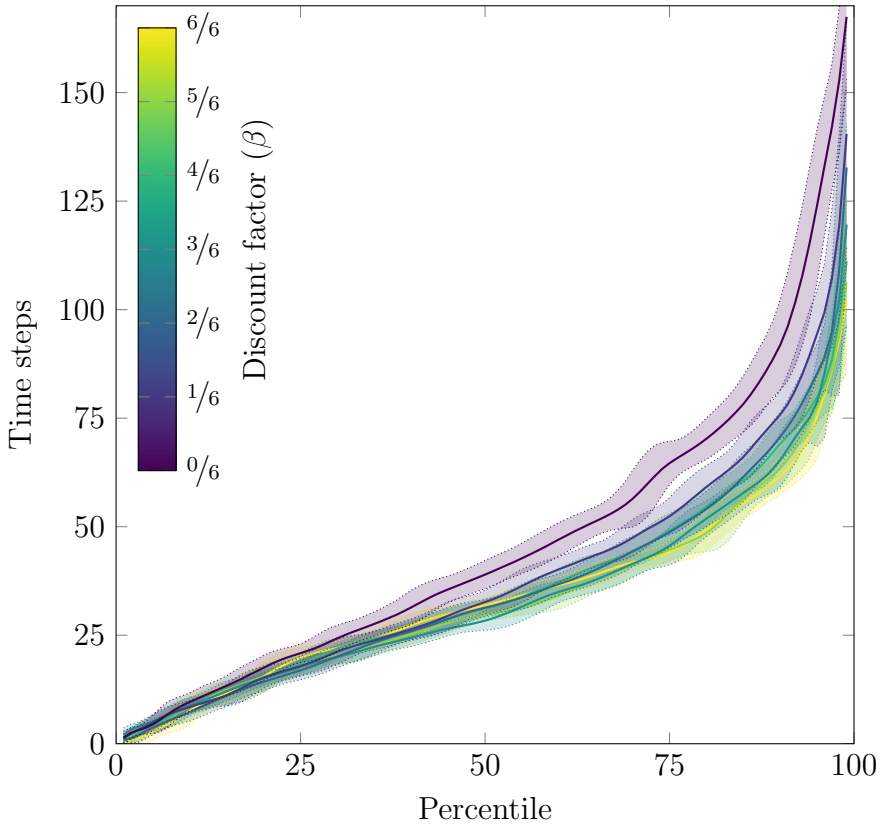


Figure 5.26: 95% confidence bands for the percentiles of the distributions shown in Figure 5.25. The percentiles and their confidence limits were computed by bootstrapping with 1000 resamplings of the data.

of 1 m/s and average turning speeds of 45 °/s. The environment had a length of 44 m and width of 26 m. The total time needed for the searcher to find the target and return to the base station 500 times ranged from 6.8 h to 8.1 h. The simulations were performed in C++ using a standard laptop computer running Linux and took between 2.4 h to 3.5 h to perform. As the simulation times, which include the time needed to plan the search paths, were lower than the real time the robot would require to follow the planned search paths, my sampling-based search planner could be used in real time on a robot with hardware comparable to a standard laptop computer.

5.9 Conclusions

When robots cannot communicate over long ranges, a team of robots may need to split up into multiple smaller disconnected teams while completing their tasks. If the tasks take variable lengths of time, it can be difficult to plan a rendezvous time and place when they separate. Instead, they can simply search for each other when they have information to share and need to communicate. In this chapter, I presented an algorithm that disconnected robots can use to find each other without making an explicit plan for reconnection. This algorithm is based on a *belief* of the target's behavior and location and paths are planned using an *sampling-based planner* which maximizes a discounted reward function.

Each robot maintains a probabilistic belief about all of the disconnected robots in the team. This belief is updated using a hidden Markov model, which is built using historic or simulated data about the teammate's behavior. This HMM is based on a polygonal lattice that covers the environment. Its observable states are the cells of the lattice; its hidden states include:

- **Direction states** which represent a 2-dimensional pose consisting of a lattice cell and lattice direction; and

- **Transit states** which model the variable transit times between direction states due to the different turning angles and linear direction between states as well as the inherent variability in the target robot's velocity.

The searcher uses both positive and negative observations alongside the HMM to update its belief of the target's state and physical location.

Using the belief, the searcher plans a path which maximizes a discounted reward function. This discounted reward uses a discount factor $\beta \in [0, 1]$ to give higher weight to finding the target quickly while also rewarding paths that find the target eventually. Search paths are obtained by building a tree of possible search paths. New vertices are added to this tree by sampling a random point in the environment and adding one new vertex at each layer of the tree in the direction of this randomly sampled point. Old vertices are removed from the tree whenever the upper bound on its reward is lower than the lower bound on the reward of a recently added vertex, which guarantees that the best path cannot be through that vertex. The searcher follows the first several vertices of the highest reward path in this tree to search for the target. If it does not find the target, it can *re-root* the search tree so that its current location is the root of the tree and any existing vertices of the tree which are descendants of this new root are maintained to reduce the computation needed to build a new planning tree.

I compared this sampling-based search algorithm with two benchmark algorithms: one where the searcher follows the shortest path to the mode of its belief distribution, and a second where the searcher follows the shortest path to a random location. My approach had the best mean search time after completing 500 search attempts. The quantiles of the search time distributions indicated that the three algorithms had equivalent performance for the fastest 40% of the searches, which occur when the target starts near the searcher and is easy to find. In the remaining 60% of searches, the target is more difficult to find as it is initially further away and in these scenarios my algorithm is significantly faster than both benchmarks.

I also evaluated the effect of the discount factor on the performance of the searcher and found the best performance when $\beta \in [0.5, 1]$, however the performance is not very sensitive to the discount factor. All of my simulations were performed faster than real-time indicating that this search algorithm could be implemented on a real robot.

Chapter 6

Robust multirobot coverage

Coverage in real environments is much more difficult than in the idealized setting I presented in Chapter 4. Robots don't always behave as expected, maps may be incorrect, and humans or animals may interfere with the robot. These challenges can prevent a robot from executing its plans as expected or may mean that the plan is insufficient for properly completing the coverage task. This chapter focuses on several ways to make coverage more robust—primarily through semantic commands, replanning, and searching for teammates—in both single- and multirobot settings.

The methods in this chapter all use feedback to reduce uncertainty, albeit often in an indirect way. The feedback comes in the from of data from the robot's sensors—often including cameras, lidar, wheel encoders, and contact sensors—and potentially information shared by another cooperative robot over a wireless network. The robot uses these data in a simultaneous localization and mapping (SLAM) system to follow a planned path without usually needing to provide direct feedback to the high level planner while executing plans. It also uses these data to determine when to replan because a previous plan is no longer valid, and to determine when to change between different modes of behavior such as coverage and search. Due to the many processes running simultaneously on a robot—SLAM, low-level path following, high-level coverage planning, belief estimation, communication—the high level planning components typically respond indirectly to feedback via information received from other concurrent processes.

6.1 Related work

Coverage robots have been the most successful consumer robots, representing approximately 60% of that market [164]. Their success has been largely due to their ability to work adequately in a wide variety of environments, despite usually being inefficient. This robustness has generally been achieved through a *lack* of planning. Rather than follow an exact path, the robots simply follow pre-programmed behaviors and change their behavior in response to interactions with the environment, such as bumping into an obstacle. The simplicity of these behaviors has enabled them to operate in new environments without even requiring a map. However, it also makes it difficult to guarantee complete coverage and often results in lots of duplicate coverage [161].

Common behaviors used by commercially available coverage robots include:

- **Random bounce**: the robot travels in a straight line and turns at a random angle when it bumps into an obstacle. This behavior was used by the early versions of the iRobot Roomba [91] and can potentially achieve full coverage, given enough time, in any environment.

- **Spiral**: the robot follows a spiral path with a larger radius on each pass. The iRobot Roomba's spot-cleaning mode spirals around a point [91], whereas the John Deere robotic mower spirals around landmarks and switches to a new spiral when it bumps into obstacles [9].

- **Parallel ranks**: the robot follows series of parallel straight lines—called *ranks*—turning when it reaches an obstacle. This behavior, also called *serpentine*, is used by many robots including vacuum cleaners [70, 182], lawnmowers [152], and mops [200].

- **Object following**: the robot follows the perimeter of an obstacle, using its sensors to ensure that it remains close to the obstacle as it moves around it.

> Many different coverage robots [17, 91, 182, 200] use this behavior as it is necessary to ensure good quality coverage near the edges of environments.

Additionally, some robots have *escape behaviors* which are not intended for coverage but are nevertheless used during a coverage mission [121, 192]. These behaviors help the robot escape a dangerous situation such as getting stuck on some rough terrain or near the edge of a cliff

The built-in behaviors of robots are additionally specific to that robot's mechanical constraints. One large robotic lawnmower uses a variant of the parallel rank behavior where it follows offset parallel loops to accommodate its large turning radius [165]. A robotic mop moves back-and-forth along short curved paths on either side a straight line while spraying water or a cleaning solution to ensure that it properly cleans on both sides of the line [50]. A triangular robotic vacuum cleaner has specialized behavior in concave corners, enabling it to clean right into the corner [171].

Robot lawnmowers cannot typically rely on bumping into obstacles because many edges of lawns do not consist of physical obstacles, but instead are simply a transition from grass to a garden. Some lawnmowers instead require the installation of a boundary wire or series of posts which emit a signal to tell the robot when it has reached the boundary [152]. Other lawnmowers use a GPS system for localization and require user input through a mobile device to demarcate the edge of mowing region [17].

Many coverage robots are now able to create a map of their environment as they perform coverage using onboard cameras and sensors [142]. These maps are often used simply to determine when coverage is complete [182] and for providing information to a user [10]. They have additionally been used in a limited capacity in planning to enable room-by-room coverage [106]. At the start of a mission, the previously constructed map is partitioned into smaller components via a watershed algorithm [105] which each represent one room of a house. The robot

then covers each room via its pre-programmed behaviors before moving onto the next room. Although this strategy makes the robot appear more intelligent, it still relies on its original simplistic behaviors—not an optimal plan—to cover each room. One difficulty of using maps for planning is that robot maps are noisy and are constructed of several small local maps which must be aligned to create the global map [124]. If these local maps are slightly misaligned or too noisy, a robot following a path planned using the incorrect map may end up trying to travel through a wall! Additionally, robots have difficulty following an exact path due to wheel slip or uneven terrain and must rely on landmarks recognized by their camera to improve localization and return to the desired path [175]. As a result of this fragility of exact plans, even the most successful high-end robots do not use exact coverage plans.

Currently, cooperative multirobot coverage does not exist yet in any consumer product. The only case I am aware of coordinated coverage robots is the *teaming* of iRobot's vacuum and mopping robots [61]. In this situation, the robots share a map but the work is divided based on floor type—the mopping robot cleans tiled floor and the vacuum cleans carpeted floor—and the robots clean consecutively rather than concurrently. As consumer coverage robots are generally considered part of the *smart home*, they will likely be coordinated via the cloud [63] which requires a strong wireless connection and ensures constant communication at all times. The cloud also provides the benefit of more powerful computing resources than would be available onboard a robot, making coordinated planning more feasible. In situations where robots do not have constant communication, mobile robots may be used as communication relays between teammates [20, 126]. This role could be filled by lower cost robots without the capability of coverage, or by a coverage robot that can no longer cover—due to a damaged tool, or full dustbin—but is nevertheless still valuable to the team.

6.2 Sources of unpredictability

Contrary to the idealized scenarios regularly considered by academics, such as the one in Chapter 4, the real world is dynamic and has many sources of unpredictability. Both the environment and the robots may differ from the simplified models used for planning. These differences may make the plan difficult to execute as it does not match up with reality, but a successful robust coverage strategy must be able to respond to any of these sources of unpredictability.

Certain forms of feedback can be used to limit the effects of uncertainty without needing to explicitly identify the source of this uncertainty or its exact magnitude. In robotics, low level control systems typically use feedback in this manner. For example, a controller adjusting the voltage to a motor in response to measurements made by a wheel encoder, doesn't need to know how the terrain type, mass carried by the robot, or other factors affect its velocity to effectively make the robot travel forwards at a desired velocity.

Higher level robotics tasks, such as coverage or search, typically depend on many different processes which are affected differently by different sources of unpredictability. Therefore, it is helpful to use data from multiple sensors to distinguish between different sources of uncertainty and respond differently depending on the source. For example, suppose a robot was trying to enter a room when it bumped into something near the doorway. It would be helpful to use the robot's camera to determine if the door is closed and the robot needs to replan or if the robot bumped into the wall beside the door and just needs to relocalize. Simply using direct feedback, the high level process would not treat these situations differently. Instead, it can be more effective at its overall mission by understanding different sources of unpredictability and responding differently to each one.

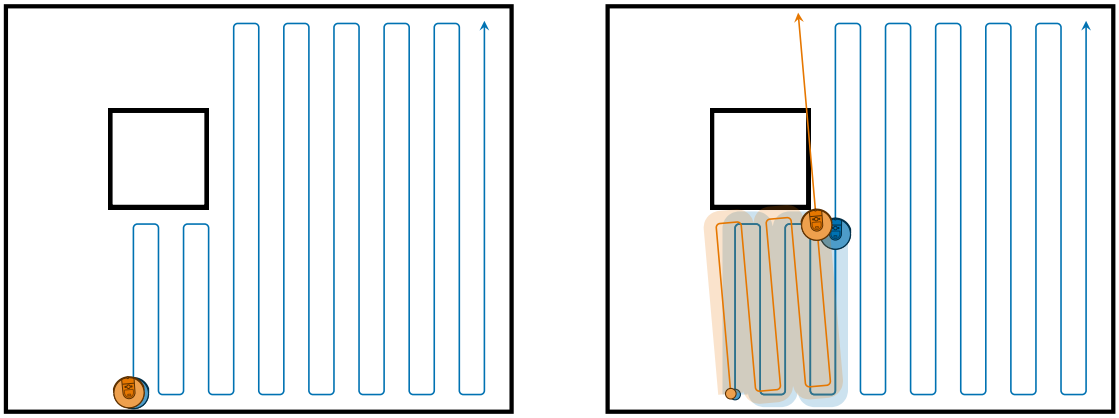


Figure 6.1: Localization errors occur when a robot's belief of its position (blue) is different from its true position (orange). When a robot plans based on its incorrect belief (left), its real path is a shifted and rotated version of its planned path which often results in collisions with obstacles preventing the robot from completing its planned path (right).

6.2.1 Mapping and localization errors

Consumers expect their robots to work immediately out of the box without any setup, but each robot operates in a different environment—no two homes or cities are identical. Instead of giving robots maps, which would require significant setup, coverage robots are equipped with simultaneous localization and mapping (SLAM) systems. They use these systems to create their own maps using a combination of sensor data from sensors such as wheel encoders, bumpers, cameras, and lidar [54]. These maps tend to be noisy and localization within them is difficult, especially if the lighting conditions are different from when the map was made [104]. Small errors in localization, especially with respect to the robot's heading, can result in coverage plans that tell the robot to travel through obstacles (Figure 6.1).

6.2.2 Environment changes

Aside from small errors due to the SLAM system, there can be large systematic errors in the map due to changes in the real environment. If a door was opened when the robot made its map but is closed when the robot is using the map, the

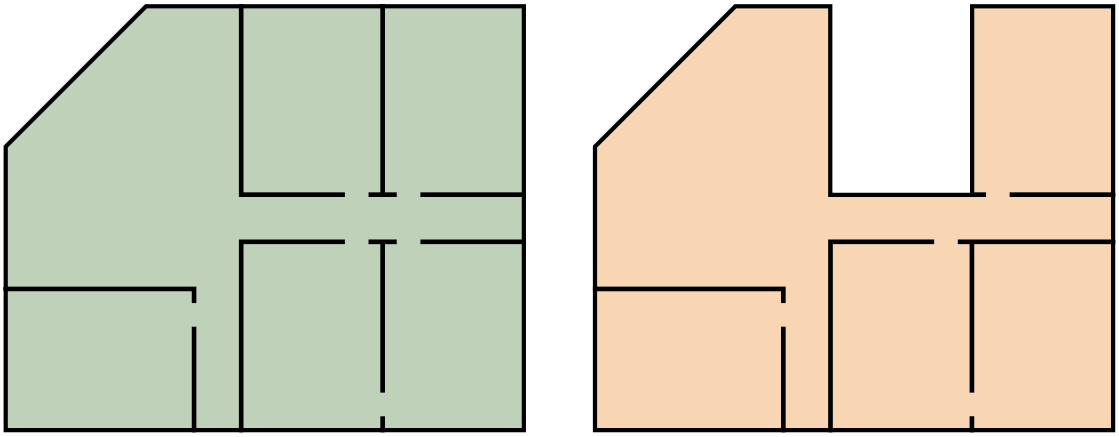


Figure 6.2: A robot's map depends on whether or not doors are open (left) or closed (right). Closing doors can make some portions of the map inaccessible or can change the topology of the free space.the free space.

robot may be unable to enter a large portion of its former environment, or the topology of the environment may change (Figure 6.2). Similarly, if furniture gets rearranged, the accessible floor in a room can change drastically. A robot must be able to distinguish these actual changes in the environment from small mapping errors and adapt its behavior to the current environment.

6.2.3 Interactions with humans

The most successful coverage robots are consumer robots, and they will naturally come into contact with consumers (or their pets). Humans commonly interact with coverage robots when either

1. The robot bumps into the human; or

2. The human picks up the robot and carries it to some other location

In the first case, the human essentially behaves as an obstacle which the robot can treat similarly to another new obstacle. The second case, commonly called *kidnapping*, is a problem which happens quite often for robotic vacuum cleaners [104]. Often humans kidnap their robots when the robot is cleaning a part of the

house that the human doesn't want cleaned. After its release, the robot must be able to relocalize and modify its behavior so that it doesn't annoy the human by returning to the same spot they just took it away from.

6.2.4 Battery or capacity constraints

The batteries that power most robots have limited capacities. Ideally, the robot would always start its mission with a full battery and the battery's capacity would be enough to complete the entire mission. In reality, robots are sold at the lowest price point possible so batteries are often small and the robot may need to recharge mid-mission.

An equivalent problem can happen with the robot's storage capacity. Robotic vacuums collect dust and debris in a bin which has a limited capacity; robotic mops have a limited capacity of cleaning solution to spray on the floor. When the vacuum's bin is full or the mop's cleaning solution is empty, they can no longer cover. Although human action is typically required in these cases, some robotic vacuums now have an evacuation station combined with their charging station [139]. For these robots, a fully bin is essentially the same as a low battery—the robot must return to a specific location to empty the bin before it can continue coverage.

6.2.5 Damaged robot

Hazards in the environment can damage a robot. Sticks get caught in the wheels of robotic lawnmowers and rocks can damage their blades. Wires and tissues laying on consumers' floors regularly get caught in the roller brushes or wheels of robotic vacuum cleaners [104]. It is also quite common for a robot to get itself wedged under a low overhang and be unable to free itself even when using escape behaviors. These, and other, hazards can prevent a robot from moving or can prevent its tool from working properly. For a single robot working independently,

both problems will prevent the robot from finishing its mission, requiring human intervention before it can continue. For robots operating within a team, however, the robot can still be productive member of the team, as a communication relay, whether mobile with a damaged tool or stationary.

6.2.6 Velocity

A robot's velocity is unpredictable. Robots are regularly required to cover regions with different terrains which impact their velocity—robotic lawnmowers take longer to mow longer grass. Velocity can also be affected by other hazards such as a cat sitting on top of a robot or something getting stuck in a wheel. For single robots, the robot's velocity does not actually affect its optimal coverage strategy, even if the robot is faster in some regions than in others (assuming velocity does not depend on direction of coverage).

For teams of robots, on the other hand, velocity is important. The minmax objective in multirobot coverage results in each robot having balanced workloads in terms of time. If one robot is faster than expected, this robot will finish before the others and the workloads will no longer be balanced. Similarly, if part of the environment has terrain that slows robots down, the robot assigned that region will take longer than expected and the workloads will not be balanced.

6.2.7 Changes in team size

The size of a robotic team can change mid-mission. If a robot's battery runs out or it gets stuck, it can no longer perform coverage and the team shrinks by one member. The other robots will need to redistribute the workload of this robot so that the team still covers everywhere. If this robot later has finished recharging or a human has freed it, it can resuming covering and the team will need to redistribute work again to take advantage of this "new" team member.

6.3 Semantic commands

Localization errors make it difficult to tell a robot to go to a precise coordinate. If this coordinate is near an obstacle, a small localization error may cause the robot to try penetrating the obstacle in an attempt to reach the coordinate. As many coverage plans regularly involve coordinates along walls—or worse, in corners—this problem makes it impossible to accurately execute a full coverage plan whenever there are localization errors. Instead, the robot should be issued a *semantic command* which is a high-level specification of what the robot should do, rather than a coordinate-by-coordinate description.

Room-by-room coverage is an example of a sequence of semantic commands. In this scenario, the map is first partitioned into a set of smaller regions representing the rooms of house. This partition can be obtained manually, via user interaction with a smartphone app, or could be computed automatically using a watershed algorithm [106]. Once the coverage region has been partitioned into rooms, the robot is given semantic commands like "clean the kitchen" or "clean the bathroom". Lower level behaviors are responsible for executing these semantic commands, potentially using simpler semantic commands like "go to the kitchen" and "clean the kitchen's perimeter" which may themselves consist of several still simpler semantic commands. Each command is successfully completed once all of its component commands have been successfully completed using the corresponding behavior. Completion of the simplest semantic commands is determined directly from sensor data: the command "drive straight until you bump into an obstacle" is complete when the bumper gets pressed in by an obstacle. Using the semantic commands of room-by-room coverage not only makes the robot's behavior much more robust, but also makes the robot appear more intelligent to the user, especially as the current semantic behavior of the robot can be displayed in a smartphone app [91].

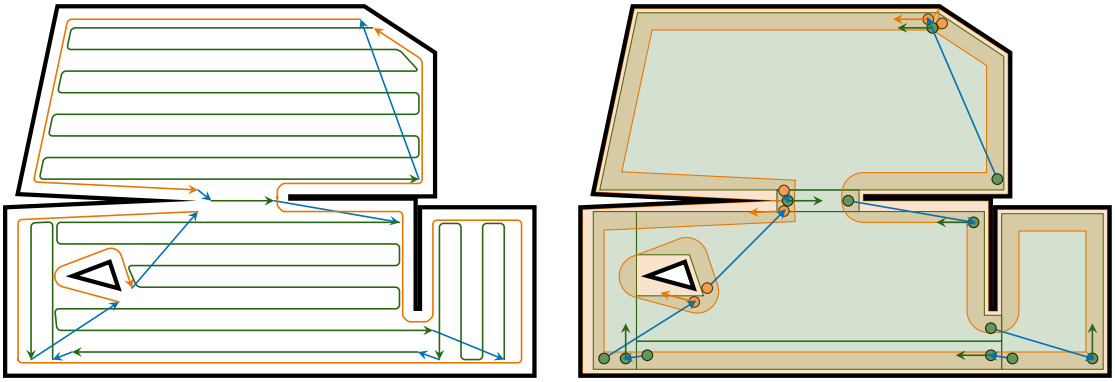


Figure 6.3: Conversion of a coverage path (left) into a sequence of semantic behaviors (right). The robot performs coverage by completing the go-to (blue), interior coverage (green), and perimeter coverage (orange) commands but is not required to follow the original coverage path exactly.

The coverage strategy of Chapter 4 consists of three kinds of behavior:

1. Coverage along the perimeter by following a sequence of consecutive perimeter ranks;

2. Coverage of an interior region by a sequence of adjacent antiparallel interior ranks; and

3. Efficient motion from the end of one coverage region to the start of another coverage region.

The plans produced by the coverage planner are paths which consist of a sequence of waypoints describing the exact locations the robot must visit, assuming no sources of uncertainty. This path can instead be converted into a sequence of semantic commands alternating between go-to commands and coverage commands (Figure 6.3), telling the robot what to do without specifying exactly how. The how of each semantic command is determined in real time by the robot's behaviors which rely on real-time sensor data in addition to the description of the command.

6.3.1 Go-to commands

The simplest semantic commands are the *go-to* commands. These commands are used when the robot moves to the start of a new coverage region. The start locations of perimeter coverage regions are always either along a wall or in a corner. The start locations of interior coverage regions are often along a edges or corners of obstacle, but may also be at edges or corners defined by previously covered regions. As these points are near obstacles, localization errors often result in the target point being slightly inside the obstacle and thus inaccessible. If the localization error is in the opposite direction, the target point may be a small distance away from the obstacle. Regardless of the direction of localization error, the correct target point should be wherever the wall or corner is, not at the exact coordinate. Therefore the robot receives a semantic command of "go to the corner near q" or "go to the edge of the previously covered region near q" and uses its sensors to determine when it is in the correct location.

When told to "go to the corner near q", the robot is successful when it is (a) near the target point and (b) in a corner (Figure 6.4). To satisfy both objectives, the robot uses a simple behavior (Algorithm 6.1). First, it follows a *safe* path—far enough from obstacles to avoid collisions despite localization errors—from its current position to the target location using its current map. This path also continues past the target point so that the robot will bump into a wall even if the target point is further from the wall than expected. By following this path, the robot is guaranteed to bump into one of the two walls forming the corner near the target point. Once it bumps into the first wall, it makes a shallow turn and follows along the wall. This direction will lead it to the second wall of the corner. Once it collides with this wall, it is guaranteed to be in a corner and be near the target point, completing the semantic command.

Similar behaviors can be used for semantic commands for sending the robot to

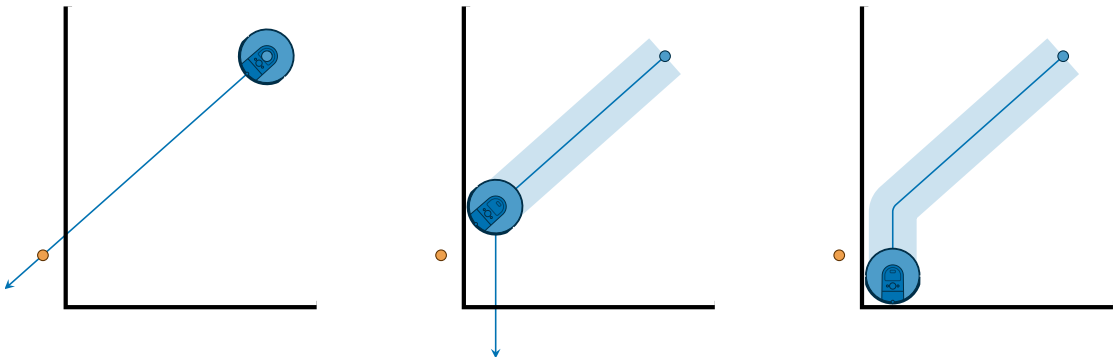


Figure 6.4: Behavior for executing a semantic command to go to corner near a specified point (orange). The robot heads in the direction of the point, potentially past the point, until it bumps into an obstacle. It then follows the obstacle in the approximate direction of the point until it bumps into the obstacle again.

Algorithm 6.1: Go to corner

Input: Map of environment, $\mathcal{Q} \subset \mathbb{R}^2$; and target point, $q \in \mathbb{R}^2$
Output: `success` or `failure`

while not near q **do**
  $p \leftarrow$ shortest path in $\mathcal{Q}$ from current position to q
  Extend last segment of p past q
  Follow p until collision or end of path
  if reached end of p without collision **then**
    return `failure`
$p \leftarrow$ path along wall in approximate direction of q
Follow p until collision or end of path
if had collision near q **then**
  return `success`
else
  return `failure`

a location near a wall or a location near the boundary of regions which have and have not already been covered. If the robot only needs to be along the wall and not in a corner, then it should travel along the wall near the point until the direction to the target point is orthogonal to the wall. When told to go to the edge of a previously covered region, its localization errors make it impossible to determine exactly when it is at this boundary. Instead, it should travel slightly past where it thinks the boundary is to guarantee that there will be at least a small amount

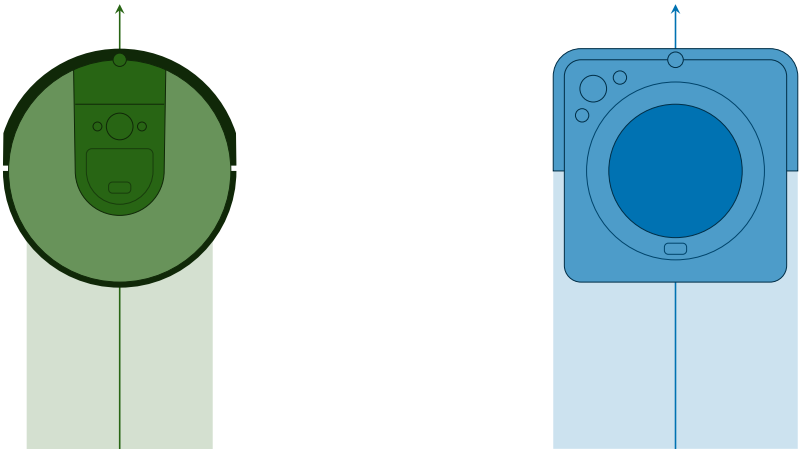


Figure 6.5: A coverage robot's coverage width is not necessarily equal to the robot width. For robotic vacuums (left), the coverage width is usually narrower; for robotic mops (right), the coverage width is often wider.

of overlap between its next rank and the previously covered region.

In coverage, a particularly important go-to semantic command is the command to go to the next rank. This command depends on the physical dimensions of the robot which can be described by two widths (Figure 6.5). The *robot width* is the physical width of the robot's body along its main wheel axis. This width determines how close it can get to an obstacle while turning. The *coverage width* is the width of the robot's coverage tool, which is typically less than the robot width. It determines the *rank width* which is slightly smaller to allow for some overlap between ranks.

When told to go to the next rank, the robot must turn 180° with a radius equal to the rank width (Algorithm 6.2). This maneuver sets the robot up to begin the next, antiparallel rank with a small overlap between consecutive ranks. If the robot is near an obstacle when told to go to the next rank, it cannot simply turn as the obstacle is in the way (Figure 6.6). Instead it must back up far enough to be able to complete the turn. The minimum distance to prevent colliding with the obstacle during the turn is the rank width plus half the robot width. If the obstacle is straight and orthogonal to the rank direction, backing up this distance

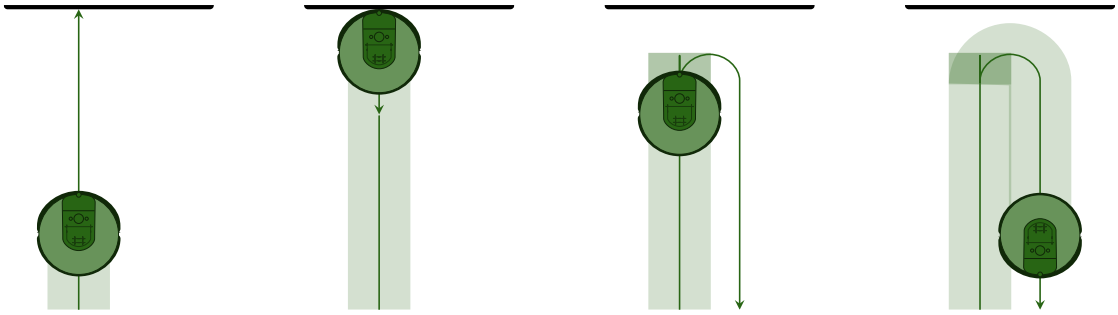


Figure 6.6: Behavior for executing a semantic command to go to the next rank after colliding with an obstacle. The robot had previously travelled forward and bumped into an obstacle. To go to the next rank, it backs up a short distance in preparation to turn. Then, it follows a tight curved path to turn around so it can begin coverage of its next rank.

guarantees the robot can complete the turn. For differently shaped obstacles, the robot may need to back up slightly farther.

Algorithm 6.2: Go to next rank

Input: Direction between ranks, θ
Output: `success` or `failure`

if near obstacle **then**
    Back up so that obstacle is rank width plus half the robot width away
if θ is clockwise from robot's heading **then**
    Turn 180° clockwise with radius equal to the rank width
else
    Turn 180° counter clockwise rank width
return `success`

6.3.2 Coverage commands

A large coverage mission gets executed using a sequence of semantic coverage commands. Prior to performing each coverage behavior, the robot first navigates to the start point of the coverage command using a go-to behavior. It then covers an interior or perimeter region with the appropriate coverage behavior.

An interior coverage command consists of a polygonal coverage region, a start

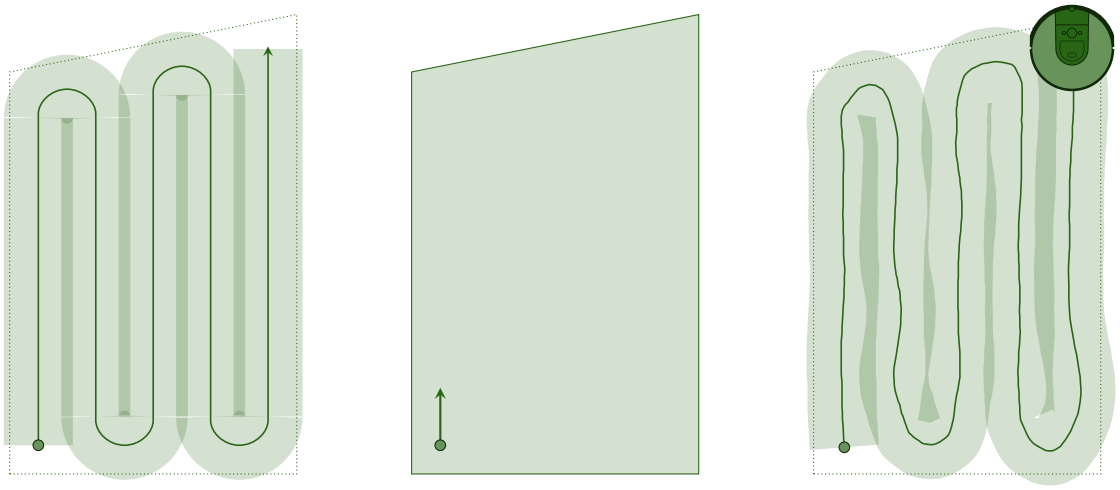


Figure 6.7: A robot's planned interior coverage path (left) is difficult to follow exactly due to localization errors and wheel. Instead, it is given a interior coverage semantic command (center). As long as there is some overlap between the regions covered by each rank, the actual path (right) will still fully cover the region despite it not matching the planned path exactly.

point in the coverage region, and two directions—one for ranking and one for turning. This command consists of significantly less information than the full planned path, enabling the robot to cover the region quickly despite its actual path differing from the planned path due to localization errors or wheel slip (Figure 6.7). The small overlap due to the difference between the rank and coverage widths ensures full coverage despite the inconsistencies between the planned and actual paths. The semantic command also does not prescribe the exact number of ranks, as small turning errors may result in the robot taking one rank more or less than expected (Figure 6.8). As long as the entire region gets covered, it does not matter how many ranks the robot used.

The actual interior coverage behavior alternates between travelling forwards in the rank direction (or its opposite) and turning 180° in the turning direction (Figure 6.9, Algorithm 6.3). As this behavior is always initiated after a go-to behavior, the robot always starts in the start point. It then rotates to the rank direction and moves forward until it either reaches the end of the rank—determined

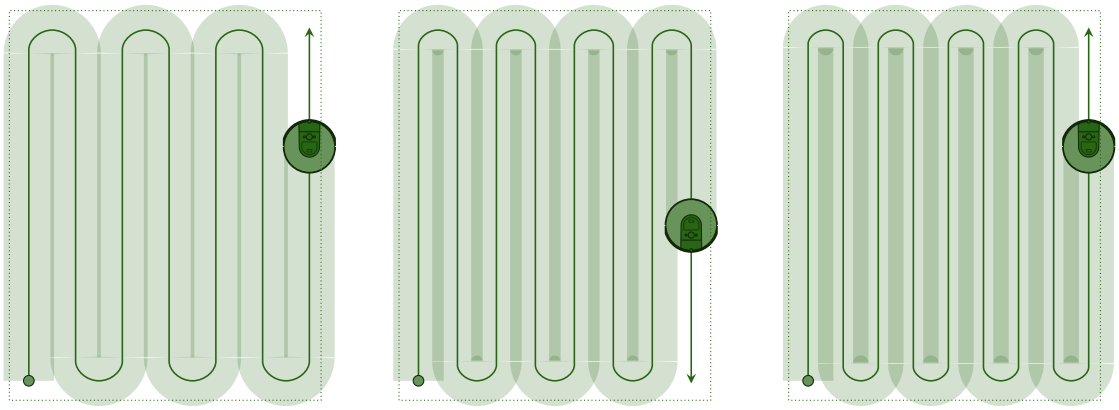


Figure 6.8: If the robot's turning radius is slightly smaller (left) or larger (right) than expected (center), the number of ranks needed to cover an interior region may be more or less than expected.

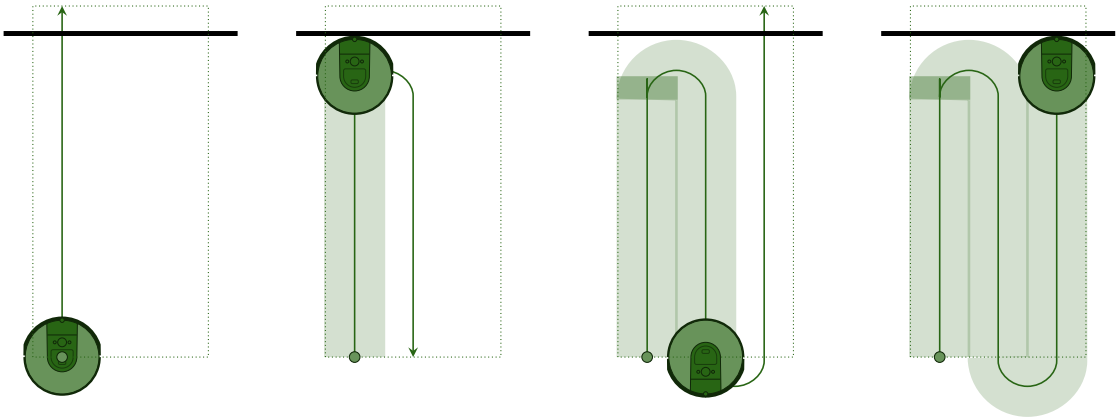


Figure 6.9: Behavior for executing a semantic command to cover an interior region (dotted). The robot heads in a straight line in the initial direction, covering the first rank, until it either collides with an obstacle or reaches the edge of the coverage region. It then turns in the secondary direction, positioning itself to start the next rank. This process repeats until the entire region has been covered.

by either leaving the end of the coverage region or bumping into an obstacle. If it encounters a small obstacle in the middle of the rank, it travels around the obstacle without disrupting its rank. Once it reaches the end of its rank, it turns 180° in the turning direction to start the next rank. The robot continues these behaviors until it leaves the side of the coverage region after making a turn at which point it has successfully completed the command.

Algorithm 6.3: Interior coverage

Input: Coverage region, $\mathcal{Q}_{\text{cov}} \subset \mathbb{R}^2$; start point, $q \in \mathcal{Q}_{\text{cov}}$; initial rank direction, θ_0; and direction between ranks, θ_1

Output: `success` or `failure`

```
1  if not at q then
2      Go to q                                  /* Algorithm 6.1 or similar */
3  Turn to direction θ_0
4  while in Q_cov do
5      while (in Q_cov) and (has not collided with obstacle) do
6          Go forward
7      if has collided with small interior obstacle then
8          Travel around obstacle
9      else if has collided with large interior obstacle then
10         return failure
11     else
12         Go to next rank in direction θ_1          /* Algorithm 6.2 */
13 if has covered most of coverage region then
14     return success
15 else
16     return failure
```

Perimeter coverage is also performed using a behavior in response to a semantic command (Figure 6.10, Algorithm 6.4). The perimeter coverage command consists of start and end points along the perimeter and a start direction along one of the walls at the start point. The robot first navigates to the start point using a go-to command which is modified slightly so the robot is more likely to be facing the correct direction when it reaches the start point. Once the robot is at start point, it covers the perimeter by following along the perimeter, using its sensors to remain as close as possible to the wall or obstacle. If the robot bumps into a wall, it turns and continues perimeter coverage. The robot successfully completes the command when it arrives at the end point. If it returns to the start point before reaching the end point (assuming the two points are different), the behavior fails because the command contained start and end points which are not part of the same perimeter.

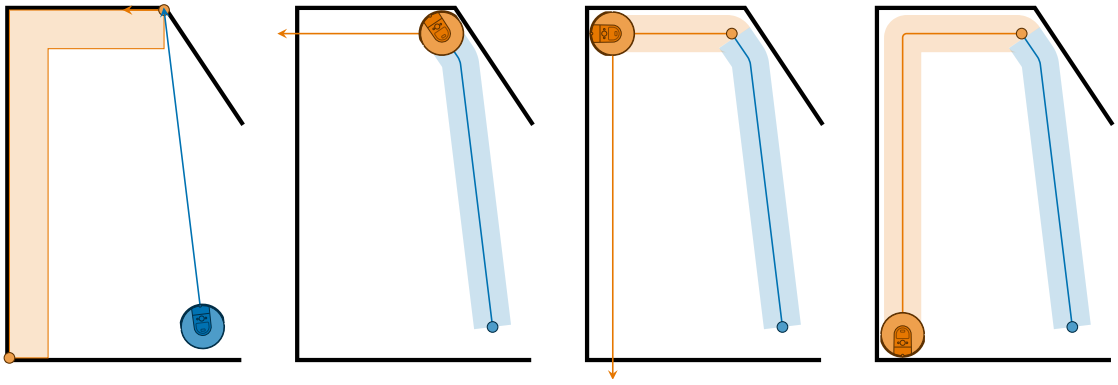


Figure 6.10: Behavior for executing a semantic command for perimeter coverage. First the robot navigates to the start point of the perimeter coverage command. It then rotates to the start direction and follows along the wall, turning when it reaches a corner, until it is near the end point of the command.

Algorithm 6.4: Perimeter coverage

Input: Start point, q; end point, q'; and direction, θ
Output: `success` or `failure`

```
1  if not at q then
2      Go to q                          /* Algorithm 6.1 or similar */
3  Turn to direction θ
4  while not near q' do
5      while has not collided with new wall do
6          Go forward while remaining close to wall
7          if has returned to q ≠ q' then
8              return failure
9      Turn to direction of new wall
10 return success
```

6.4 Processing maps for coverage

A map of the robot's environment is essential to its ability to create a coverage plan. A robot's map is not a perfect description of its environment, but is instead filtered through the robot's sensors. As this map is representative of the robot's SLAM system, it is well suited for localization but not necessarily coverage. For example, a robot with a laser scanner may produce a very high resolution map, whereas coverage benefits from a lower resolution map which only shows features

larger than the robot width. Additionally, it may be stored in the wrong format, such as an occupancy grid, instead of the two dimensional polygonal map used by the coverage planner from Chapter 4. Therefore, before planning, we first process the robot's map to create a *behavior-based* map which reflects the robot's known coverage behaviors and is as simple as possible while still containing all the details necessary to create a coverage plan. The basic procedure of processing a map (Algorithm 6.5) involves classifying all parts of the map as either free or occupied, ensuring the entire free section is connected, removing small obstacles, and straightening walls.

Algorithm 6.5: Process map

Input: Occupancy grid of free, occupied, and unknown pixels
Output: Simplified polygonal environment, $\widehat{\mathcal{Q}}$

```
Classify unknown pixels as either free or occupied  /* Algorithm 6.6 */
if free pixels do not form a connected component then
    for pairs of connected region of free pixels do
        connecting_pixels ← occupied pixels between free regions
        if there are only a few pixels in connecting_pixels then
            Mark connecting_pixelsas free
    Mark any remaining disconnected free regions as occupied
∂Q ← traced boundary of occupancy grid
Remove small inner polygons of ∂Q
∂Q̂ ← simplified version of ∂Q
Q̂ ← region bounded by ∂Q̂
return Q̂
```

6.4.1 Classifying the unknown

All maps consist of several regions which are labelled as either free, occupied, or unknown (possibly including probability of being free). If the robot has been making the map for a long time, it may be entirely known; however in most cases, there will be unknown regions. Large unknown regions correspond to areas where the robot has not explored; small unknown regions are usually mapping errors. Before planning, we first classify all unknown regions as either free or occupied based

on what regions are nearby. Although any classification algorithm will regularly make mistakes, misclassifications are not a big problem. The robot will continue mapping as it is covering and will generally discover its mistakes before it arrives at the misclassified region and can therefore replan (see Section 6.5) accordingly.

The classification algorithm depends on the map format and the methods used to create the map—different sensors tend to have different kinds of mapping errors. One of the robots used in my research used occupancy grid maps. These maps consist of an array of pixels which are each labelled as either free, occupied, or unknown. I classified unknown pixels using a simple idea: an unknown region surrounded by mostly free pixels is likely to be free (Algorithm 6.6). The algorithm first identifies *seed* pixels which are unknown pixels with at least two free neighbors and no occupied neighbors (Figure 6.11). The region grows out from the seed pixel, either horizontally or vertically becoming a one-dimensional row of unknown pixels capped by either a known pixel or the edge of the map. If this row is mostly bounded by free pixels—three of its four sides consist entirely of free pixels—then entire unknown row is labelled as free (Figure 6.12). The algorithm continues this process, iteratively creating rows of unknown pixels from seed pixels. Once all seed pixels have been checked, all remaining unknown pixels are labelled occupied (Figure 6.13). This algorithm tends to fill in small unknown regions but classifies most unknown pixels as occupied.

6.4.2 Removing small obstacles

The interior coverage behavior described by Algorithm 6.3 includes behavior where the robot goes around a small obstacle and continues its current rank. For obstacles entirely within the robot's rank, the robot can circle the obstacle and continue the rank; for obstacles straddling two ranks, the robot simply diverts the ranks to either side of the obstacle (Figure 6.14). Larger obstacles would require more complicate maneuvers disrupting the interior coverage behavior.

Algorithm 6.6: Classify unknown pixels

Input: Occupancy grid of free, occupied, and unknown pixels
Output: Occupancy grid of free and occupied pixels

```
while not all pixels have been checked do
    for unchecked unknown pixel in grid do
        if pixel has 2 free neighbors but no occupied neighbors then
            Use this pixel as seed pixel
            break for
        else
            Pixel has been checked
    for direction θ in {horizontal, vertical} do
        Initialize row of pixels containing only the seed pixel
        while pixel adjacent to row in direction θ is unknown do
            Extend row by one pixel in direction θ
        while pixel adjacent to row in direction −θ is unknown do
            Extend row by one pixel in direction −θ
        if all of the row's neighbors on three of its sides are free then
            Set all pixels in the row to free
            Reset unknown pixels to be unchecked
Set all remaining unknown pixels to occupied
return occupancy grid
```

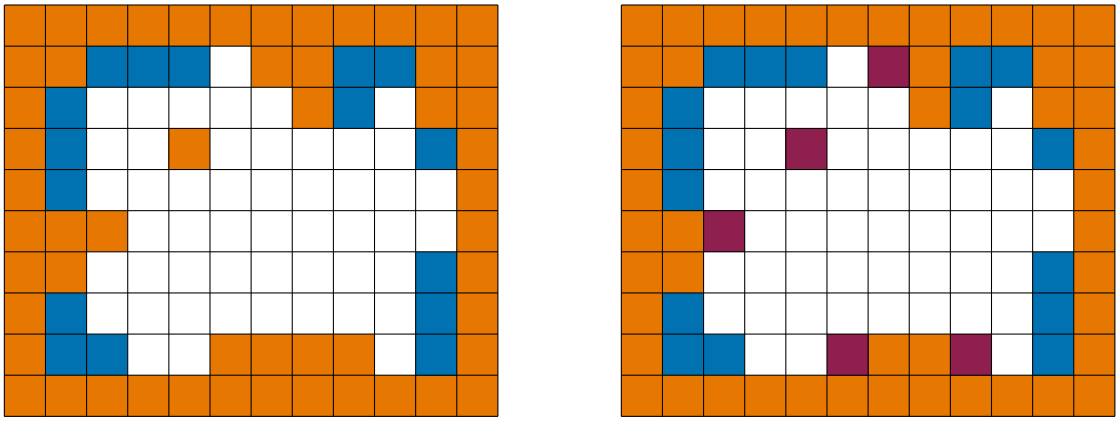


Figure 6.11: Example of an occupancy grid with free (white), occupied (blue), unknown (orange) pixels (left). When classifying unknown regions, we use seed pixels (red) which are unknown pixels adjacent to two or more free pixels and no occupied pixels (right).

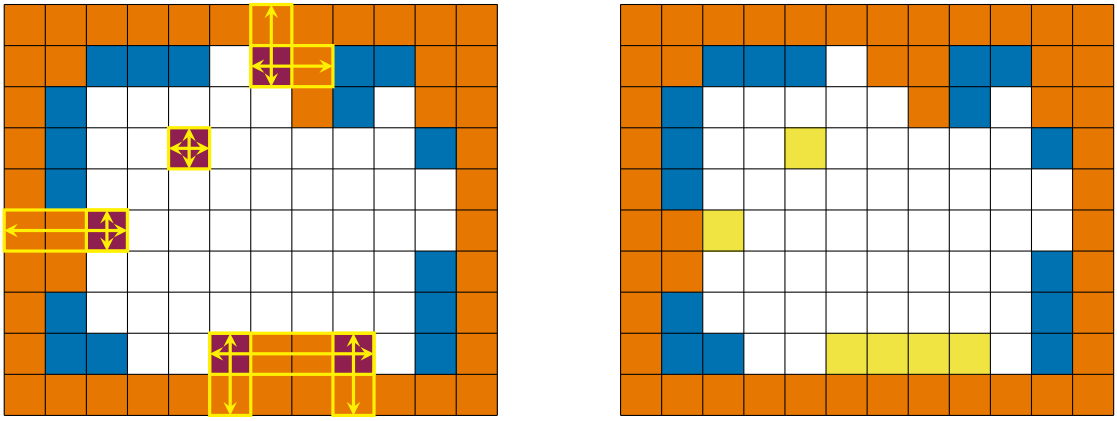


Figure 6.12: Each seed pixel can be extended both horizontally and vertically to create two different rows of unknown pixels (left). A row is likely free (yellow) if three of the row's sides are completely covered by free pixels (right).

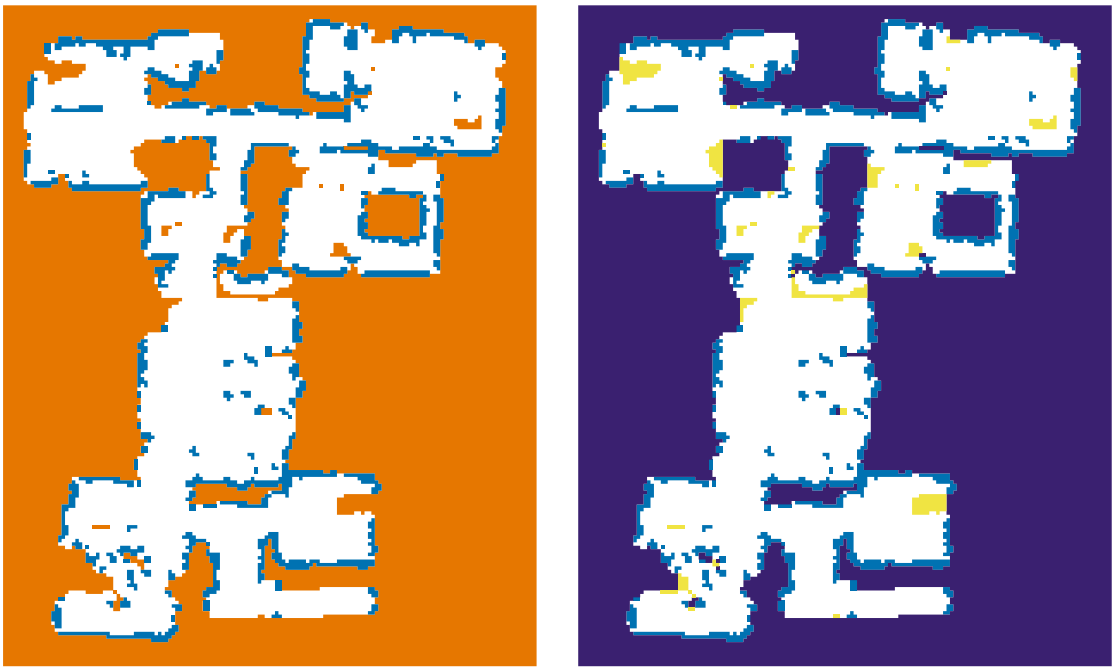


Figure 6.13: An occupancy grid map containing free (white), occupied (blue), and unknown (orange) pixels produced by an iRobot Roomba in a test environment features (left). The unknown pixels can be classified as likely free (yellow) or likely occupied (purple) using Algorithm 6.6 to obtain a map of only free and occupied pixels (right).

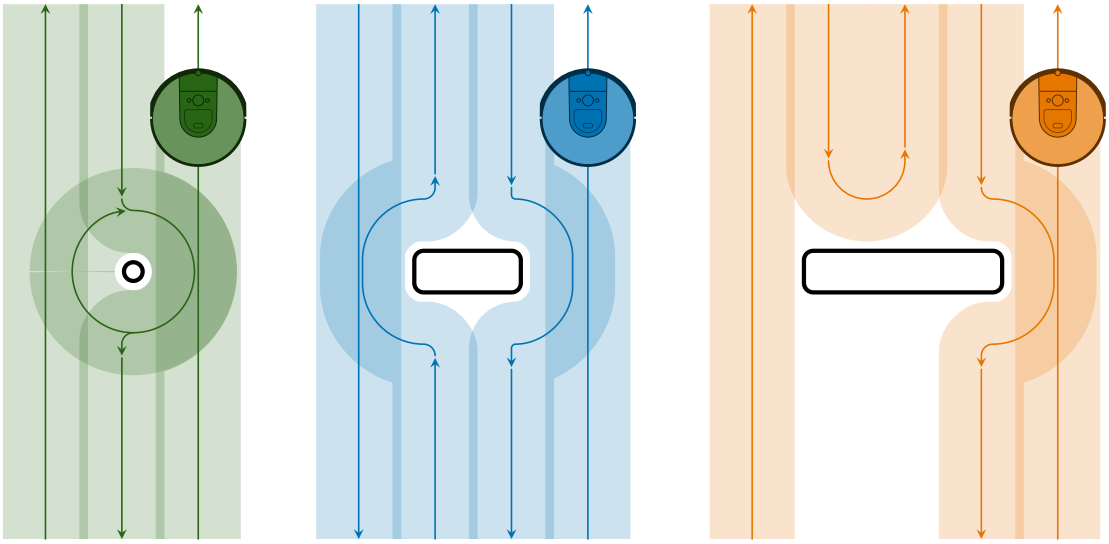


Figure 6.14: When a coverage robot encounters an obstacle narrower than a rank (left) or between one and two ranks in width (center), it can cover around the obstacle without significantly altering its ranks, so the obstacle can be removed from the map. For larger obstacles (right), the robot would have to travel more than a rank-width to get around the obstacle so it should remain in the map.

As the small obstacles do not affect the robot's coverage behavior, they are not included in the map used for coverage planning. Similarly, obstacles with very small gaps between them are combined into one larger obstacle. The map with small obstacles removed is always simpler than the original occupancy grid (Figure 6.15). The amount that the map gets simplified depends on the physical properties of the robot because the threshold for removing obstacles depends on the size of the robot. In this way, the map reflects not only the actual environment, but also the behavior of the robot. Typically, this simplification procedure also removes small regions of free space that are disconnected from the main free region and assumed to be mapping errors.

6.4.3 Straightening walls

When performing perimeter coverage via Algorithm 6.4, the robot uses its sensors to follow along the exact perimeter whether or not it is straight. Similarly, when

Figure 6.15: The raw binary occupancy grid map (left) contains many small obstacles and clustered obstacles. As a coverage robot can go around small obstacles without disrupting coverage and cannot go between the clustered obstacles, the small obstacles are removed and the clustered obstacles are connected in the simplified map (right).

performing interior coverage via Algorithm 6.3, the robot turns once it has reached the perimeter, regardless of whether or not its rank was the same length as the previous one. Both of these behaviors mean that it does not matter if the perimeter is actually straight. Coverage of two regions—one with an uneven perimeter and one with a straight perimeter—require the exact same semantic coverage commands (Figure 6.16).

Coverage planning is somewhat simpler for a map with straight edges than curved ones because the polygon describing the perimeter has fewer vertices. Therefore, we simplify the map's boundary before using it for coverage planning. If the map is an occupancy grid, its boundary can be obtained via one of many simple boundary tracing algorithms [169]. When simplifying the boundary, we want to approximate the original, detailed boundary by one with long straight walls whenever possible. Due to the robot's perimeter coverage behavior, if the

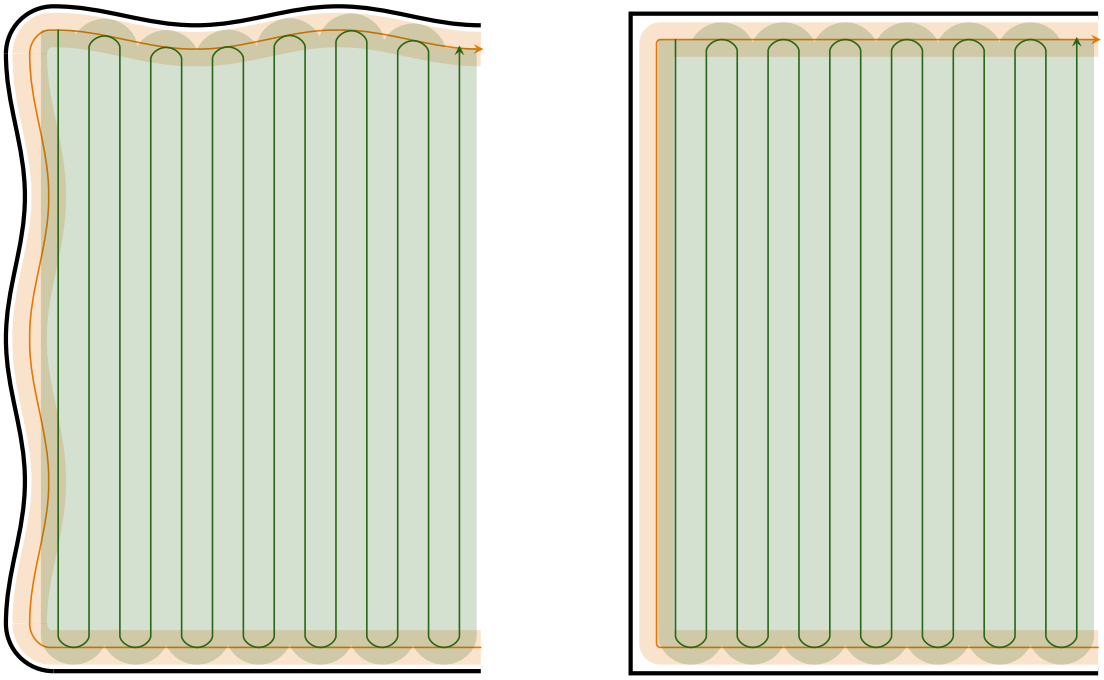


Figure 6.16: Perimeter ranks (orange) along an uneven perimeter (left) and along a straight perimeter (right) often require the same interior ranks (green). In these cases, the uneven perimeter can be straightened on the map.

actual boundary is up to half a rank width away from the boundary of the map, it has no effect on the resulting coverage plan. Therefore the simplification problem is to find a polygon with a minimal number of edges such that all vertices of the original polygon are at most half a rank width away from this new polygon. This problem can be solved in $\mathcal{O}(n^2)$ [34] although it is often approximated using the Douglas-Peucker heuristic which runs in $\mathcal{O}(n \log(n))$ [51].

6.5 Replanning

A robot will attempt to cover the entire region using its planned sequence of coverage behaviors. If it gets interrupted and does not complete the entire plan, or discovers an error in its plan, it will need to replan to ensure that it completes the entire mission. A robot needs to replan whenever:

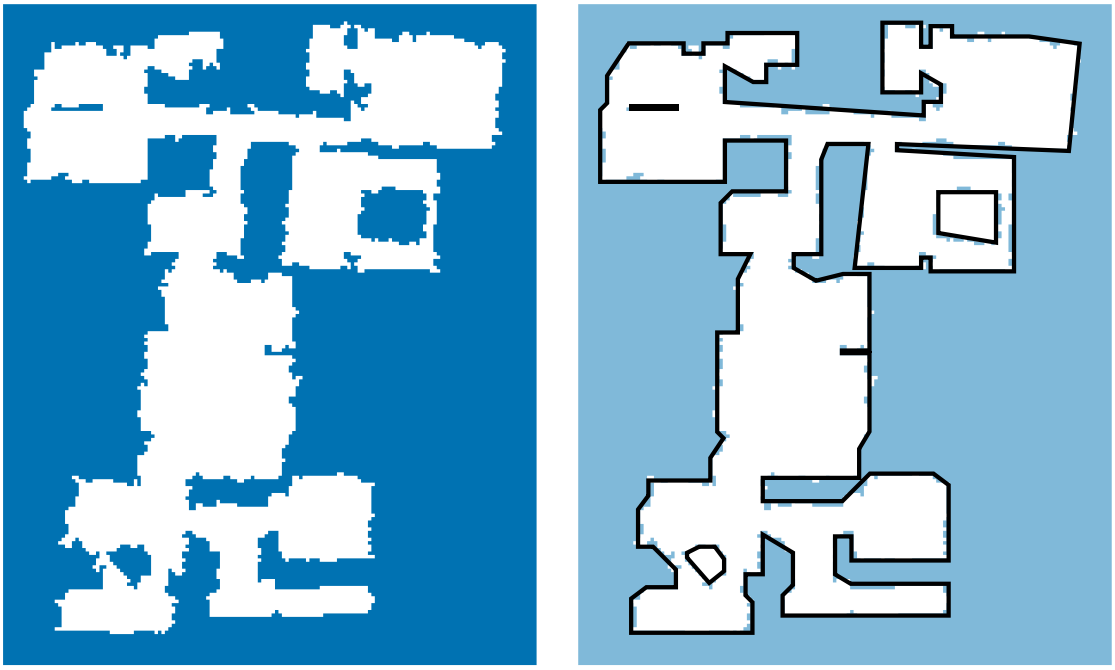


Figure 6.17: A processed occupancy grid map (left) can be converted into a polygonal map by tracing the boundary between free and occupied pixels. This boundary can be straightened using the Douglas-Peucker algorithm [51] to obtain a simplified polygonal map that is useful for coverage planning (right).

1. It runs out of battery and needs to recharge before resuming coverage;

2. A human kidnaps the robot and moves it to a different location;

3. The robot finishes coverage of an interior region at the opposite corner of where it expected to finish; or

4. Its map changes.

Multirobot teams additionally need to replan whenever:

5. One robot in the team gets damaged or stuck;

6. A new robot joins the team; or

7. Robots covered at different speeds and they need to rebalance the workload.

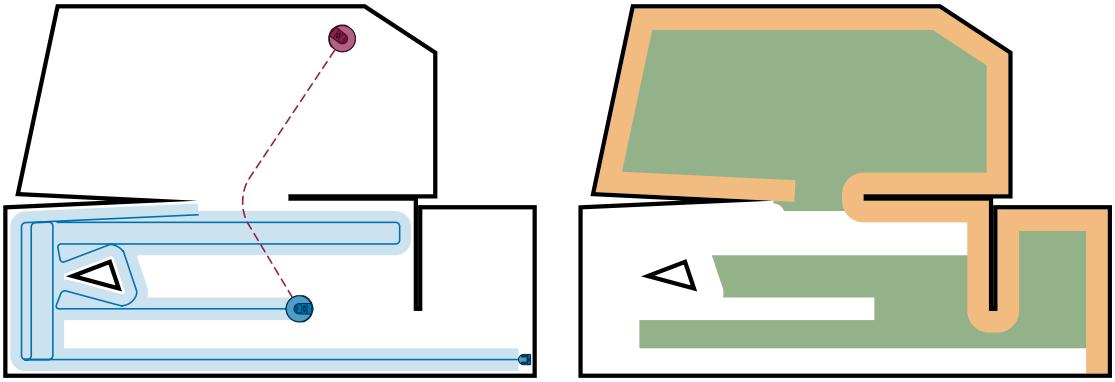


Figure 6.18: When a human kidnaps a robot (left), the robot gets moved away from its coverage plan (blue) and is carried to some random location (red). The remaining interior (green) and perimeter (orange) regions that the robot still needs to cover (right) are smaller than at the start of the mission.

When any of these events happens, the robot or team of connected robots replans based on their current positions, current map, and knowledge of where they have already covered. The new plan can also be based on the previous plan to reduce the computational burden of replanning.

6.5.1 A new location

For consumer robots, kidnapping is the common scenario where a human picks up their robot and carries it to a new location [104]. After a kidnapping, the robot only needs to cover the regions that it hasn't already covered (Figure 6.18). It could simply resume its existing plan by travelling back to the location where it got kidnapped, and following its original path. This approach is generally not the most efficient as it may spend quite a bit of time returning to that location and it might get kidnapped again if the human wants the robot to avoid that location. Instead, it should plan a new coverage path based on its new location.

When replanning (Algorithm 6.7), the robot can use some of its former plan. Its original plan consists of a set of semantic commands which each correspond to a set of ideal ranks. Depending on if the command was issued and if it was

completed, the new plan will include all, some, or none of these ranks. Completed commands do not need any of their ranks in the new plan. The command the robot was working on when the kidnapping will have some of its ranks included. The commands that were not issued will have all of their ranks included. The resulting set of ranks for the new plan (Figure 6.19) fully covers the regions that were not covered before. The robot then plans its new coverage path by solving the TSP on this set of ranks with its new location as the path's start point, and its original end location—likely its charging station—as the end point. As the set of ranks is not necessarily connected, the robot still needs to know the geometry of the full environment so that it can, if necessary, use already covered free space to plan short paths between rank endpoints.

Algorithm 6.7: Replan after kidnap

Input: Old coverage path, p; and new start location, q'
Output: New coverage path p'

```
1 R ← {}                                          /* Set of uncovered ranks */
2 for semantic command of p do
3     if command was not completed then
4         R_rem ← ranks that would have been covered by plan
5         R ← R ∪ R_rem
6 p' ← shortest path starting at q' covering R          /* Algorithm 3.4 */
7 return p
```

A similar situation happens when the robot runs out of battery before completing its mission (Figure 6.20). If the robot detects its battery is running low, it plans a path to its charging station from wherever it happens to be. This event can be treated the same as a kidnapping as the robot now has to plan a new path to cover the remaining space starting from a new location: its charging station. If this path is quite short, it may resume coverage before fully recharging as long as it has enough power to finish the planned path. Alternatively, if the robot knows that it won't be able to cover the entire environment on a single charge, it can plan two coverage paths by pretending that there are two robots which both start and

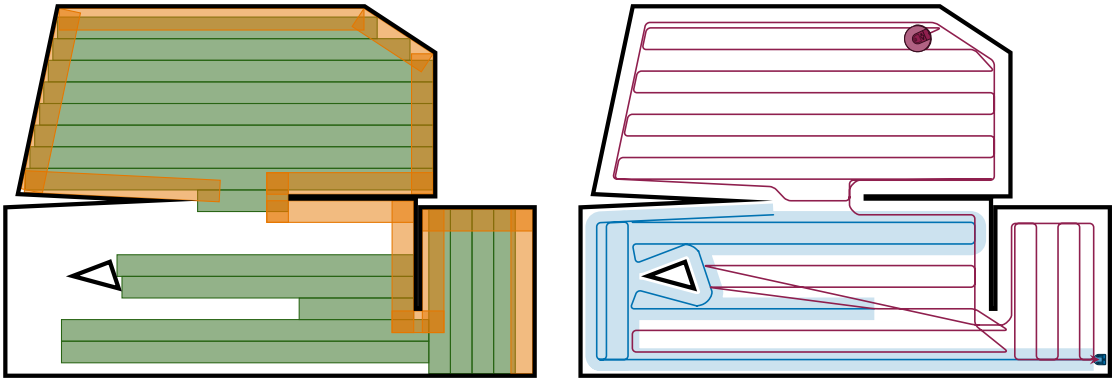


Figure 6.19: When replanning after a kidnapping, the robot can use the same rank partition as it originally used, except with some of the ranks removed (left). The new coverage path (right) covers all of these paths and the order of ranks is optimized for starting at the robot's new location after kidnapping.

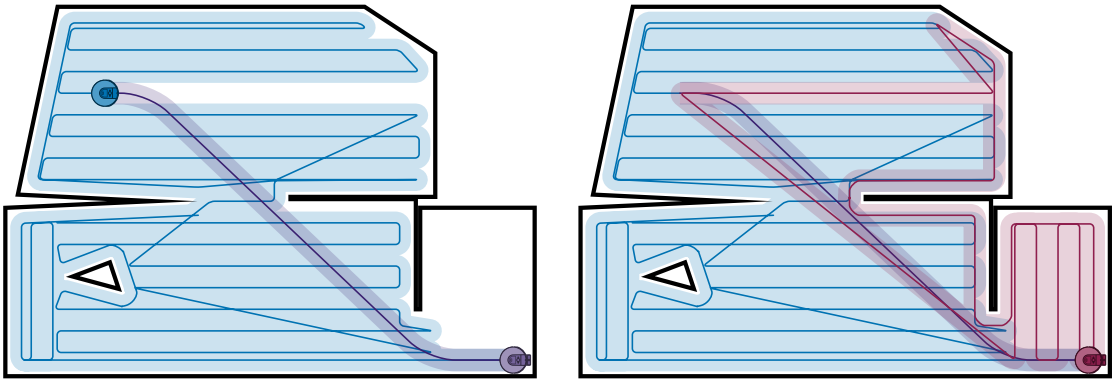


Figure 6.20: When a robot's battery is low, it returns to its charging station without completing the mission (left). After recharging, it plans a new path (red) which covers the remaining uncovered region.

end at the same charging station. Instead of these robots covering their respective coverage paths simultaneously, the same robot covers both paths consecutively and charges in between.

A third replanning scenario occurs when the robot finishes an interior coverage command in a different location than expected (Figure 6.21). Covering the same interior region with an even or an odd number of ranks will result in the robot finishing in different corners of the region. As the robot's ranks may be slightly narrow or wider than expected, it may take one more or one fewer rank and end

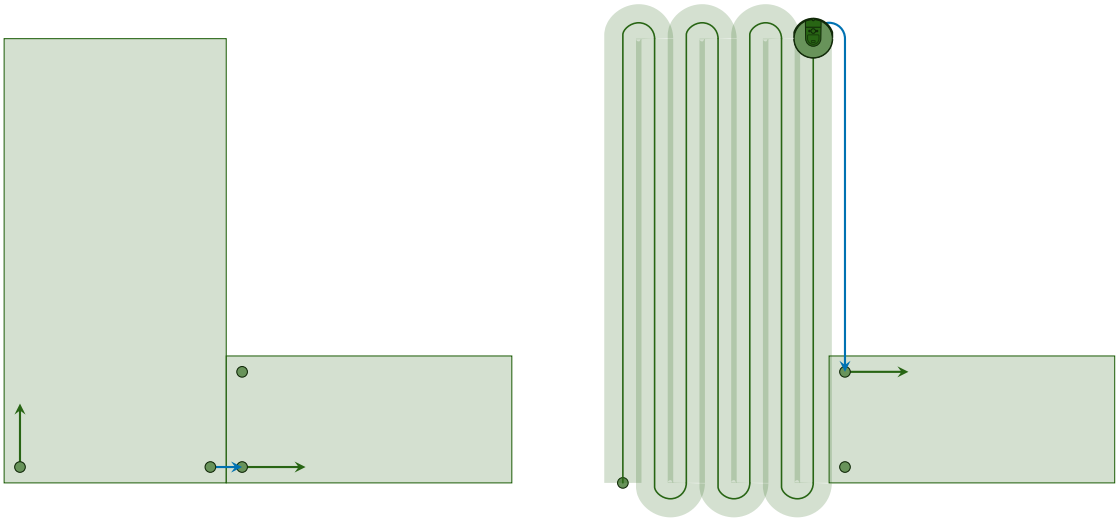


Figure 6.21: Depending on the number of ranks needed to cover an interior region, the robot may end up in one of two different corners. When planning (left), the robot expected the first region to take an even number of ranks so it would finish in the bottom right corner. In reality (right), the first region ended up taking an odd number of ranks so it ended up in the opposite corner. From this position the robot replans its coverage of the next region to start in the closest corner to where it actually finished.

up in the other corner. It can then replan the route to its remaining ranks to make coverage slightly more efficient.

6.5.2 Map changes

Differences between the robot's map and the actual geometry of the environment can also require replanning. The robot's semantic behaviors mean that small differences—a wall being further away than expected, or a new small obstacle in the middle of the room—do not affect plan so no replanning is necessary. Larger differences which drastically change the size or topology of the coverage region require replanning. In indoor environments, large differences most often occur when a door has recently been opened or closed (Figure 6.22). An open door either makes a new room accessible or adds an additional path to a room that was already accessible. A closed door has the opposite effect, reducing the number of

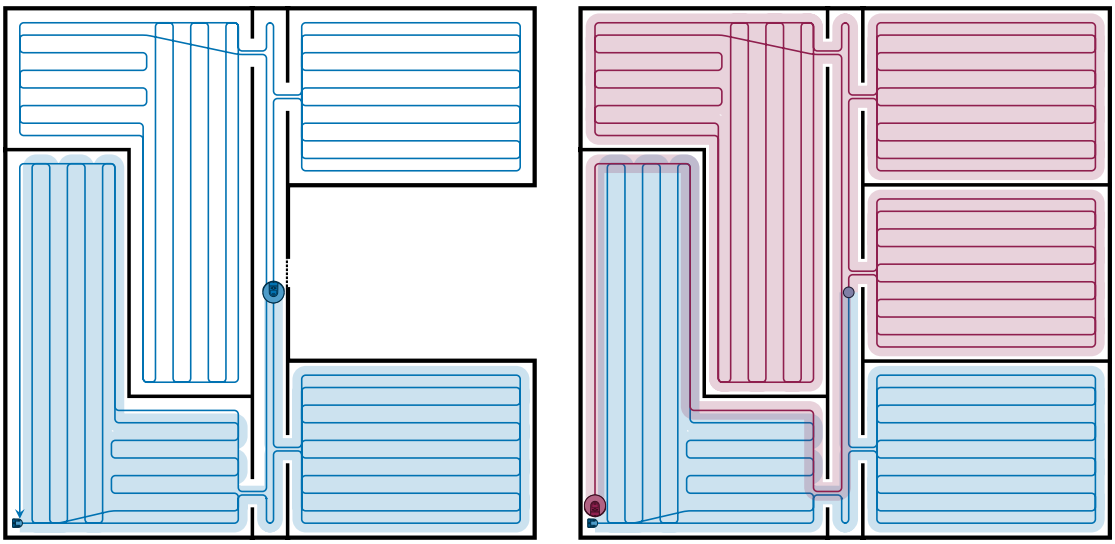


Figure 6.22: During a coverage mission, a robot may find an open door which it thought was closed (left). When it finds such a door, it replans its coverage path for the remaining region including the newly accessible part of the map behind the door (right).

ways the robot can get to certain places.

When it discovers a large change in the map, the robot replans a new coverage path that covers all the uncovered regions of the new map (Algorithm 6.8). It first computes the remaining coverage region, $\mathcal{Q}_{\text{rem}}$, by subtracting the already covered region, $\mathcal{Q}_{\text{cov}}$, from the current map, $\mathcal{Q}$. The perimeter ranks are constructed similar to Algorithm 4.1 with two changes (Figure 6.23):

1. Perimeter ranks are only added for edges of $\mathcal{Q}_{\text{rem}}$ that are also edges of $\mathcal{Q}$ and not for edges which are boundaries between $\mathcal{Q}_{\text{rem}}$ and $\mathcal{Q}_{\text{cov}}$; and

2. Perimeter ranks which cross a boundary between $\mathcal{Q}_{\text{rem}}$ and $\mathcal{Q}_{\text{cov}}$ are only extend one rank width into $\mathcal{Q}_{\text{cov}}$ to minimize redundant coverage while guaranteeing no missed coverage due to turning.

Similarly, Algorithm 4.2 is modified slightly when performing the rectilinear contraction used to construct perimeter ranks (Figure 6.24):

3. The contraction includes every grid cell fully contained in the environment

which has not already been fully covered.

With this definition, the rectilinear contraction may actually *expand* $\mathcal{Q}_{\text{rem}}$ to include cells that have only been partially covered by the robot's previous path. Aside from these three changes, the remainder of Algorithm 4.6 is unchanged when computing the rank partition during replanning. The new coverage path is obtained by solving the TSP for this rank partition with the robot's current location as the start point.

Algorithm 6.8: Replan after map change

Input: Environment, $\mathcal{Q} \subset \mathbb{R}^2$; and covered region, $\mathcal{Q}_{\text{cov}} \subset \mathcal{Q}$
Output: New coverage path, p

$\mathcal{Q}_{\text{rem}} \leftarrow \mathcal{Q} \setminus \mathcal{Q}_{\text{cov}}$
$\mathcal{R} \leftarrow \{\}$ `/* Set of uncovered ranks */`
$\mathcal{R}_{\text{per}} \leftarrow$ perimeter ranks of $\mathcal{Q}$ `/* Algorithm 4.1 */`
for perimeter rank r in $\mathcal{R}_{\text{per}}$ **do**
    if r is fully contained in $\mathcal{Q}_{\text{rem}}$ **then**
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$
    else if r is partially contained in $\mathcal{Q}_{\text{rem}}$ **then**
        Shorten r until it only extends one rank width into $\mathcal{Q}_{\text{rem}}$
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$
$\mathcal{Q}_{\text{rect}} \leftarrow$ rectilinear polygon covering $\mathcal{Q}_{\text{rem}}$ `/* Algorithm 4.2 */`
$\mathcal{R}_{\text{int}} \leftarrow$ optimized interior ranks for $\mathcal{Q}_{\text{rect}}$ `/* Algorithm 4.5 */`
$\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{R}_{\text{int}}$
$p \leftarrow$ shortest path covering $\mathcal{R}$ `/* Algorithm 3.4 */`
return p

6.6 Single robot robust coverage

While interning at iRobot, I implemented a version of my coverage strategy on the iRobot Roomba i7 robotic vacuum cleaner. The Roomba's previously published coverage strategy is not based on a map [70, 178]. The coverage strategy is based on a system of *frontiers* and rectangular coverage regions. The robot iteratively identifies a frontier—the boundary between known and unknown grid cells—and tries to cover a rectangular region near that frontier via a ranking behavior. When

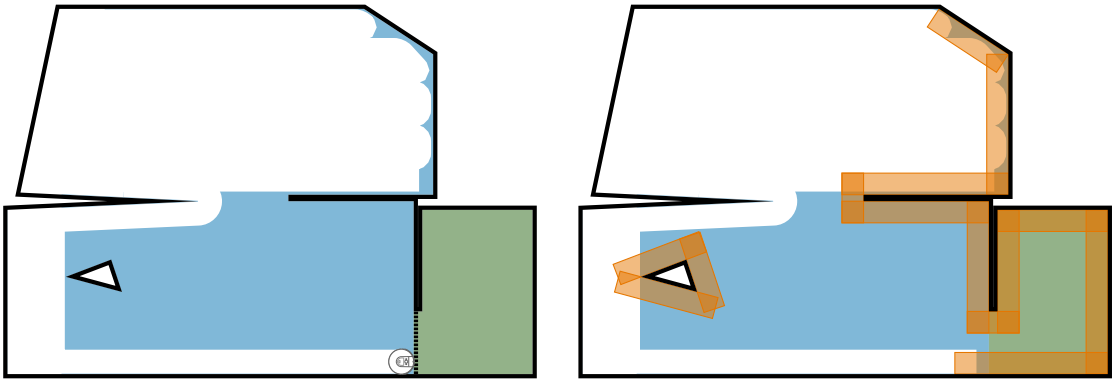


Figure 6.23: When a robot discovers a new region of the map (left) it replans using the uncovered part of the old map (blue) and this new region (green). Perimeter ranks (orange) are only added for edges of this coverage region adjacent to boundaries (right). These edges are extended at convex corners and at boundaries with the already covered region to prevent missed coverage due to turning.

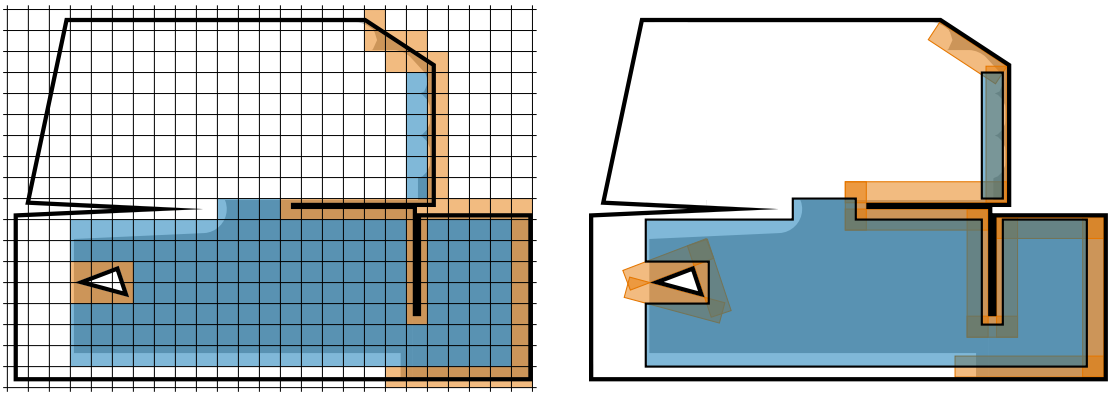


Figure 6.24: When replanning, the robot must cover all of the partially or fully uncovered covered grid cells (left). These cells can either be classified as interior (blue) or perimeter (orange) depending on whether or not they are fully contained in the interior of the new environment. The rectilinear contraction includes any interior cells that have not been fully covered (right).

it collides with obstacles, it has specific rules to determine whether or not continue ranking and where the new frontiers are. After completing interior coverage via this strategy, it finishes its mission by following the perimeter of the recently covered region. Despite the relative simplicity of the strategy, it has been effective in a wide variety of homes—the product has been a huge commercial success—and is therefore a good comparison for my strategy. The robot's existing map can also be used in a limited capacity for *room-by-room* coverage where the map is first partitioned into rooms which are covered consecutively using the same coverage strategy [106].

The implementation of my coverage strategy (Algorithm 6.9) uses the Roomba's map for planning and the same low-level behaviors as the comparison strategy to execute the plan. It first simplifies the map, removing small obstacles and straightening walls, and then uses this map to plan a coverage strategy for the remaining uncovered area. The individual semantic commands of both strategies are executed by low-level behaviors which are semantically equivalent to Algorithms 6.3 and Algorithm 6.4. Replanning happens whenever the robot robot is kidnapped, has a low battery, finishes in the opposite corner, updates its map, or fails to complete a command. The robot will continue to replan and perform the planned coverage behaviors until its entire map has been covered (success) or the robot gets stuck (failure).

To compare my strategy with the Roomba's previously published strategy, I needed to run the robot in identical environments using both strategies. Although I was able to run both strategies on real robots, it was difficult to compare them in a real environment. The only environment I had access to was the office where many humans and robots occupy the same space and test robots regularly get interrupted. As the coverage missions regularly take more than an hour to complete and their coverage times can be impacted by humans and other robots nearby, it was difficult and frustrating to try comparing robots fairly. Furthermore, changing

Algorithm 6.9: Cover real environment

Input: Initial map, $\mathcal{Q}$
Output: `success` or `failure`

```
1  Q̂ ← processed and simplified version of Q          /* Algorithm 6.5 */
2  while some of Q still needs to be covered do
3      p ← coverage path for uncovered regions of Q̂   /* Algorithm 6.8 */
4      for semantic command of p do
5          Execute command                  /* Algorithms 6.3 or 6.4 */
6          if map changed then
7              Q ← updated map from robot's mapping module
8              Q̂ ← processed and simplified version of Q
9          else if low battery then
10             Go to charging station
11         else if stuck then
12             return failure
13         if command not completed successfully then
14             break for
15 return success
```

lighting conditions throughout the day often have a significant impact on the robot's localization making it even more difficult to isolate the actual effect of the different coverage strategy.

Rather than use a real environment, I resorted to using iRobot's simulator which is intended to help developers quickly test changes to robots without needing to construct custom physical environments or worry about human interference or varying lighting conditions. This simulator uses a simulated indoor environment built-in Gazebo and the simulated Roomba runs the exact same code as the development robots. Furthermore, it faithfully reproduces the wheel slip and SLAM errors of a real robot.

6.6.1 Results

I created a simulated indoor environment consisting of two rooms—one rectangular and one L-shaped—which were covered by the simulated Roomba using both my coverage strategy and the comparison strategy in a room-by-room coverage mode

(Figure 6.25). Although, the actual paths followed by the robot when using the two strategies are quite similar, there are a few main differences:

1. My strategy uses two directions of coverage, which is noticeable in the upper left of the L-shaped room;

2. There are fewer repeated ranks in my strategy, most notable in the middle of the rectangular room and the right edge of L-shaped room; and

3. The diagonal connecting paths are slightly shorter for my strategy.

All of these differences result in more efficient interior coverage when using my strategy. Perimeter coverage, on the other hand, is largely unchanged and in both cases the robot performs some unnecessary back-and-forth motions near doorways. This behavior is not intended by either planner and is instead a quirk of the perimeter follow behavior used by both strategies.

In addition to the qualitative analysis of the coverage paths, the strategies can be compared by their coverage times (Figure 6.26). As the strategies were both run within a room-by-room framework, I was able to divide the total coverage times by room and by behavior (interior or perimeter). For both rooms, my strategy was much faster for interior coverage. This improvement was largest in the L-shaped room where turn-minimization resulted in a second direction of coverage. Additionally, coverage of both rooms benefited from improved planning that reduced repeat ranks. The perimeter coverage times were more similar for both strategies with the differences primarily due to differences in how long it took for the robot to make it through the door. Over a large number of missions, I expect all significant improvements to occur during interior coverage.

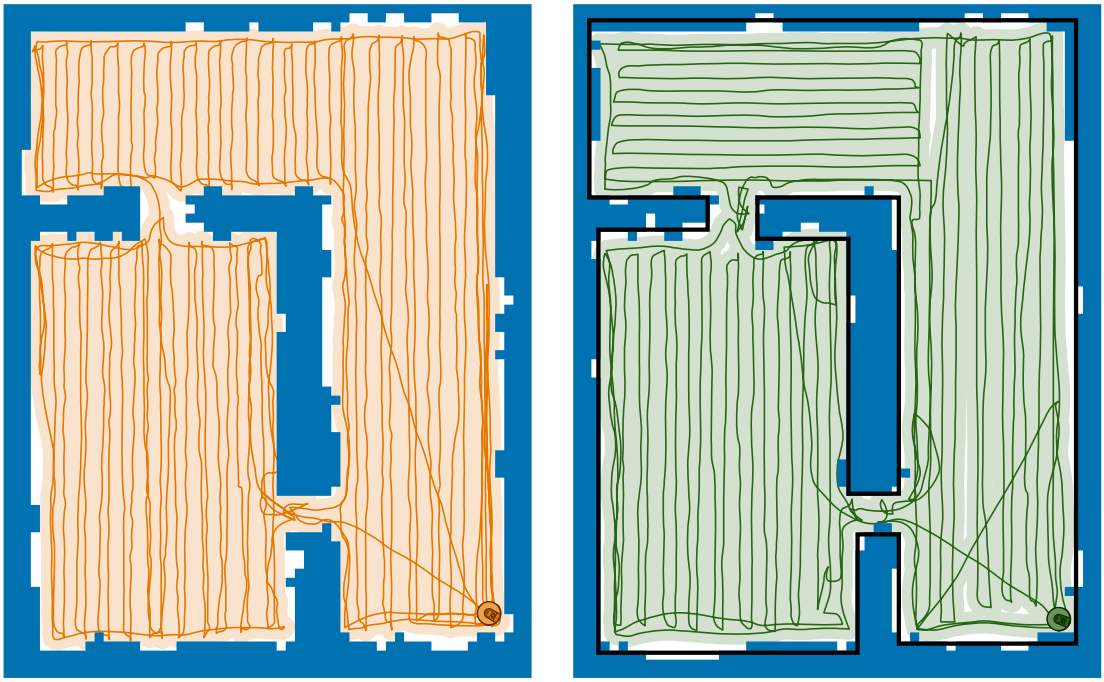


Figure 6.25: Comparison of coverage paths for different coverage strategies on the iRobot Roomba. A previously published Roomba strategy [70] (left) primarily covers in a single direction; my coverage strategy (right) uses a simplified boundary (black) to compute better coverage directions resulting in more efficient coverage.

6.7 Communication during coverage

As for single robot coverage, robust multirobot coverage requires robots to replan in response to unexpected events. In addition to the reasons that single robots need to replan, robot teams also need to replan whenever:

1. One robot is slower than another and the team wants to rebalance the workload;

2. One robot gets stuck or damaged and its tasks must be redistributed to its teammates; or

3. One robot needs to recharge and another robot can perform some of its tasks while it recharges.

As all of these scenarios involve multiple robots, the robots must replan together

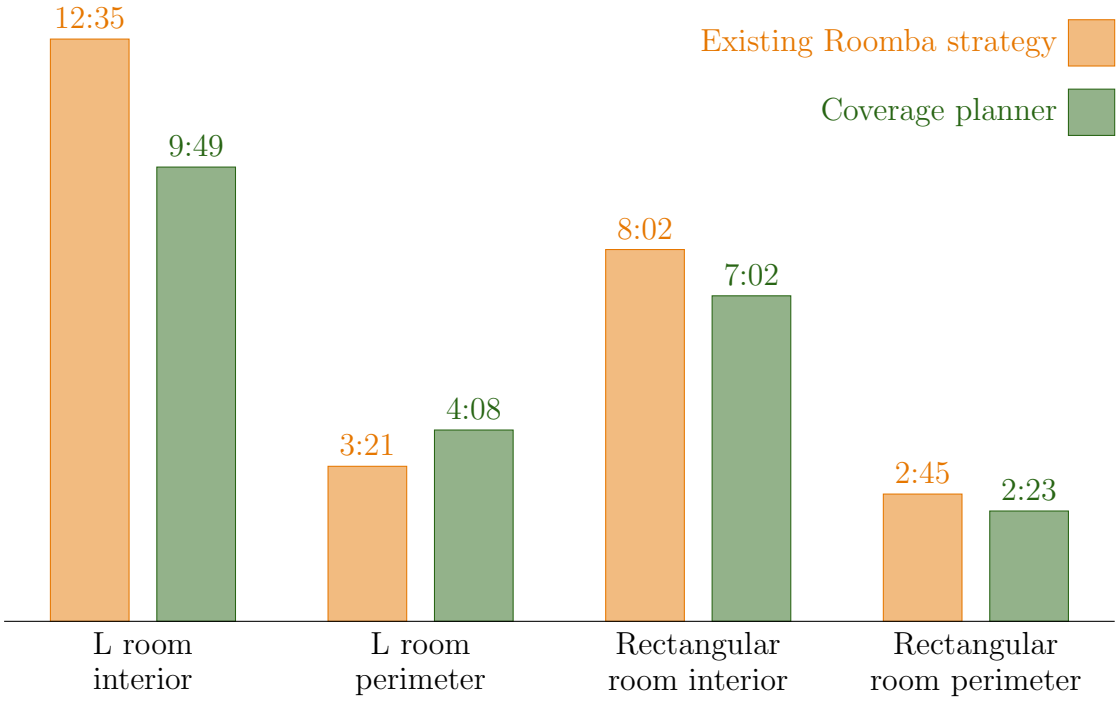


Figure 6.26: Comparison of coverage times for different coverage strategies on the iRobot Roomba. My coverage strategy (green) outperforms the comparison strategy (orange) for interior coverage and has similar coverage times for perimeter coverage.

so they must be able to communicate. Ideally, the robots would replan with the entire team; however, in communication-restricted environments, the team is often disconnected.

At any time during a mission, the team can be divided into several connected *subteams*, which can range in size from a single robot to the entire team. Every robot can communicate—potentially via a multi-hop route—with all the other robots in its subteam and cannot communicate with any robots in other subteams. As coverage paths are based on efficient coverage and not on communication constraints, the structure of these subteams can change regularly and they are often quite small. When a robot needs to replan, its priority should be to continue covering as soon as possible. Therefore, robots should replan opportunistically with their current subteam and only replan with any given robot when they happen to

be close.

Multirobot replanning can be performed in essentially the same way as single robot replanning with two modifications to (Algorithm 6.10). First, the coverage paths are obtained by solving the m-TSP, where m is the size of the subteam, instead of the 1-TSP. Second, the coverage region is only the uncovered parts of the regions previously assigned to robots in the subteam. Any regions assigned to other disconnected regions are not included in the replanning. As replanning only redistributes work within the subteam, the workload may be unbalanced across full team. This lack of balance is especially noticeable when one robot gets stuck or needs to recharge (Figure 6.27). As soon as it is able to communicate with some other teammate, all of its remaining work gets transferred to that teammate, which may end up with twice as much assigned work as the rest of the team.

Algorithm 6.10: Replan with subteam

Input: Subteam's remaining set of coverage paths, $\mathcal{C}$
Output: New coverage plan, $\mathcal{C}'$

```
1  Q_cov ← {}                    /* Remaining coverage region for subteam */
2  for old path p ∈ C do
3      Q_rem ← region that would be covered by path p
4      Q_cov ← Q_cov ∪ Q_rem
5  if map has changed then
6      Add new regions of map to Q_cov
7      Remove parts of Q_cov no longer in map
8  R ← set of perimeter and interior ranks for Q_cov
9  C' ← shortest coverage paths covering R          /* Algorithm 3.4 */
10 return C'
```

As replanning can only happen within a subteam, it is useful to replan whenever a new subteam is formed. Before it forms, some of the robots were disconnected and may have been unable to communicate for a long time. Now that they are able to communicate again, it is likely that they have unbalanced workloads, either due to map changes, differing speeds, or changes in the size of the team. The extra work transferred from a robot needing to recharge will get slowly redistributed

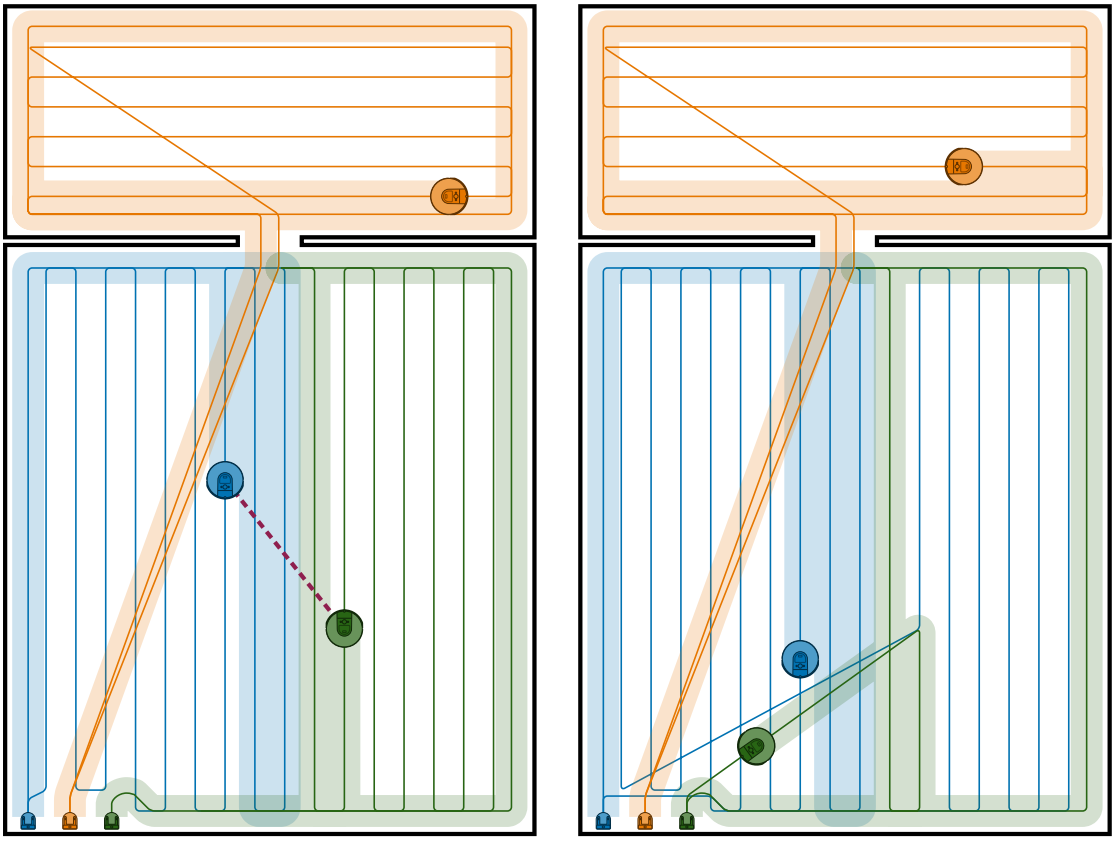


Figure 6.27: In the middle of a coverage mission, one robot (green) is running low battery and needs to recharge. It can currently communicate with one of its teammates (blue) but not the other (orange) and so it only tells this one teammate that it is going to recharge (left). A few seconds later, the blue robot has taken all of the green robots remaining coverage tasks while the green robot heads to its charger (right). The resulting coverage plan for the blue and orange robots covers the entire environment but is not balanced.

amongst the entire team as tasks get transferred to robots with less assigned work every time new subteams form (Figure 6.28).

The necessity of communication in multirobot coverage means that stuck, damaged, and recharging robots can still be useful teammates despite not able to perform any coverage tasks. Stuck and recharging robots are unable to move but may nevertheless function as communication relays helping far apart teammates replan. Additionally, they can be used as a store of information by receiving new information, such as a change to the map, from one robot and later sending that

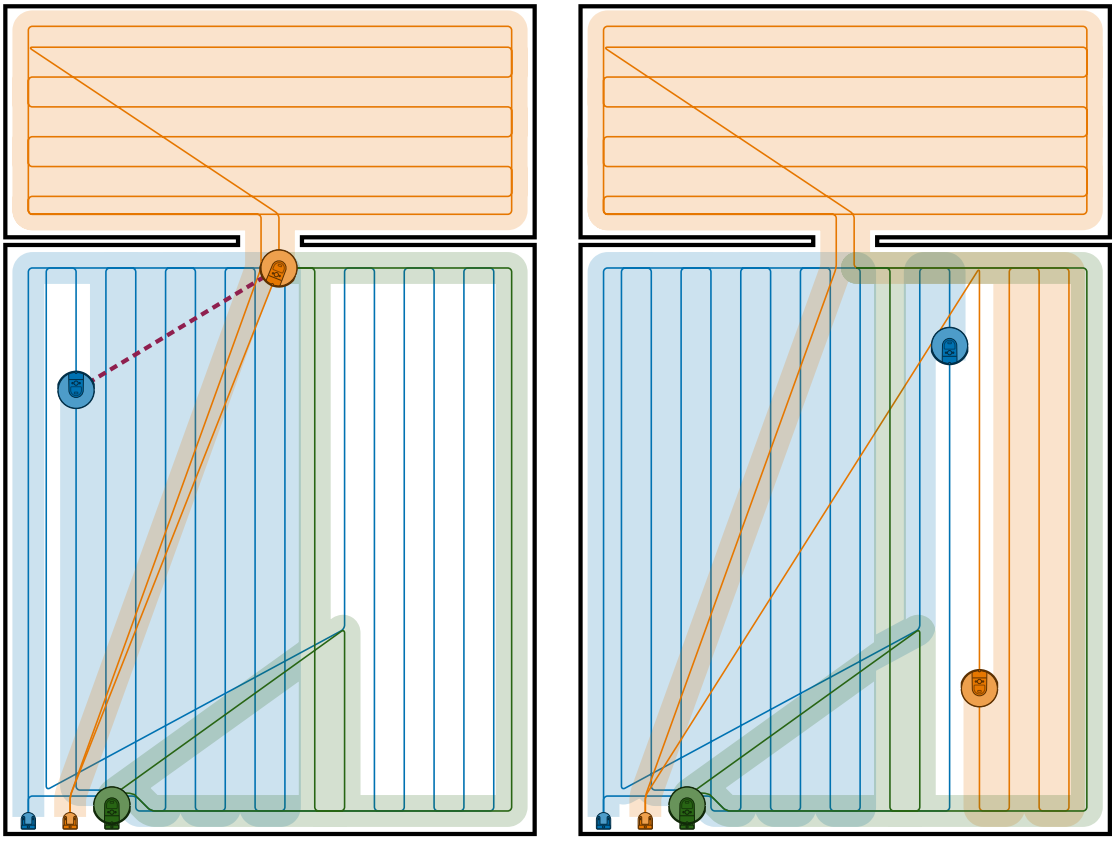


Figure 6.28: One robot (blue) has more remaining coverage tasks than a teammate (orange) because it previously took some coverage tasks from a third teammate (green) that needed to recharge. When the blue and orange robots are finally able to communicate (left), they can replan to rebalance the tasks. A few seconds later, some of the blue tasks have been reassigned to the orange robot and now the team will finish at approximately the same time (right).

information to a second robot despite the first and second robot never being simultaneously connected to the stuck robot. A damaged robot which can move but has a broken coverage tool, can function even better in this role by actively searching for far apart teammates to quickly relay information between them.

6.7.1 Multirobot coverage without communication

While interning at iRobot, I created a simple system for simultaneous multirobot coverage. At the time, we had the infrastructure to coordinate the robots at the

start of their missions through the cloud, but not for coordinating them mid-mission. As a result, my approach was simply to assign each robot a coverage region and then have them cover their regions simultaneously. The coverage regions were determined by first dividing the environment into a set of rooms using a watershed algorithm [105] and then assigning the rooms by solving an instance of the minmax m-TSP (Chapter 3). The graph used for the m-TSP used estimates of the length of path needed to cover each room (its area divided by the rank width), the distance between the centers of the rooms, and the distances from the robots' chargers to the centers of the room.

Using this approach, I ran an experiment where two iRobot Roombas cleaned a previously mapped test environment simultaneously (Figure 6.29). After the initial coordination to assign coverage regions, the robots had no communication. As a result, the two robots could not redistribute their coverage tasks, notify each other of changes to the map, or avoid collisions with each other. Without the ability to redistribute tasks, one robot finished covering its region while the other robot still had quite a bit of work to do. There was also a large region of the map that was covered by *both* robots due to the implementation of the robots' interior coverage behavior (Algorithm 6.3). This implementation only considered the coverage command to be successful after the robots performed several ranks past the edge of the coverage region to compensate for potential localization errors.

In this specific experiment, the overlapping coverage region was covered by the two robots at different times so the robots didn't collide. However, when initially testing the system, there were missions where the robots did collide. Unaware of their teammate, they each treated the collision as if the other robot was an obstacle and would use a behavior to try to go around it. When both robots performed this behavior simultaneously, they would continually collide with each other while moving side-by-side in a direction perpendicular to their planned ranks (Figure 6.30). Meanwhile, both robots were updating their maps based on this

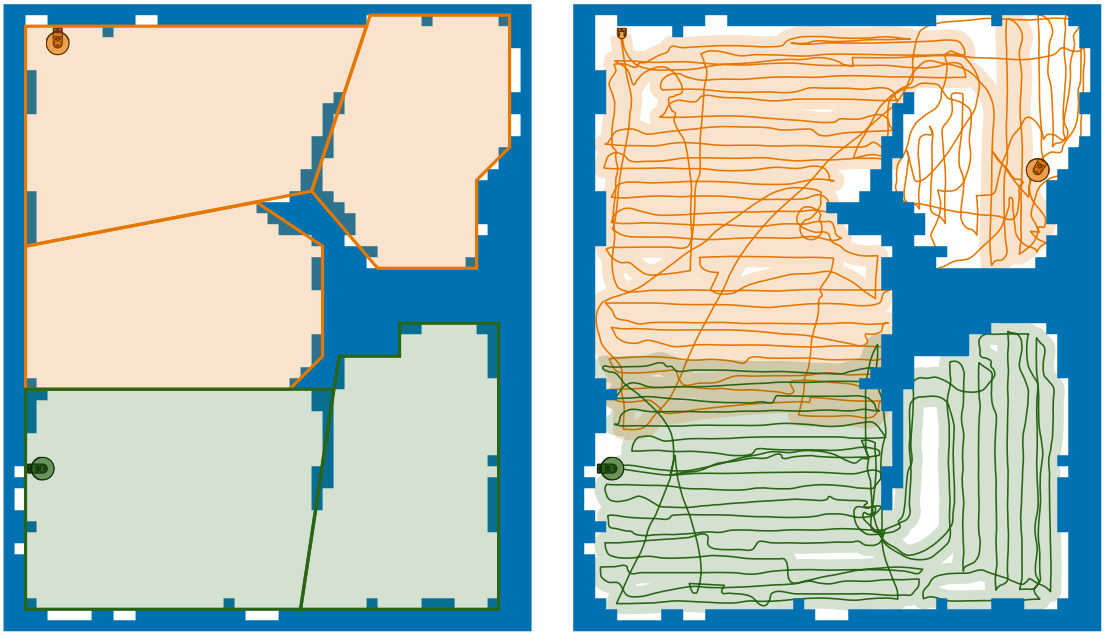


Figure 6.29: Results of a simultaneous coverage experiment using two iRobot Roombas. The test environment is first mapped and divided into several rooms which are assigned based on proximity to the robots' chargers with the intention of balancing the workload between robots (left). The robots covered the environment by simultaneously covering their individual rooms (right). The lines indicate the actual paths taken by the robots and the locations of the robots indicate their actual positions when the faster robot finished coverage of its rooms.

newly discovered "obstacle". Soon both robots believed there to be a long wall in the middle of the environment and were unable to finish their mission because their maps were wrong. This kind of robot-robot interaction highlights the difficulty of simultaneous multirobot coverage and the need for constant coordination. This difficulty is reflected by iRobot's multirobot technology, Imprint™ Link, where a robot vacuum and robot mop share a map but clean consecutively instead of concurrently [61].

6.8 Search and coverage

A robot's coverage plan does not guarantee it will ever be connected to all of its teammates, so sometimes robots need to search for each other to replan. If a

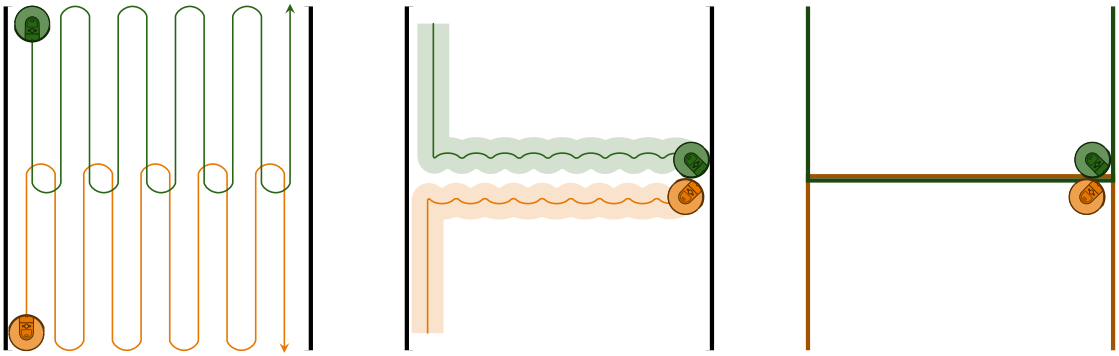


Figure 6.30: When two robots' coverage paths overlap (left) and they try to follow the paths simultaneously, the robots will collide. Both robots will view the other as an obstacle and try to go around it. If each robot tries to go around the other in the same direction, they will end up following each other, colliding repeatedly (center). The end result is that both robots believe there is a wall in the middle of the room (right).

robot still has assigned coverage tasks that it can complete, these tasks should take priority—after all, it may happen to find one of its teammates while doing useful coverage. Once it has finished all of its assigned tasks, or its coverage tool is broken, it should then search for disconnected teammates. By searching, it may find a teammate which hasn't completed all of its tasks, either because it was slower, discovered a new region, got stuck, or needed to recharge. It can then replan with this newly rediscovered teammate to rebalance the workload and help the team finish the mission faster. Search is also necessary to ensure that the entire mission gets completed despite the possibility that a robot may get stuck in a remote location where it cannot communicate with any other robots.

Robots can search for each other using the methods of Chapter 5 where they maintain a belief of their teammates' positions using an HMM and then plan a search path that is likely to find the teammate quickly given that belief. Since the robots are cooperative, they know how their targets behave and can use simulated coverage behavior to create a realistic HMM. Many coverage tasks are repetitive, so after a few missions, these models can be further improved using real data of

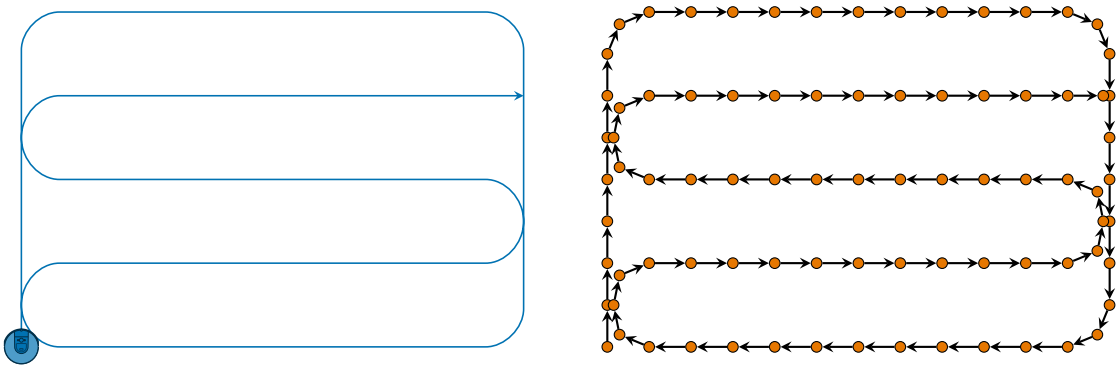


Figure 6.31: A known target path (left) can be incorporated into an HMM by adding a chain of path states (right) along the path with each path state transitioning to the next one along the path.

the robots' actual paths in previous coverage missions. Additionally, the HMMs can be augmented with *path* and *stationary* states to model even more kinds of known behaviors.

6.8.1 Path states

Coverage robots plan their paths in advance and they can share these paths with teammates to help each other search when they get disconnected. A known target path can be incorporated into an HMM by adding a chain of *path* states along that path (Figure 6.31). These path states have two kinds of transitions: a high probability transition to the next path state and some low probability transitions to the nearby direction states to represent the probability of a robot replanning and abandoning its communicated path. These nearby states are determined in the same way as in Figure 5.9 and the transition probability to that state is the product of the convex coefficient for that direction state and the fixed probability of abandoning the path. A chain of transit states can also be added between adjacent transit states to model the variability in the target robot's speed.

6.8.2 Stationary states

Real coverage robots, operating in cluttered home environments with low overhangs and wires, often get stuck. The probability that a robot gets stuck should be reflected in the HMM by the addition of a *stationary* state in each cell of the lattice (Figure 6.32). Each direction state has a small probability of transitioning to that stationary state, corresponding to the probability that the robot gets stuck in that location. However, there are no transitions out of a stationary state because stuck robots never get unstuck without the help of human (usually after the mission is already complete). Initially, the transition probabilities to stationary states are all set at some nominally small value. As the robots continually perform missions in the same environment, they can use data of how often the robot actually gets stuck in each location and update these transition probabilities. In this way, the robots have higher stuck probabilities for dangerous areas, such as underneath a low couch, and very low stuck probabilities for safer areas like the middle of a room.

Stationary states can also be used to model a robot having a low battery. When the robot has a low battery, it will return to its charging station or the nearest charging station if there are multiple. There should therefore be one stationary state per charging station and every direction state should have transitions to the nearest such stationary state. As the probability of a low battery depends on the length of the mission and not the robots' positions, these transition probabilities should all be the same but they should increase with time. To make the transitions to charging states more realistic, a chain of path states along the shortest path from each lattice cell to its nearest charger can be added in between the direction and charging states. Including these additional path and stationary states model the reality that after a long mission the target robot is most likely to be found at a charger.

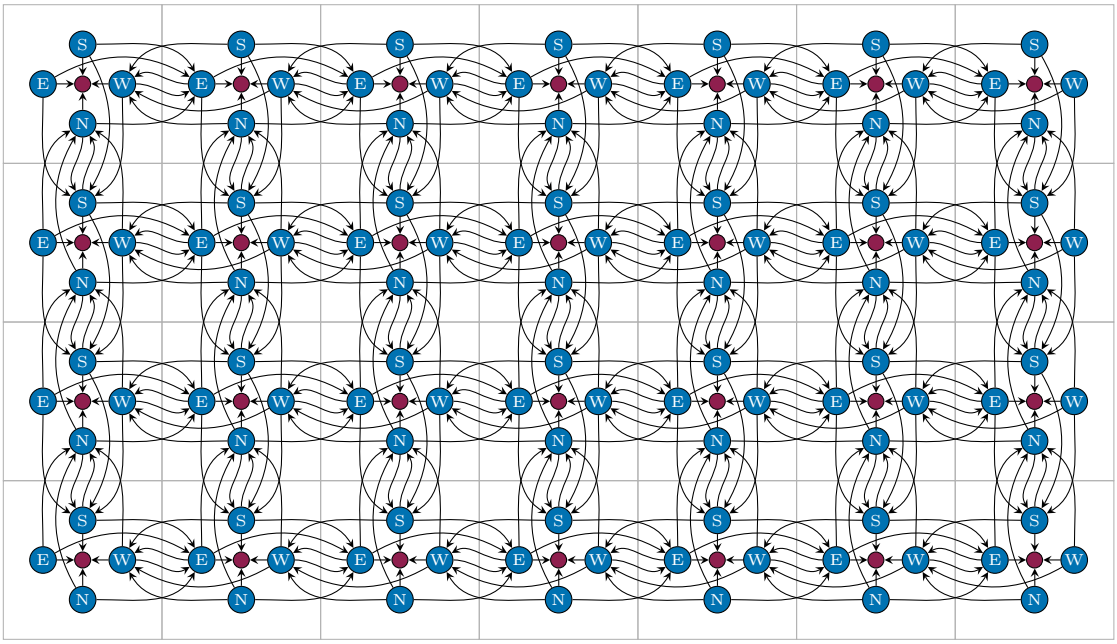


Figure 6.32: An HMM can include one stationary state (red) for each lattice cell to model the fact that a robot can get stuck. These states do not have any transitions out, representing the fact that a stuck robot never moves.

6.8.3 Overall layered HMM for search

The most complex form of HMM consists of 4 different kinds of states: path, direction, transit, and stationary. These states can be arranged into 3 layers with path states on top, direction states in the middle, and stationary states at the bottom (Figure 6.33). Chains of transit states can be added between adjacent path states in the top layer and between adjacent direction states in the middle layer to model the target's variable speed. Initially, the belief's probability is concentrated in the top layer because the target was seen very recently so the searcher is quite confident that the target is still following the known path. As path states can transition to direction states but direction states do not transition to path states, the probability density will slowly transition to the middle layer. The middle layer, consisting of direction states, represents a target that is moving, but has been disconnected for long enough that the searcher doesn't know its current

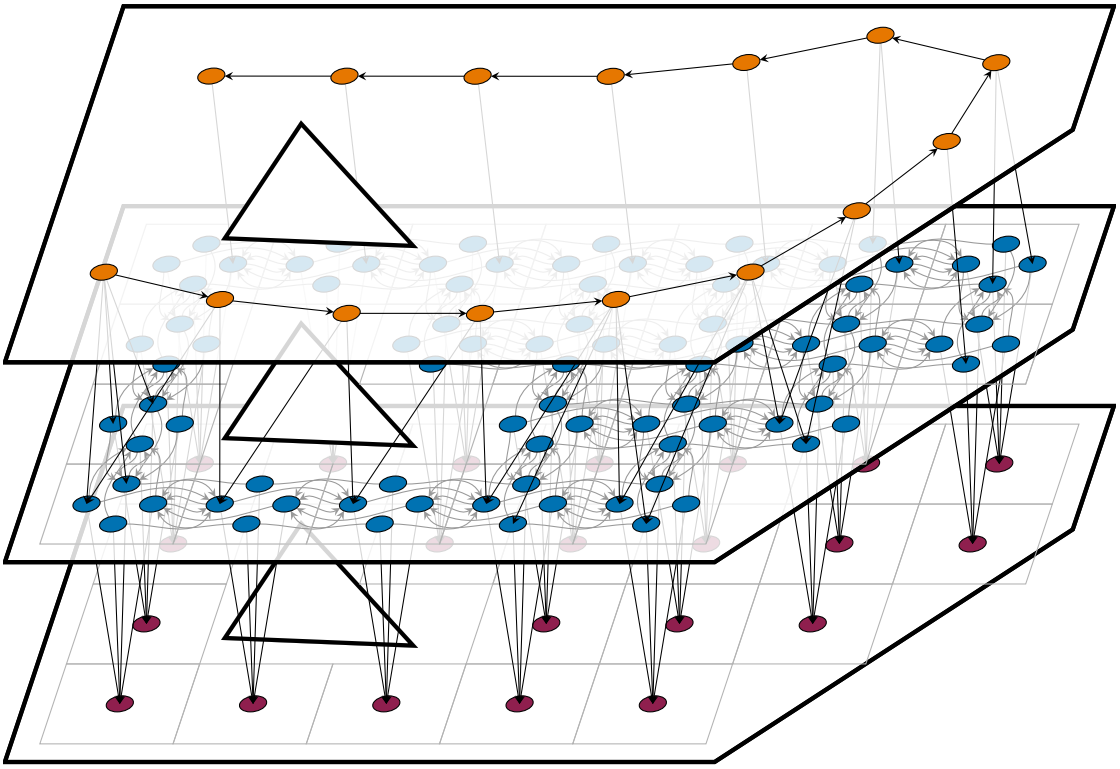


Figure 6.33: The overall HMM consists of three layers: path states (orange) in the top layer, direction states (blue) in the middle layer, and stationary states (red) in the bottom layer. It additionally contains chains transit states (not shown) between pairs of path states in the top layer and between pairs of direction states in the middle layer. As time progresses, the probability density transitions from the top layer to the middle layer to the bottom layer.

behavior and instead uses a generic model based on simulated or historic data. Similarly, the direction states can transition to stationary states but stationary states cannot transition to any other states. Therefore, after a very long time, all the probability density will transition to the bottom layer, representing the fact that a target which has not been seen in a very long time is probably either stuck or charging.

This kind of model could be further augmented with additional middle layers if the target's behavior is likely to change through out the mission. For this problem, there could be two middle layers: the upper middle layer using transition probabilities learned from historic coverage paths and the lower middle layer using

transition probabilities learned from historic search paths. This additional structure reflects the fact that the target robot may itself be searching for other robots near the end of the mission. Similar approaches with multiple layers could be used for any kind of mission where individual robots have multiple different modes of behavior with different modes being more common at different times throughout the mission.

6.9 Robust multirobot coverage

From the perspective of a single robot in a team, a coverage mission is only successful once it knows that every part of the environment has been covered by at least one robot. Regions of a robot's map can be marked as covered by either:

(a) Covering that region itself; or

(b) Communicating with another robot that has covered that region.

The benefit of multirobot coverage is that missions are completed faster than a single robot could do alone, so robots must ensure they communicate with each other by the end of the mission so that every robot knows the mission is complete.

Robust multirobot coverage, therefore consists of individual robots covering their assigned regions, communicating whenever possible, and searching for each other near the end of the mission (Algorithm 6.11). If a robot still has uncompleted coverage tasks assigned to it, it performs a multirobot version of robust coverage using coverage behaviors to complete its assigned tasks and replans with nearby robots in the event of a kidnapping, map change, or other unexpected event. Only after a subteam has completed all of its tasks does it search for the remaining disconnected robots. Once it finds another robot, it forms a new subteam and can again replan, splitting any remaining tasks to finish the mission faster. As this process repeats, the subteams tend to grow larger as previously distant robots all

end up in the same vicinity of the remaining uncovered regions. Eventually all working robots are aware that all tasks have been completed and they can return to their charging stations. As long as at least one robot can still cover, this method guarantees that the mission will get completed.

Algorithm 6.11: Robust multirobot coverage

Input: Environment to be covered, team of robots
Output: Covered environment

```
1  Plan coverage with connected robots                    /* Algorithm 4.7 */
2  while coverage mission is not complete do
3      if change in subteam, map, or plan execution then
4          Replan with subteam                            /* Algorithm 6.10 */
5      else if has remaining assigned coverage tasks then
6          Continue executing tasks                       /* Algorithms 6.3 or 6.4 */
7      else if there are disconnected robots then
8          Search for disconnected robots                 /* Algorithm 5.2 */
9      else
10         Mission complete
11     Share current position and plan with connected robots
12     Update beliefs about teammates                     /* (5.11) or (5.14) */
13 Return to charging station                             /* Algorithm 6.1 or similar */
```

6.9.1 Results

To test my approach to robust multirobot coverage, I ran some simulations comparing two coverage strategies. In these simulations, two robots cover a large environment simultaneously. Their initial plans are based on the assumption that both robots travel at the same speed, however, in reality one robot is much slower than the other. The robot's speed is randomly selected between 50% and 95% of the expected speed representing a systematic mechanical problem such as a weak motor or excess friction in an axle. As I have previously considered other sources of uncertainty that affect both single and multirobot coverage (see Section 6.6), these simulations assume the robots can follow their planned paths perfectly, the real environment is identical to the map, the robots cannot get stuck, and there are

no humans to interfere with the robot. Therefore the only sources of uncertainty are ones which only apply to multirobot coverage: differences between teammates' speeds and limited communication ability.

The baseline strategy is for the two robots to both cover their assigned paths despite the differences in the robots' speed. Using this strategy, the fast robot will finish covering its path at the expected time and then will simply wait at its charger for the slow robot to finish covering its path. The overall mission time will therefore equal the time it takes for the slower robot to finish its path, which may be much slower than the fast robot.

When using the robust strategy (Algorithm 6.11), the robots have the same initial plan, but they are able to communicate during the mission and replan. When the fast robot finishes its initial plan, the slow robot will inevitably still have many remaining regions to cover (Figure 6.34). To help the slow robot finish its remaining tasks, the two robots need to be able to communicate so they can replan. However, as they need to be near each other and have a line of sight to communicate, the fast robot will have to search for the slow robot. Using its belief, based on its knowledge of the slow robot's plan and where it has seen the slow robot, the fast robot creates a search tree for planning a search path (Figure 6.35). It follows the best search path, growing the search tree as necessary, until it finds the slow robot. Once the two robots are connected, they replan coverage paths for the remaining uncovered region (Figure 6.36). This new plan is based on their actual average speeds since the start of the mission (see Section 3.6 for planning for heterogeneous robots), so the fast robot gets assigned a longer path and the two robots will finish approximately at the same time.

I ran 300 simulations of the fast and slow coverage robots using both the basic and the robust coverage strategies (Figure 6.37). In addition to varying the slow robot's actual speed from 50% to 95% of the expected speeds, I also varied the locations of the robots' chargers—always located side-by-side—between several

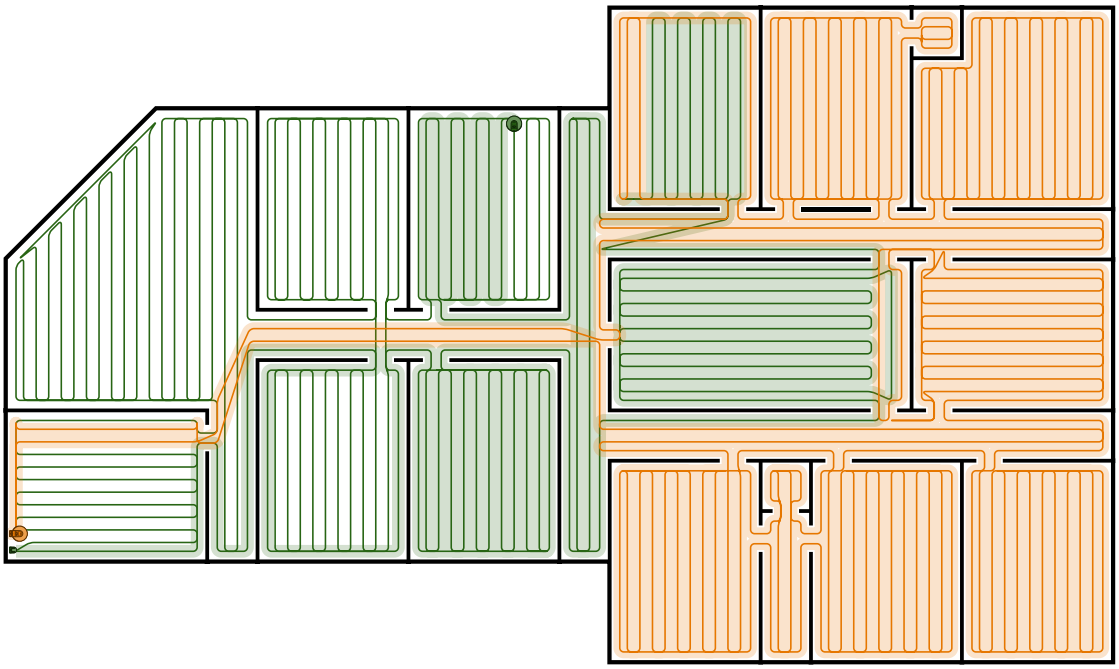


Figure 6.34: Progress along planned coverage paths for two simulated robots. One robot (green) is slower than expected and still has a large part of its path remaining when the faster robot (orange) has finished.

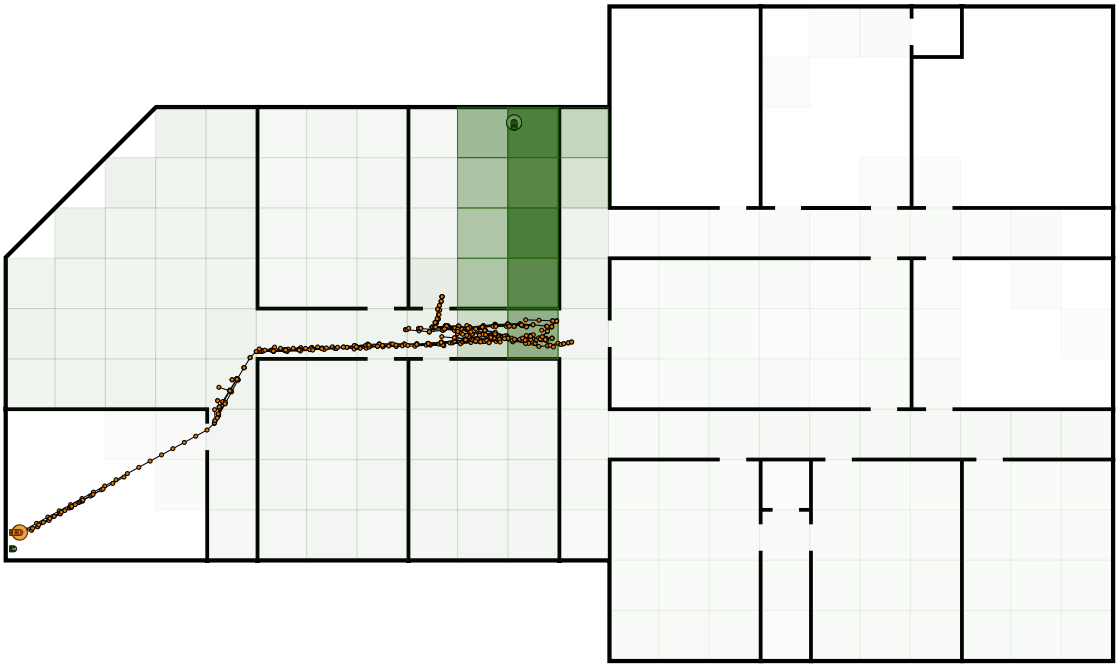


Figure 6.35: Fast robot's (orange) belief of its slower teammate's (green) position when the robots are in the positions in Figure 6.34. The fast robot uses this belief to construct a planning tree to plan a search path to reconnect with the slow robot.

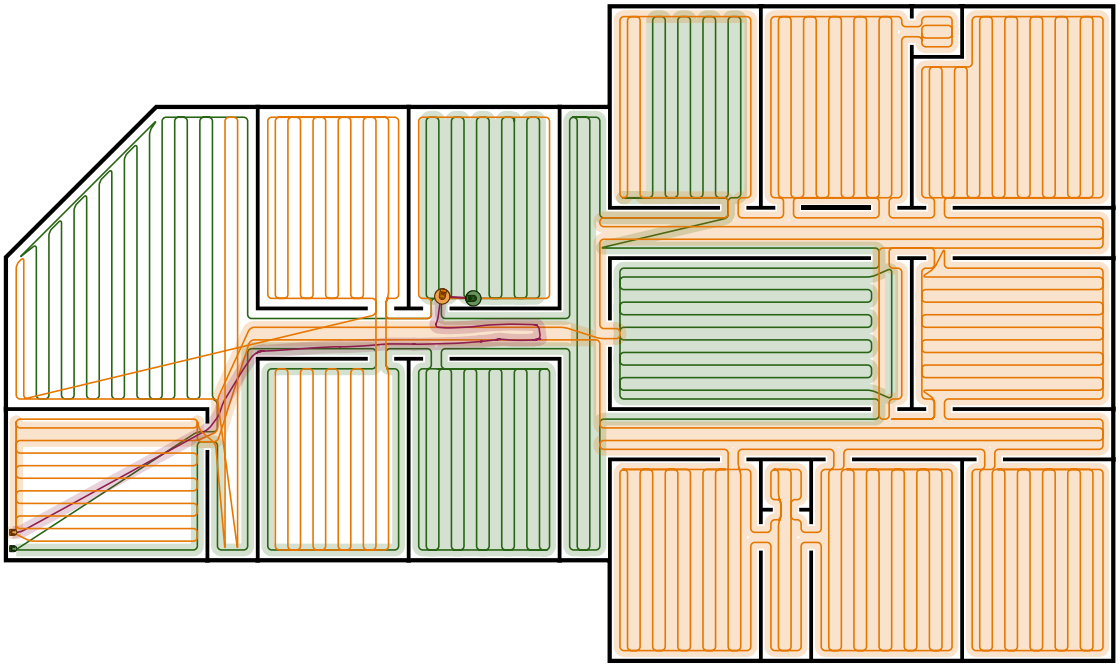


Figure 6.36: The fast robot (orange) follows a search path (red) based on the planning tree in Figure 6.35 to find the slow robot (green). Once the robots are reconnected, they replan their coverage paths based on their previous speeds to balance the workload and finish the coverage mission faster.

different corners of the environment. In almost all cases, the robust strategy was faster than the basic strategy. When the slow robot is almost as fast as expected, the difference is relatively small—the robots are behaving similar to expected so the initial plan is already quite good. When the slow robot is much slower, the robust plan often results in much better performance, often taking only slightly longer than the theoretical minimum time (assumes no duplicate coverage and the robots finish simultaneously). In other cases when it takes the fast robot a long time to find the slow one, the robust strategy is not much better than the basic strategy. The HMM used to update the fast robot's belief of the slow robot's position is biased towards locations near the chargers at the end of the mission. Therefore, the fast robot can easily find the slow robot near the end of the mission just by staying near the charger, so the robots almost always find each other before the slow robot finishes or as it is finishing its path. The result is that the robust

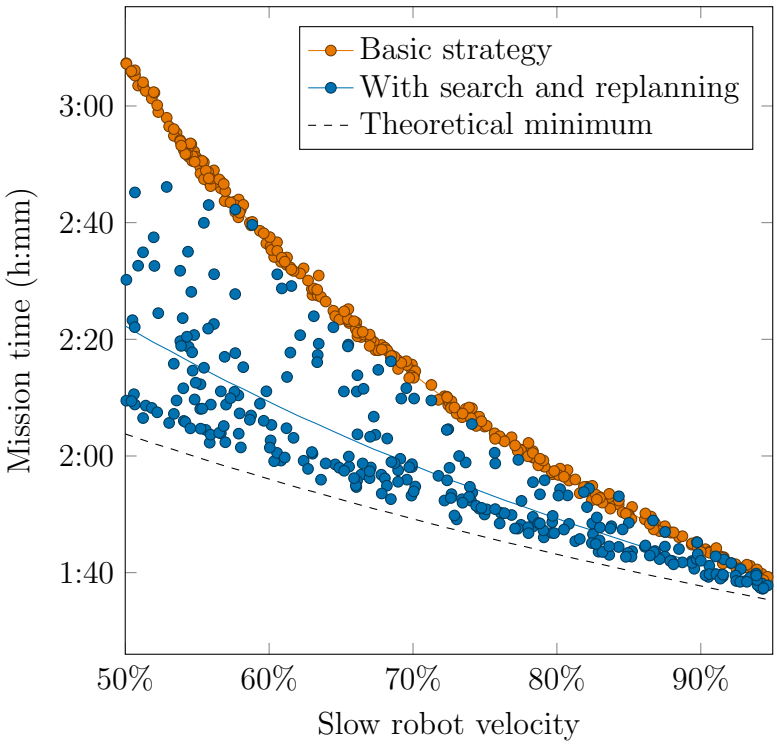


Figure 6.37: Comparison of multirobot coverage times using basic (orange) and robust (blue) strategies. Each dot represents one of 300 coverage mission which each have a randomly selected velocity for the slow robot (between 50% and 95% of the fast robot's velocity) and a randomly selected depot location. The theoretical minimum is based on the required coverage if both robots finish simultaneously and there is no duplicate coverage.

strategy's use of search either improves the mission time (if the fast robot finds the slow one before it finishes), or has no effect (if the fast robot finds the robot as it is finishing). The improvement in mission time can be especially large in cases where one robot is much slower than expected.

6.10 Conclusions

Performing coverage on real robots is often much more difficult than simply telling each robot to follow a planned coverage path. In the real world, robots must deal with incorrect maps, human interference, localization errors, low batteries, and hazards that can damage or trap the robot. All of these sources of uncertainty

make it difficult or impossible for the team to complete a coverage mission with the naïve strategy of simply following a single set of paths produced by a coverage planner like the one in Chapter 4. Instead, the team uses a robust strategy where the planned paths are converted into semantic commands, subteams replan when necessary, and robots search for teammates near the end of the mission.

Semantic commands such as "go to the corner of the room" or "cover the hallway by ranks" are more meaningful than simply telling the robot to go to a precise location. These commands are completed by behaviors which use the robot's sensors to determine whether it has completed the behavior, even if it did not reach the exact coordinates expected by the planner. Using semantic commands and behaviors makes the robot robust to localization errors and offloads the responsibility for low level tasks—navigating around small obstacles or along irregularly shaped walls—to the robot's behaviors. Since these tasks do not affect coverage planning, the map of the environment is also processed and simplified based on the robot's behaviors before it is used for coverage planning.

Replanning is often necessary when the team cannot complete its current coverage plan. There are many reasons for replanning including humans physically moving a robot, changes in the map when a door opens or closes, and a robot needing to recharge when it runs out of battery. If the map has not changed, the robots can replan by simply planning new paths—through solving the m-TSP (Chapter 3)—to cover the remaining set of uncovered ranks based on where the available robots are currently located. If, however, the map has changed, the robots must additionally compute a new rank partition for the uncovered parts of the current map. As limitations of wireless communication technologies restrict robots' abilities to communicate over large distances or through walls, teams of robots are only able to replan with nearby connected robots. At the end of a mission, when a robot has no remaining coverage tasks, it can search (Chapter 5) for lost teammates and replan with them once they reconnect so that the team

finishes the overall mission faster. Similarly, damaged robots which can move but no longer perform coverage can repeatedly search for different teammates, spreading information throughout the team and serving as a valuable teammate despite being unable to perform any coverage tasks.

I tested these approaches to robust coverage through several experiments, both simulated and performed on real robots at iRobot. For single robot coverage, I found that a robust coverage strategy using a plan produced by my coverage planner and executed through semantic behaviors outperformed the existing strategy used by the commercially successful iRobot Roomba robotic vacuum cleaner. For multirobot coverage, I showed, in a real world experiment, that coverage using a basic strategy without coordination is generally faster than single robot coverage, but has two major flaws. The robots are unlikely to finish at the same time so they overall mission takes longer, and they also can interfere with each other if they collide when covering within close proximity with each other. Using a robust strategy, I showed the effectiveness of using search to ensure that all robots finish at approximately the same time by rebalancing the workload during the mission. The combination of these results is a multirobot coverage strategy that is robust to changes in the environment and in the team and will take advantage of every robot in the team to finish its mission as quickly as possible.

Chapter 7

Conclusion

As robots become more common in society, they are increasingly likely to be working in teams with other robots and within environments designed for and shared with humans. Just as planning is important for individual robots working in restricted, specialized environments, it is also vital for coordination of these robotic teams. The two main differences between multirobot and single robot planning are:

1. Teams must decide which robot does which task instead of one robot performing all tasks; and

2. Cooperative planning is only possible if robots can communicate so they must plan how they will communicate.

In real environments, with many sources of uncertainty, planning is not sufficient to complete some complex mission and the team must adapt and respond to the unknown to successfully execute their plan, or something similar to it.

In this thesis, I presented several algorithms to solve the multirobot problems of task allocation (Chapter 3), coverage (Chapter 4), and search (Chapter 5). The algorithms are analyzed theoretically and evaluated using a combination of simulations and experimental results, including results combining the three problems (Chapter 6). These algorithms outperform existing algorithms intended to solve the same problems and can be used for by teams of robots whereas previous algorithms were often limited to a single robot. Furthermore, the algorithms are

computationally efficient enough to run on commercially available robots, such as the iRobot Roomba vacuum cleaner, a platform I used when testing the algorithms. Collectively these algorithms solve important multirobot planning problems, bringing teams of cooperative robots closer to being a reality in everyday society.

Coverage is one of the most common tasks performed by robots today. Many successful consumer robotics—vacuums, pool cleaners, lawnmowers, and window cleaners—all perform versions of coverage. In coverage, each robot has a tool which it must pass over every point of its environment. A coverage mission is complex and can be solved in many different ways depending on which robot covers which region, how each robot covers its assigned region, and how the team responds to the unexpected. Despite its complexity, consumers expect coverage robots to work quickly, reliably, and intelligently in any environment they are placed in. Its commercial relevance and complexity make coverage an excellent example of a complex mission which requires multirobot teams to successfully assign tasks and communicate sufficiently often.

Deciding which robot does what is the problem of *task allocation*. Solving this problem first requires some large overall mission to be divided into a set of small individual tasks which can each be completed relatively quickly by a single robot. To complete the mission as quickly, these tasks should be assigned to the robots in a balanced way so that every robot finishes its tasks at approximately the same time and the slowest robot—which determines the overall team's speed—finishes as soon as possible. For robotic tasks, which are spatially distributed, the order of tasks affects how long it takes each robot to complete its tasks, so the overall problem is both an allocation and *routing* problem.

Solving the combined allocation and routing problem is equivalent to solving

the minmax multiple travelling salesperson problem (Chapter 3). Although allocation and routing are usually seen as highly coupled problems, there is an approximately monotonic relationship between the best possible time a set of tasks can be completed in—obtained by solving an NP-hard routing problem—and the average time needed to complete the same set of tasks (Subsection 3.3.1). The cost function of the minmax m-TSP uses a maximum function, which behaves well with monotonic functions as they preserve inequalities. Therefore, the average time, which is much easier to compute, can be used as a proxy for the minimum time needed to compute a set of tasks. Then, the set of tasks can be quickly partitioned (Section 3.4) and the order that each robot performs its tasks can be computed separately (Section 3.5) with only a few small changes needed to rebalance the assigned tasks once the best routes have all been determined. The algorithm based on this idea was able to outperform two state-of-the-art m-TSP algorithms on standardized large scale problems involving over 1000 tasks and up to 100 robots (Section 3.9). Runtimes for this algorithm scale approximately quadratically with the number of tasks and do not depend heavily on the number of agents, making it applicable to problems with many tasks and large teams.

For the overall task of multirobot coverage, individual tasks should be defined based on the robots' basic behaviors. The simplest behaviors of a coverage robot is travelling along a *rank*—either a long straight line in an open area or a curve along the perimeter of the environment. Coverage of ranks is efficient. Robots can cover a rank near their maximum speed as they only need to move in a straight line. Transitioning between ranks, however, is much slower. Robots need to make time consuming turns and often pass over regions that have already been covered when travelling from the end of one rank to the start of the next. In addition to ordering ranks to minimize the lengths of these redundant connecting paths, coverage can be made more efficient by choosing the directions of ranks to minimize the number of turns that the robots will have to make.

In turn-minimizing coverage (Chapter 4), the environment is first partitioned into a set of interior and perimeter ranks which are then allocated to individual robots by solving the m-TSP. The interior ranks can be horizontal or vertical and are chosen to cover a rectilinear polygon (Subsection 4.2.2), which fully covers the interior of the environment. This rectilinear polygon is sliced at each of its concave vertices to create a checkerboard partition (Subsection 4.2.3) consisting only of rectangles which can each be covered by one direction of interior ranks. The coverage direction for each rectangle is determined using a heuristic which iteratively changes the orientations of rectangles based on their neighbors' orientations (Subsection 4.2.4). Once the directions of each rectangle has been determined, adjacent rectangles are merged and then sliced into the interior ranks. The entire set of ranks—including trivially defined perimeter ranks—are then used to obtain coverage plans for each robot by solving a constrained version of the minmax m-TSP. Using turn-minimization, coverage paths for real indoor environments mapped by the iRobot Roomba had on average 6.7% fewer turns than optimal paths obtained with only one direction of interior ranks (Section 4.4). Compared to strategies currently used by commercial coverage robots, which do not use turn-minimization or optimize the order of ranks, this optimized strategy is much faster.

Communication constraints can have a great impact on robots' abilities to function as a team. Coordinated planning is only possible between robots that can communicate, but robots equipped with inexpensive wireless communication devices often cannot communicate over large distances or through walls. Unexpected circumstances which disrupt a robot's plan and force it to replan can also cause it to get disconnected from its teammates. Even if the team had previously planned a rendezvous, in real world environments it is impossible to guarantee that they will actually be able to reconnect in any specific time and place. Therefore, robots must be able to search for their teammates when cooperating in real, unpredictable environments. Search also enables robots to separate as needed to complete some

complex individual tasks and then flexibly reconnect to share information and replan.

An effective search strategy uses knowledge of a target robot's likely behavior and observations of where the robot is not located to maintain a *belief* of where the robot might be and then uses this belief to plan a search path (Chapter 5). This belief is modeled as a hidden Markov model (Section 5.3) which uses a distribution over a set of hidden states to model the robot's semantic behavior, momentum, and variable velocity and can be converted to a distribution over physical space. Additionally, historic or simulated data about the target's behaviour can be used to determine the model's transition probabilities (Subsection 5.3.4) and additional states can be added if the target's plan is known (Subsection 6.8.1) as it often is in a cooperative mission. The belief is further improved using the searcher's observations (Section 5.4) both of when it sees its target and when it doesn't. Any robot can maintain a belief of another—even if it is not actively searching for that robot—and two nearby robots can merge their beliefs about a third robot when they communicate (Section 5.5). Once a robot decides it needs to search for a lost teammate, it constructs a tree of possible search paths using randomly sampled locations (Section 5.7) and evaluates the candidate paths using the probability of finding the target—based on its belief—after different lengths of time. The resulting search algorithm is effective for finding disconnected robots, with similar best-case-scenario search times when compared with two baseline strategies but much better search times in the worst-case-scenarios (Section 5.8).

Real robots' behavior must be robust and adaptive (Chapter 6) because real environments are dynamic and unpredictable. For the task of coverage, robust behavior is achieved through semantic commands, replanning, and—for multirobot coverage—search. Semantic commands (Section 6.3) and the corresponding behaviors used to complete them use real-time sensor data to accomplish the objective of a plan, such as covering a large region of a room, rather than the trying to follow

the exact path specified by the plan. Replanning (Section 6.5) is necessary in many unexpected circumstances—a low battery, human interference, or the discovery of a new room—to adapt a coverage plan to new information. A coverage strategy planned by the algorithms of Chapter 4 and executed via semantic commands with replanning when necessary is robust enough to run successfully on the iRobot Roomba in a realistic environment and the efficiency of planned turn-minimizing paths results in faster coverage than the Roomba's current strategy (Section 6.6). For multirobot coverage, robots which do not coordinate throughout a mission are unlikely to finish simultaneously (Subsection 6.7.1) resulting in a mission which takes longer than is necessary. If the faster robots search for slower robots once they have finished their assigned coverage tasks (Section 6.8), the team can replan and guarantee they all finish at similar times. When search is combined in a robust multirobot coverage strategy (Subsection 6.9.1), the team compensates for differing speeds and completes the overall mission almost as fast as possible despite originally planning based on incorrect information.

Using solutions to multirobot-specific problems, such as search and task allocation, teams of robots can quickly complete tasks that would normally be performed slowly by a single robot. Although coverage is an example of one such task, other tasks, such as delivery, surveillance, and exploration, could also benefit from the same search and task allocation algorithms. As long as a large task can be divided into smaller ones, a minmax task allocation algorithm can be used to fairly divide the small tasks amongst the team. When completing these tasks, inevitable interruptions forcing the team to separate but requiring them to replan are not a problem as robots can search for teammates whenever necessary. With these coordination methods, the team is able to cooperate effectively despite unexpected circumstances and teams of robots become truly effective ways of solving problems in the real world quickly and robustly.

7.1 Future work

Although the algorithms presented in this thesis were presented in a general form that makes them applicable to a large class of robots, there are some limitations due to some of the assumptions made about the robots and their environments. None of the algorithms in this thesis explicitly consider motion constraints and instead assume that the robots can freely navigate to any location in their environment. This assumption is generally true for the scenarios encountered by today's commercially available coverage robots where a single robot or small team of robots is operating in a large environment. However, in heavily cluttered environments or when many robots share an environment, robots may be blocked by obstacles or other robots and unable to freely navigate. In these situations, the algorithms presented in this thesis may be significantly less effective. Similarly, they may be inadequate for robots with nonholonomic constraints which are unable to freely navigate due to their inherent mechanical constraints. The algorithms in this thesis were also mainly developed for ground-based robots which only move in two-dimensions and may not be fully applicable to underwater or aerial robots which move in three dimensions.

These limitations present opportunities for future research. The more robots there are in a given environment, the higher the chance of collisions. Therefore an interesting extension of the search and coverage algorithms presented in this thesis would be collision-aware planning where the planning algorithms are guaranteed to result in collision free paths. Such a system would also need to be able to resolve potential collisions mid mission if two robots are likely to collide because they were unable to follow their plans exactly. Similarly, another interesting extension would be to explicitly consider the robot's kinematics when planning. By actively considering kinematics the planner could guarantee the path is feasible for the robot's actual hardware and could also be used to minimize energy expenditure

instead of time.

Other interesting directions for future research are semantic planning and the relationship between planning and mapping. Although semantic commands were mentioned in Section 6.3, they were treated as a final step in where a polygonal path gets converted into semantic commands. An alternative approach to planning would be to generate high level semantic commands directly instead of a polygonal plan. These commands would then be executed by lower level semantic behaviors which use feedback between sensors and actuators to ensure the does what it was instructed to do. Such an approach would likely require a different description of the robot's environment—a semantic map. Currently, robots typically use detailed occupancy grid maps which attempt to describe the local properties of every part of a robot's environment. These maps do not however, represent how different parts of the environment relate to each other and high level concepts that humans think in terms of—walls, streets, rooms, neighborhoods—are absent from the map. If the maps will be used for planning, and the plans are represented in terms of high-level semantics, a map that contains or is entirely based on these same semantics and the relationships between them would result in better plans. Both semantic plans and semantic maps would also have the advantage of being easier for humans to interpret, making it easier for us to interact with these robots, greatly aiding their adoption into society.

Bibliography

[1] E. Acar, H. Choset, A. Rizzi, P. Atkar, and D. Hull, "Morse decompositions for coverage tasks," *The International Journal of Robotics Research*, vol. 21, no. 4, pp. 331–344, 2002.

[2] N. Agmon, N. Hazon, and G. Kaminka, "The giving tree: Constructing trees for efficient offline and online multi-robot coverage," *Annals of Mathematics and Artificial Intelligence*, vol. 52, no. 2, pp. 143–168, 2008.

[3] M. Ahmadi and P. Stone, "A multi-robot system for continuous area sweeping tasks," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2006, pp. 1724–1729.

[4] S. Alpern, "Rendezvous search: A personal perspective," *Operations Research*, vol. 50, no. 5, pp. 772–795, 2002.

[5] S. Alpern and S. Gal, "Searching for an agent who may or may not want to be found," *Operations Research*, vol. 50, no. 2, pp. 311–323, 2002.

[6] R. Alves and C. Lopes, "Using genetic algorithms to minimize the distance and balance the routes for the multiple traveling salesman problem," in *Congress on Evolutionary Computation (CEC)*. IEEE, 2015, pp. 3171–3178.

[7] F. Amigoni, J. Banfi, and N. Basilico, "Multirobot exploration of communication-restricted environments: A survey," *IEEE Intelligent Systems*, vol. 32, no. 6, pp. 48–57, 2017.

[8] E. Anderson and R. Weber, "The rendezvous problem on discrete locations," *Journal of Applied Probability*, vol. 27, no. 4, pp. 839–851, 1990.

[9] N. Anderson, "System and method for area coverage using sector decomposition," Jul. 17 2012, US Patent 8,224,516.

[10] C. Angle, D. Snelling, M. O'Dea, T. Farlow, S. Duffley, J. Mammen, and M. Halloran, "Mobile robot providing environmental mapping for household environmental control," Jan. 12 2016, US Patent 9,233,472.

[11] J. Antonio, R. Ramabhadran, and T. Ling, "A framework for optimal trajectory planning for automated spray coating," *International Journal of Robotics and Automation*, vol. 12, pp. 124–134, 1997.

[12] D. Applegate, W. Cook, and A. Rohe, "Chained Lin–Kernighan for large traveling salesman problems," *INFORMS Journal on Computing*, vol. 15, no. 1, pp. 82–92, 2003.

[13] M. Arif and S. Haider, "An evolutionary traveling salesman approach for multi-robot task allocation." in *International Conference on Agents and Artificial Intelligence (ICAART)*. IEEE, 2017, pp. 567–574.

[14] E. Arkin, S. Fekete, and J. Mitchell, "Approximation algorithms for lawn mowing and milling," *Computational Geometry*, vol. 17, no. 1-2, pp. 25–50, 2000.

[15] G. Avellar, G. Pereira, L. Pimenta, and P. Iscold, "Multi-UAV routing for area coverage and remote sensing with minimum time," *Sensors*, vol. 15, no. 11, pp. 27 783–27 803, 2015.

[16] T. Baker, J. Gill, and R. Solovay, "Relativizations of the P=?NP question," *SIAM Journal on computing*, vol. 4, no. 4, pp. 431–442, 1975.

[17] P. Balutis, A. Beaulieu, B. Yamauchi, K. Karlson, and D. Jones, "Robot lawnmower mapping," Aug. 23 2016, US Patent 9,420,741.

[18] T. Bandyopadhyay, N. Rong, M. Ang, D. Hsu, and W. Lee, "Motion planning for people tracking in uncertain and dynamic environments," in *International Conference on Robotics and Automation (ICRA), Workshop on People Detection & Tracking*. IEEE, 2009.

[19] N. Banerjee, R. Connolly, D. Lisin, J. Briggs, M. Narayana, and M. Munich, "View management for lifelong visual maps," *arXiv preprint arXiv:1908.03605*, 2019.

[20] J. Banfi, N. Basilico, and S. Carpin, "Optimal redeployment of multirobot teams for communication maintenance," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 3757–3764.

[21] J. Banfi, J. Guzzi, A. Giusti, L. Gambardella, and G. Di Caro, "Fair multitarget tracking in cooperative multi-robot systems," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 5411–5418.

[22] H. Bast and S. Hert, "The area partitioning problem," *Canadian Conference on Computational Geometry (CCCG)*, 2000.

[23] A. Bayoumi, P. Karkowski, and M. Bennewitz, "Speeding up person finding using hidden Markov models," *Robotics and Autonomous Systems*, vol. 115, pp. 40–48, 2019.

[24] J. Beasley, "Route first cluster second methods for vehicle routing," *Omega*, vol. 11, no. 4, pp. 403–408, 1983.

[25] T. Bektas, "The multiple traveling salesman problem: An overview of formulations and solution procedures," *Omega*, vol. 34, no. 3, pp. 209–219, 2006.

[26] J. Bellingham, "Coordination and control of UAV fleets using mixed-integer linear programming," Ph.D. dissertation, Massachusetts Institute of Technology, 2002.

[27] R. Bellman, "The theory of dynamic programming," Rand Corp Santa Monica CA, Tech. Rep., 1954.

[28] S. Benkoski, M. Monticino, and J. Weisinger, "A survey of the search theory literature," *Naval Research Logistics (NRL)*, vol. 38, no. 4, pp. 469–494, 1991.

[29] S. Bochkarev and S. Smith, "On minimizing turns in robot coverage path planning," in *International Conference on Automation Science and Engineering (CASE)*. IEEE, 2016, pp. 1237–1242.

[30] R. Bolaños, M. Echeverry, and J. Escobar, "A multiobjective non-dominated sorting genetic algorithm (NSGA-II) for the multiple traveling salesman problem," *Decision Science Letters*, vol. 4, no. 4, pp. 559–568, 2015.

[31] S. Brown, "Optimal search for a moving target in discrete time and space," *Operations Research*, vol. 28, no. 6, pp. 1275–1289, 1980.

[32] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 387–408, 2012.

[33] J. Chalopin, S. Das, and P. Widmayer, "Deterministic symmetric rendezvous in arbitrary graphs: Overcoming anonymity, failures and uncertainty," in *Search Theory*. Springer, 2013, pp. 175–195.

[34] W. Chan and F. Chin, "Approximation of polygonal curves with minimum number of line segments or minimum error," *International Journal of Computational Geometry & Applications*, vol. 6, no. 1, pp. 59–77, 1996.

[35] B. Charrow, V. Kumar, and N. Michael, "Approximate representations for multi-robot control policies that maximize mutual information," *Autonomous Robots*, vol. 37, no. 4, pp. 383–400, 2014.

[36] H. Choi, L. Brunet, and J. How, "Consensus-based decentralized auctions for robust task allocation," *IEEE Transactions on Robotics*, vol. 25, no. 4, pp. 912–926, 2009.

[37] H. Choset, "Coverage for robotics—a survey of recent results," *Annals of Mathematics and Artificial Intelligence*, vol. 31, no. 1, pp. 113–126, 2001.

[38] H. Choset and P. Pignon, "Coverage path planning: The boustrophedon cellular decomposition," in *Field and service robotics*. Springer, 1998, pp. 203–209.

[39] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," DTIC Document, Tech. Rep., 1976.

[40] T. Chung, G. Hollinger, and V. Isler, "Search and pursuit-evasion in mobile robotics," *Autonomous Robots*, vol. 31, no. 4, pp. 299–316, 2011.

[41] S. Cook, "The complexity of theorem-proving procedures," in *Symposium on Theory of Computing*. ACM, 1971, pp. 151–158.

[42] W. Cook. (2016) Concorde TSP solver. [Online]. Available: http://www.math.uwaterloo.ca/tsp/concorde/

[43] W. Cook and A. Rohe, "Computing minimum-weight perfect matchings," *INFORMS Journal on Computing*, vol. 11, no. 2, pp. 138–148, 1999.

[44] G. Croes, "A method for solving traveling-salesman problems," *Operations Research*, vol. 6, no. 6, pp. 791–812, 1958.

[45] P. Dames, "Distributed multi-target search and tracking using the PHD filter," in *International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*. IEEE, 2017, pp. 1–8.

[46] M. De Gennaro and A. Jadbabaie, "Decentralized control of connectivity for multi-agent systems," in *Conference on Decision and Control (CDC)*. IEEE, 2006, pp. 3628–3633.

[47] B. Dias, "TraderBots: A new paradigm for robust and efficient multirobot coordination in dynamic environments," Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, January 2004.

[48] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[49] D. Ding, Z. Pan, D. Cuiuri, and H. Li, "A tool-path generation strategy for wire and arc additive manufacturing," *The International Journal of Advanced Manufacturing Technology*, vol. 73, no. 1-4, pp. 173–183, 2014.

[50] M. Dooley, J. Case, and N. Romanov, "System and method for autonomous mopping of a floor surface," Aug. 1 2019, US Patent App. 16/382,864.

[51] D. Douglas and T. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica*, vol. 10, no. 2, pp. 112–122, 1973.

[52] G. Dudek and N. Roy, "Multi-robot rendezvous in unknown environments, or, what to do when you're lost at the zoo," in *National Conference on Artificial Intelligence, Workshop on Online Search*. AAAI, 1997.

[53] E. Eade, P. Fong, and M. Munich, "Monocular graph slam with complexity reduction," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2010, pp. 3017–3024.

[54] E. Eade, M. Munich, and P. Fong, "Systems and methods for VSLAM optimization," Mar. 15 2016, US Patent 9,286,810.

[55] J. Eagle and J. Yee, "An optimal branch-and-bound procedure for the constrained path, moving target search problem," *Operations Research*, vol. 38, no. 1, pp. 110–114, 1990.

[56] Ecovacs. (2019, Oct.) Winbot window cleaning robots. [Online]. Available: https://www.ecovacs.com/global/winbot-window-cleaning-robot

[57] A. Farinelli, L. Iocchi, D. Nardi, and V. Ziparo, "Assignment of dynamically perceived tasks by token passing in multirobot systems," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1271–1288, 2006.

[58] D. Ferguson, N. Kalra, and A. Stentz, "Replanning with RRTs," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2006, pp. 1243–1248.

[59] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak Mathematical Journal*, vol. 23, no. 2, pp. 298–305, 1973.

[60] R. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[61] P. Fong, C. Smith, M. Munich, and S. O'Dea, "Mobile cleaning robot teaming and persistent mapping," Jul. 11 2019, US Patent App. 15/863,681.

[62] G. Frederickson, M. Hecht, and C. Kim, "Approximation algorithms for some routing problems," in *Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1976, pp. 216–227.

[63] S. Friedman and H. Moravec, "Distributed multi-robot system," Jun. 17 2014, US Patent 8,755,936.

[64] E. Galceran, R. Campos, N. Palomeras, D. Ribas, M. Carreras, and P. Ridao, "Coverage path planning with real-time replanning and surface reconstruction for inspection of three-dimensional underwater structures using autonomous underwater vehicles," *Journal of Field Robotics*, vol. 32, no. 7, pp. 952–983, 2015.

[65] E. Galceran and M. Carreras, "A survey on coverage path planning for robotics," *Robotics and Autonomous Systems*, vol. 61, no. 12, pp. 1258–1276, 2013.

[66] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.

[67] B. Gerkey and M. Maja, "Sold!: Auction methods for multirobot coordination," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 5, pp. 758–768, 2002.

[68] B. Gerkey and M. Maja, "A formal analysis and taxonomy of task allocation in multi-robot systems," *The International Journal of Robotics Research*, vol. 23, no. 9, pp. 939–954, 2004.

[69] C. Geyer, "Active target search from UAVs in urban environments," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2008, pp. 2366–2371.

[70] D. Goel, J. Case, D. Tamino, J. Gutmann, M. Munich, M. Dooley, and P. Pirjanian, "Systematic floor coverage of unknown environments using rectangular regions and localization certainty," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2013, pp. 1–8.

[71] A. Goldhoorn, A. Garrell, R. Alquézar, and A. Sanfeliu, "Searching and tracking people with cooperative mobile robots," *Autonomous Robots*, vol. 42, no. 4, pp. 739–759, 2018.

[72] S. Gorenstein, "Printing press scheduling for multi-edition periodicals," *Management Science*, vol. 16, no. 6, pp. 373–383, 1970.

[73] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[74] M. Held, *On the computational geometry of pocket machining*. Springer Science & Business Media, 1991, vol. 500.

[75] M. Held and R. Karp, "The traveling-salesman problem and minimum spanning trees: Part II," *Mathematical programming*, vol. 1, no. 1, pp. 6–25, 1971.

[76] K. Helsgaun. LKH version 2.0.7. [Online]. Available: http://www.akira.ruc.dk/~keld/research/LKH/

[77] K. Helsgaun, "An effective implementation of the Lin–Kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[78] S. Hert and V. Lumelsky, "Polygon area decomposition for multiple-robot workspace division," *International Journal of Computational Geometry & Applications*, vol. 8, no. 4, pp. 437–466, 1998.

[79] R. Hohzaki, "Search games: Literature and survey," *Journal of the Operations Research Society of Japan*, vol. 59, no. 1, pp. 1–34, 2016.

[80] G. Hollinger and S. Singh, "Multirobot coordination with periodic connectivity: Theory and experiments," *IEEE Transactions on Robotics*, vol. 28, no. 4, pp. 967–973, 2012.

[81] G. Hollinger, S. Singh, J. Djugash, and A. Kehagias, "Efficient multi-robot search for a moving target," *The International Journal of Robotics Research*, vol. 28, no. 2, pp. 201–219, 2009.

[82] G. Hollinger and G. Sukhatme, "Sampling-based robotic information gathering algorithms," *The International Journal of Robotics Research*, vol. 33, no. 9, pp. 1271–1287, 2014.

[83] G. Hollinger, S. Yerramalli, S. Singh, U. Mitra, and G. Sukhatme, "Distributed data fusion for multirobot search," *IEEE Transactions on Robotics*, vol. 31, no. 1, pp. 55–66, 2015.

[84] W. Hönig, S. Kiesel, A. Tinka, J. Durham, and N. Ayanian, "Conflict-based search with optimal task assignment," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2018, pp. 757–765.

[85] W. Huang, "Optimal line-sweep-based decompositions for coverage algorithms," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2001, pp. 27–32.

[86] Y. Huang and K. Gupta, "RRT-SLAM for motion planning with motion and map uncertainty for robot exploration," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2008, pp. 1077–1082.

[87] B. Ichter, J. Harrison, and M. Pavone, "Learning sampling distributions for robot motion planning," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 7087–7094.

[88] iRobot. (2019, Oct.) Braava m Series. [Online]. Available: https://www.irobot.com/braava/m-series

[89] iRobot. (2019, Oct.) Roomba s Series. [Online]. Available: https://www.irobot.com/roomba/s-series

[90] L. Jaillet, J. Cortés, and T. Siméon, "Transition-based RRT for path planning in continuous cost spaces," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2008, pp. 2145–2150.

[91] J. Jones and P. Mass, "Method and system for multi-mode coverage for an autonomous robot," Jun. 11 2013, US Patent 8,463,438.

[92] R. Jonker and T. Volgenant, "Technical note—an improved transformation of the symmetric multiple traveling salesman problem," *Operations Research*, vol. 36, no. 1, pp. 163–167, 1988.

[93] A. Jotshi and R. Batta, "Search for an immobile entity on a network," *European Journal of Operational Research*, vol. 191, no. 2, pp. 347–359, 2008.

[94] Y. Kantaros and M. Zavlanos, "Distributed intermittent connectivity control of mobile robot networks," *IEEE Transactions on Automatic Control*, vol. 62, no. 7, pp. 3109–3121, 2017.

[95] A. Kapoutsis, S. Chatzichristofis, and E. Kosmatopoulos, "DARP: Divide areas algorithm for optimal multi-robot coverage path planning," *Journal of Intelligent & Robotic Systems*, vol. 86, no. 3-4, pp. 663–680, 2017.

[96] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *Robotics Science and Systems VI*, vol. 104, 2010.

[97] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for a class of pursuit-evasion games," in *Algorithmic Foundations of Robotics IX*. Springer, 2011, pp. 71–87.

[98] D. Karapetyan and G. Gutin, "Lin–Kernighan heuristic adaptations for the generalized traveling salesman problem," *European Journal of Operational Research*, vol. 208, no. 3, pp. 221–232, 2011.

[99] N. Karapetyan, K. Benson, C. McKinney, P. Taslakian, and I. Rekleitis, "Efficient multi-robot coverage of a known environment," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 1846–1852.

[100] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*. Springer, 1972, pp. 85–103.

[101] L. Kavraki, P. Svestka, J. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[102] E. Kivelevitch, K. Cohen, and M. Kumar, "Market-based solution to the allocation of tasks to agents," *Procedia Computer Science*, vol. 6, pp. 28–33, 2011.

[103] E. Kivelevitch, B. Sharma, N. Ernest, M. Kumar, and K. Cohen, "A hierarchical market solution to the min-max multiple depots vehicle routing problem," *Unmanned Systems*, vol. 2, pp. 87–100, 2014.

[104] A. Kleiner, "The low-cost evolution of AI in domestic floor cleaning robots," *AI Magazine*, vol. 39, no. 2, pp. 89–91, 2018.

[105] A. Kleiner, R. Baravalle, A. Kolling, P. Pilotti, and M. Munich, "A solution to room-by-room coverage for autonomous cleaning robots," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 5346–5352.

[106] A. Kleiner and M. Munich, "Systems and methods for configurable operation of a robot based on area classification," Jun. 4 2019, US Patent App. 10/310,507.

[107] D. Knuth, "Postscript about NP-hard problems," *ACM SIGACT News*, vol. 6, no. 2, pp. 15–16, 1974.

[108] A. Kolling and I. Vandermeulen, "Turn-minimizing or turn-reducing robot coverage," Mar. 19 2020, US Patent App. 16/565,721.

[109] C. Kong, A. New, and I. Rekleitis, "Distributed coverage with multi-robot system," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2006, pp. 2423–2429.

[110] B. Koopman, "Search and screening," *Center for Naval Analysis, Alexandria, Virginia*, 1946.

[111] G. Korsah, A. Stentz, and M. Dias, "A comprehensive taxonomy for multi-robot task allocation," *The International Journal of Robotics Research*, vol. 32, no. 12, pp. 1495–1512, 2013.

[112] J. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[113] Y. Kuo, A. Barbu, and B. Katz, "Deep sequential models for sampling-based planning," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 6490–6497.

[114] H. Lau, S. Huang, and G. Dissanayake, "Optimal search for multiple targets in a built environment," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2005, pp. 3740–3745.

[115] H. Lau, S. Huang, and G. Dissanayake, "Probabilistic search for a moving target in an indoor environment," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2006, pp. 3393–3398.

[116] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[117] S. LaValle, M. Branicky, and S. Lindemann, "On the relationship between classical grid search and probabilistic roadmaps," *The International Journal of Robotics Research*, vol. 23, no. 7-8, pp. 673–692, 2004.

[118] E. Lawler, *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

[119] A. Li, P. Penumarthi, J. Banfi, N. Basilico, J. OKane, I. Rekleitis, S. Nelakuditi, and F. Amigoni, "Multi-robot online sensing strategies for the construction of communication maps," *Autonomous Robots*, pp. 1–21, 2019.

[120] Y. Li, H. Chen, M. Er, and X. Wang, "Coverage path planning for UAVs based on enhanced exact cellular decomposition method," *Mechatronics*, vol. 21, no. 5, pp. 876–885, 2011.

[121] M. Liggett, I. Kamada, F. Hopke, G. Huat, C. Fiebig, S. Connor, and A. Alan, "Robotic cleaner," Mar. 7 2019, US Patent App. 16/100,687.

[122] S. Lin and B. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations Research*, vol. 21, no. 2, pp. 498–516, 1973.

[123] W. Liu, S. Li, F. Zhao, and A. Zheng, "An ant colony optimization algorithm for the multiple traveling salesmen problem," in *Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2009, pp. 1533–1537.

[124] M. Llofriu, P. Fong, V. Karapetyan, and M. Munich, "Mapping under changing trajectory estimates," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 1403–1410.

[125] M. López-Ibáñez, C. Blum, J. Ohlmann, and B. Thomas, "The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization," *Applied Soft Computing*, vol. 13, no. 9, pp. 3806–3815, 2013.

[126] W. Luo, S. Yi, and K. Sycara, "Behavior mixing with minimum global and subgroup connectivity maintenance for large-scale multi-robot systems," *arXiv preprint arXiv:1910.01693*, 2019.

[127] H. Ma and S. Koenig, "Optimal target assignment and path finding for teams of agents," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2016, pp. 1144–1152.

[128] K. Mak and A. Morton, "A modified Lin–Kernighan traveling-salesman heuristic," *Operations Research Letters*, vol. 13, no. 3, pp. 127–132, 1993.

[129] K. Mak and A. Morton, "Distances between traveling salesman tours," *Discrete Applied Mathematics*, vol. 58, no. 3, pp. 281–291, 1995.

[130] M. Malek, M. Guruswamy, M. Pandya, and H. Owens, "Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem," *Annals of Operations Research*, vol. 21, no. 1, pp. 59–84, 1989.

[131] R. Matai, S. Singh, and M. Mittal, "Traveling salesman problem: An overview of applications, formulations, and solution approaches," *Traveling Salesman Problem, Theory and Applications*, pp. 1–24, 2010.

[132] Maytronics. (2019, Oct.) Dolphin residential pool cleaning robots. [Online]. Available: https://maytronicsus.com/products/residential-pool-cleaning-robots/

[133] I. Maza and A. Ollero, "Multiple UAV cooperative searching operation using polygon area decomposition and efficient coverage algorithms," in *Distributed Autonomous Robotic Systems 6*. Springer, 2007, pp. 221–230.

[134] M. McIntire, E. Nunes, and M. Gini, "Iterated multi-robot auctions for precedence-constrained task scheduling," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2016, pp. 1078–1086.

[135] M. Meghjani, S. Manjanna, and G. Dudek, "Multi-target rendezvous search," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 2596–2603.

[136] N. Megiddo, S. Hakimi, M. Garey, D. Johnson, and C. Papadimitriou, "The complexity of searching a graph," *Journal of the ACM*, vol. 35, no. 1, pp. 18–44, 1988.

[137] M. Missura, D. Lee, and M. Bennewitz, "Minimal construct: Efficient shortest path finding for mobile robots in polygonal maps," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 7918–7923.

[138] R. Mole, D. Johnson, and K. Wells, "Combinatorial analysis for route first-cluster second vehicle routing," *Omega*, vol. 11, no. 5, pp. 507–512, 1983.

[139] R. Morin, F. Bursal, and H. Boeschenstein, "Evacuation station," Apr. 3 2018, US Patent 9,931,007.

[140] Y. Mostofi, A. Gonzalez-Ruiz, A. Gaffarkhah, and D. Li, "Characterization and modeling of wireless channels for networked robotic and control systems-a comprehensive overview," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2009, pp. 4849–4854.

[141] Y. Mostofi, M. Malmirchegini, and A. Ghaffarkhah, "Estimation of communication signal strength in robotic networks," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2010, pp. 1946–1951.

[142] M. Munich, N. Romanov, D. Goel, and P. Fong, "Systems and methods for performing simultaneous localization and mapping using machine vision systems," Apr. 6 2017, US Patent App. 15/353,368.

[143] B. Na, "Heuristic approaches for no-depot k-traveling salesmen problem with a minmax objective," Ph.D. dissertation, Texas A&M University, College Station, Texas, 2006.

[144] R. Nallusamy, K. Duraiswamy, R. Dhanalaksmi, and P. Parthiban, "Optimization of non-linear multiple traveling salesman problem using k-means clustering, shrink wrap algorithm and meta-heuristics," *International Journal of Nonlinear Science*, vol. 9, no. 2, pp. 171–177, 2010.

[145] M. Nanjanath and M. Gini, "Repeated auctions for robust task execution by a robot team," *Robotics and Autonomous Systems*, vol. 58, no. 7, pp. 900–909, 2010.

[146] N. Nguyen, T. Nguyen, and J. Rothe, "Approximate solutions to max-min fair and proportionally fair allocations of indivisible goods," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2017, pp. 262–271.

[147] J. Norris and J. R. Norris, *Markov chains*. Cambridge University Press, 1998, no. 2.

[148] T. Oksanen and A. Visala, "Coverage path planning algorithms for agricultural field machines," *Journal of Field Robotics*, vol. 26, no. 8, pp. 651–668, 2009.

[149] J. Pacheco and R. Martí, "Tabu search for a multi-objective routing problem," *Journal of the Operational Research Society*, vol. 57, no. 1, pp. 29–37, 2006.

[150] C. Papadimitriou, "The Euclidean travelling salesman problem is NP-complete," *Theoretical Computer Science*, vol. 4, no. 3, pp. 237–244, 1977.

[151] Y. Pei, M. Mutka, and N. Xi, "Coordinated multi-robot real-time exploration with connectivity and bandwidth awareness," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2010, pp. 5460–5465.

[152] E. Peless, S. Abramson, R. Friedman, and I. Peleg, "Area coverage with an autonomous robot," Jul. 10 2008, US Patent App. 12/054,123.

[153] J. Potvin, G. Lapalme, and J. Rousseau, "A generalized k-opt exchange procedure for the MTSP," *INFOR: Information Systems and Operational Research*, vol. 27, no. 4, pp. 474–481, 1989.

[154] R. Prim, "Shortest connection networks and some generalizations," *Bell Labs Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[155] G. Reinelt. (2015) TSPLIB. [Online]. Available: http://elib.zib.de/pub/mp-testdata/tsp/tsplib/

[156] I. Rekleitis, A. New, E. Rankin, and H. Choset, "Efficient boustrophedon multi-robot coverage: An algorithmic approach," *Annals of Mathematics and Artificial Intelligence*, vol. 52, no. 2, pp. 109–142, 2008.

[157] J. Riehl, G. Collins, and J. Hespanha, "Cooperative graph-based model predictive search," in *Conference on Decision and Control (CDC)*. IEEE, 2007, pp. 2998–3004.

[158] A. Riva, J. Banfi, C. Fanton, N. Basilico, and F. Amigoni, "A journey among pairs of vertices: Computing robots' paths for performing joint measurements," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2018, pp. 229–237.

[159] C. Robin and S. Lacroix, "Multi-robot target detection and tracking: Taxonomy and survey," *Autonomous Robots*, vol. 40, no. 4, pp. 729–760, 2016.

[160] Robomow. (2019, Oct.) Meet the Robomow models. [Online]. Available: https://usa.robomow.com/products/

[161] N. Romanov, C. Johnson, J. Case, D. Goel, S. Gutmann, and M. Dooley, "Mobile robot for cleaning," Feb. 24 2015, US Patent 8,961,695.

[162] J. Royset and H. Sato, "Route optimization for multiple searchers," *Naval Research Logistics (NRL)*, vol. 57, no. 8, pp. 701–717, 2010.

[163] L. Sabattini, C. Secchi, N. Chopra, and A. Gasparri, "Distributed control of multirobot systems with global connectivity maintenance," *IEEE Transactions on Robotics*, vol. 29, no. 5, pp. 1326–1332, 2013.

[164] M. Sahi. (2016, May) Consumer attitudes about household robots. [Online]. Available: https://www.tractica.com/robotics/consumer-attitudes-about-household-robots/

[165] P. Sandin, J. Jones, D. Ozick, D. Cohen, D. Lewis, C. Vu, Z. Dubrovsky, J. Preneta, J. Mammen, D. Gilbert, T. Campbell, and J. Bergman, "Lawn care robot," Feb. 10 2015, US Patent 8,954,193.

[166] S. Sariel and T. Balch, "Real time auction based allocation of tasks for multi-robot exploration problem in dynamic environments," in *Workshop on Integrating Planning into Scheduling*. AAAI, 2005, pp. 27–33.

[167] A. Sarmiento, R. Murrieta-Cid, and S. Hutchinson, "An efficient motion strategy to compute expected-time locally optimal continuous search paths in known environments," *Advanced Robotics*, vol. 23, no. 12-13, pp. 1533–1560, 2009.

[168] S. Schneckenburger, B. Dorn, and U. Endriss, "The Atkinson inequality index in multiagent resource allocation," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2017, pp. 272–280.

[169] J. Seo, S. Chae, J. Shim, D. Kim, C. Cheong, and T. Han, "Fast contour-tracing algorithm based on a pixel-following method for image sensors," *Sensors*, vol. 16, no. 3, pp. 353–379, 2016.

[170] W. Sheng, N. Xi, H. Chen, Y. Chen, and M. Song, "Surface partitioning in automated CAD-guided tool planning for additive manufacturing," in *International Conference on Intelligent Robots and Systems (IROS)*, vol. 2. IEEE, 2003, pp. 2072–2077.

[171] M. SHIGETO, K. Watanabe, H. Ogahara, and S. Matsumura, "Autonomous travel-type cleaner," Apr. 30 2019, US Patent App. 10/271,705.

[172] V. Shim, K. Tan, and C. Cheong, "A hybrid estimation of distribution algorithm with decomposition for solving the multiobjective multiple traveling salesman problem," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 5, pp. 682–691, 2012.

[173] F. Shkurti, N. Kakodkar, and G. Dudek, "Model-based probabilistic pursuit via inverse reinforcement learning," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 7804–7811.

[174] A. Sintov and A. Shapiro, "Time-based RRT algorithm for rendezvous planning of two dynamic systems," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 6745–6750.

[175] J. Song, K. Lee, S. Moon, S. Lee, and J. Ko, "Robot cleaner, robot cleaning system and method of controlling same," Jan. 27 2009, US Patent 7,480,958.

[176] B. Soylu, "A general variable neighborhood search heuristic for multiple traveling salesmen problem," *Computers & Industrial Engineering*, vol. 90, pp. 390–401, 2015.

[177] L. Stone, "OR forum—what's happened in search theory since the 1975 Lanchester prize?" *Operations Research*, vol. 37, no. 3, pp. 501–506, 1989.

[178] M. Stout, G. Brisson, E. Di Bernardo, P. Pirjanian, D. Goel, J. P. Case, and M. Dooley, "Methods and systems for complete coverage of a surface by an autonomous robot," Feb. 20 2018, US Patent 9,895,808.

[179] E. Stump, A. Jadbabaie, and V. Kumar, "Connectivity management in mobile robot teams," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2008, pp. 1525–1530.

[180] Y. Sung and P. Tokekar, "Algorithm for searching and tracking an unknown and varying number of mobile targets using a limited FoV sensor," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 6246–6252.

[181] J. Svestka and V. Huckfeldt, "Computational experience with an m-salesman traveling salesman algorithm," *Management Science*, vol. 19, no. 7, pp. 790–799, 1973.

[182] C. Taylor, A. Parker, S. Lau, E. Blair, A. Heninger, and E. Ng, "Robot vacuum with internal mapping system," Sep. 28 2010, US Patent 7,805,220.

[183] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. MIT press, 2005.

[184] J. Thunberg, D. Anisi, and P. Ögren, "A comparative study of task assignment and path planning methods for multi-UGV missions," in *Optimization and Cooperative Control Strategies*. Springer, 2009, pp. 167–180.

[185] A. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, no. 1, pp. 230–265, 1937.

[186] I. Vandermeulen, R. Groß, and A. Kolling, "Re-establishing communication in teams of mobile robots," in *International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 7947–7954.

[187] I. Vandermeulen, R. Groß, and A. Kolling, "Balanced task allocation by partitioning the multiple traveling salesperson problem," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, 2019, pp. 1479–1487.

[188] I. Vandermeulen, R. Groß, and A. Kolling, "Turn-minimizing multirobot coverage," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 1014–1020.

[189] I. Vandermeulen, R. Groß, and A. Kolling, "Sampling based search for a semi-cooperative target," *under review for publication in International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[190] P. Varakantham, H. Mostafa, N. Fu, and H. Lau, "DIRECT: A scalable approach for route guidance in selfish orienteering problems," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2015, pp. 483–491.

[191] P. Venkatesh and A. Singh, "Two metaheuristic approaches for the multiple traveling salesperson problem," *Applied Soft Computing*, vol. 26, pp. 74–89, 2015.

[192] J. Vicenti, "Mobile robot area cleaning," Mar. 29 2018, US Patent App. 15/698,005.

[193] J. Vokřínek, A. Komenda, and M. Pěchouček, "Agents towards vehicle routing problems," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2010, pp. 773–780.

[194] Y. Wang, Y. Chen, and Y. Lin, "Memetic algorithm based on sequential variable neighborhood descent for the minmax multiple traveling salesman problem," *Computers & Industrial Engineering*, vol. 106, pp. 105–122, 2017.

[195] A. Washburn, "Search for a moving target: The FAB algorithm," *Operations Research*, vol. 31, no. 4, pp. 739–751, 1983.

[196] M. Wei and V. Isler, "Coverage path planning under the energy constraint," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 368–373.

[197] E. Welzl, "Constructing the visibility graph for n-line segments in $\mathcal{O}(n^2)$ time," *Information Processing Letters*, vol. 20, no. 4, pp. 167–171, 1985.

[198] D. Wicke, D. Freelan, and S. Luke, "Bounty hunters and multiagent task allocation," in *International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*. IFAAMAS, 2015, pp. 387–394.

[199] A. Winfield, "Distributed sensing and data collection via broken ad hoc wireless connected networks of mobile robots," in *Distributed Autonomous Robotic Systems 4*. Springer, 2000, pp. 273–282.

[200] B. Wolfe and P. Lu, "Wall following robot," Mar. 20 2018, US Patent 9,918,605.

[201] A. Xu, C. Viriyasuthee, and I. Rekleitis, "Efficient complete coverage of a known arbitrary environment with applications to aerial operations," *Autonomous Robots*, vol. 36, no. 4, pp. 365–381, 2014.

[202] S. Yoon and J. Kim, "Efficient multi-agent task allocation for collaborative route planning with multiple unmanned vehicles," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 3580–3585, 2017.

[203] H. Yu, R. Beard, M. Argyle, and C. Chamberlain, "Probabilistic path planning for cooperative target tracking using aerial and ground vehicles," in *American Control Conference (ACC)*. IEEE, 2011, pp. 4673–4678.

[204] S. Yu, "Hidden semi-Markov models," *Artificial Intelligence*, vol. 174, no. 2, pp. 215–243, 2010.

[205] M. Zavlanos, H. Tanner, A. Jadbabaie, and G. Pappas, "Hybrid control for connectivity preserving flocking," *IEEE Transactions on Automatic Control*, vol. 54, no. 12, pp. 2869–2875, 2009.

[206] A. Zelinsky, R. Jarvis, J. Byrne, and S. Yuta, "Planning paths of complete coverage of an unstructured environment by a mobile robot," in *International Conference on Advanced Robotics*, vol. 13, 1993, pp. 533–538.

[207] R. Zlot and A. Stentz, "Market-based multirobot coordination for complex tasks," *The International Journal of Robotics Research*, vol. 25, no. 1, pp. 73–101, 2006.

Appendix A

Visibility graphs

Unobstructed paths are essential in robotics, both for motion planning and for communication. A visibility graph is a set of straight, unobstructed paths between a set of key locations in an environment (Figure A.1). They are extremely useful, both for shortest path planning (Appendix B) and for determining when two robots will be able to communicate with each other. In this appendix, I will present two approaches to computing a visibility graph. The graph is computed using a set of points, $\mathcal{V}$, which are all contained in (or on the boundary of) a polygonal environment, $\mathcal{Q} \subset \mathbb{R}^2$. The naïve algorithm runs in $\mathcal{O}(|\mathcal{V}|^3)$ but is conceptually simpler, and thus easier to implement than Welzl's $\mathcal{O}(|\mathcal{V}|^2)$ algorithm.

A.1 Naïve algorithm

The definition of a visibility graph provides an obvious, albeit inefficient, way of constructing a graph. In this naïve approach, we simply check for intersections between each possible edge of the graph with each edge of the environment's boundary (Algorithm A.1). An edge is only added to the graph if it does not intersect with any edge of the boundary. Assuming the boundary consists of one main outer polygon, and possibly several internal polygon obstacles, we can check intersections with the boundary by iterating over all edges of the boundary. Overall

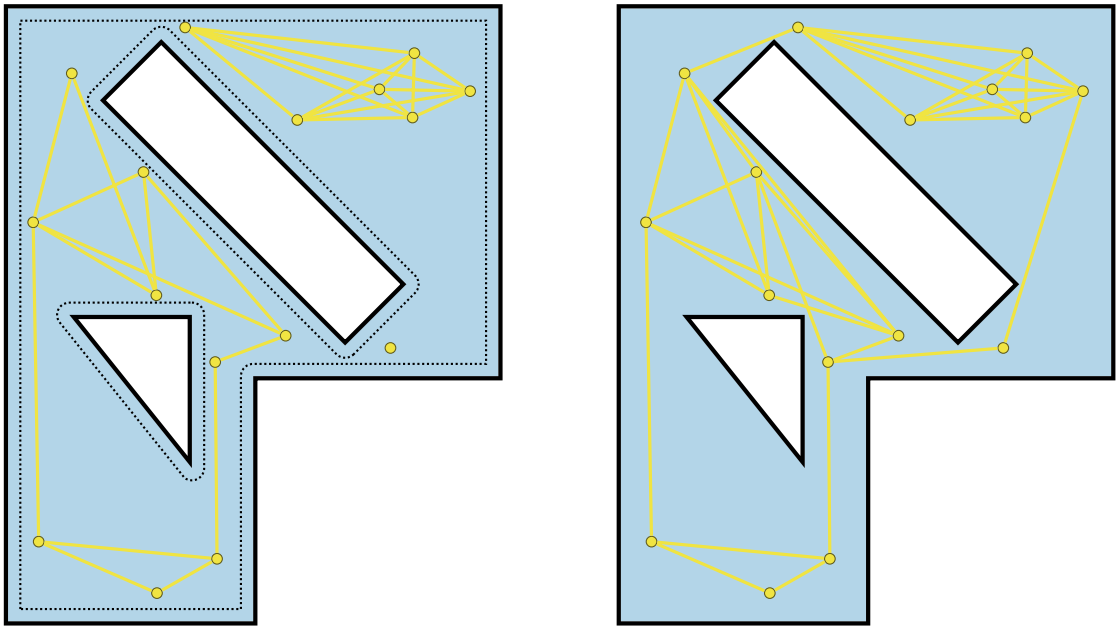


Figure A.1: Visibility graph used for path planning (left) and communication (right). The planning graph is based on a buffered environment since the center of robot can't get more than half of its width away from the wall.

this approach requires us to check $\mathcal{O}(|\mathcal{V}|^2)$ pairs of vertices and each vertex is compared to $\mathcal{O}(|\partial\mathcal{Q}|)$ edges of the boundary for a total complexity of $\mathcal{O}(|\mathcal{V}|^2|\partial\mathcal{Q}|)$. Assuming we treat vertices of the boundary as points of interest, $|\partial\mathcal{Q}| \leq |\mathcal{V}|$ and so the complexity is $\mathcal{O}(|\mathcal{V}|^3)$.

Algorithm A.1: Naïve visibility graph

Input: Environment, $\mathcal{Q} \subset \mathbb{R}^2$; and set of locations, $\mathcal{V}$
Output: Set of visible edges, $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$

```
1  E ← {}
2  for pairs of vertices (v0, v1) ∈ V × V do
3      intersects ← false
4      for edge e ∈ ∂Q do
5          if (v0, v1) intersects with e then
6              intersects ← true
7              break for
8      if intersects = false then
9          E ← E ∪ {(v0, v1)}
10 return E
```

A.2 Welzl's algorithm

The naïve visibility graph algorithm runs in cubic time, which often makes it a limiting step of algorithms that rely on it. For example, shortest path planning algorithms (Appendix B) are based on the visibility graph and Dijkstra's algorithm (Section B.1). Although Dijkstra's algorithm is quadratic, the overall planning algorithm is cubic if we use the naïve visibility graph algorithm. Fortunately, Welzl's algorithm [197] is a quadratic visibility graph algorithm! If we use this algorithm, we can solve shortest path planning algorithms in $\mathcal{O}(|\mathcal{V}|^2)$ after all.

The form of the algorithm that Welzl presented [197] is only valid for sets of line segments and the vertices of these each segment must not be co-linear with vertices of any other segment. For most robotics problems, we have a polygonal boundary and single points as points of interest. I had to modify Welzl's algorithm to incorporate points and polygons and colinear vertices.

The basic idea of Welzl's algorithm is to extended parallel rays out from each vertex and see which polygon edge or line segment they first encounter (Figure A.2). During the algorithm, the rays are all rotated simultaneously and we keep track of which edge each ray hits first. Whenever the edge that is visible from a given vertex changes, there must be a line of visibility between that vertex and some other vertex. Therefore, we can compute the visibility graph by rotating the rays and seeing when the visible edges change. It turns out that the edges which are hit by the rays only changes when the ray's direction is equal to the direction between pairs of vertices (points of interest and vertices of the environment polygon). Furthermore, for each of these directions, only a single ray's visibility changes. Since there are $\mathcal{O}(|\mathcal{V}|^2)$ possible ray directions and we can compute the change in visibility in constant time, the entire algorithm runs in $\mathcal{O}(|\mathcal{V}|^2)$.

The algorithm (Algorithm A.2) therefore works as follows. We start by sorting a list of all pairs of vertices (Algorithm A.3). These pairs are defined so that the

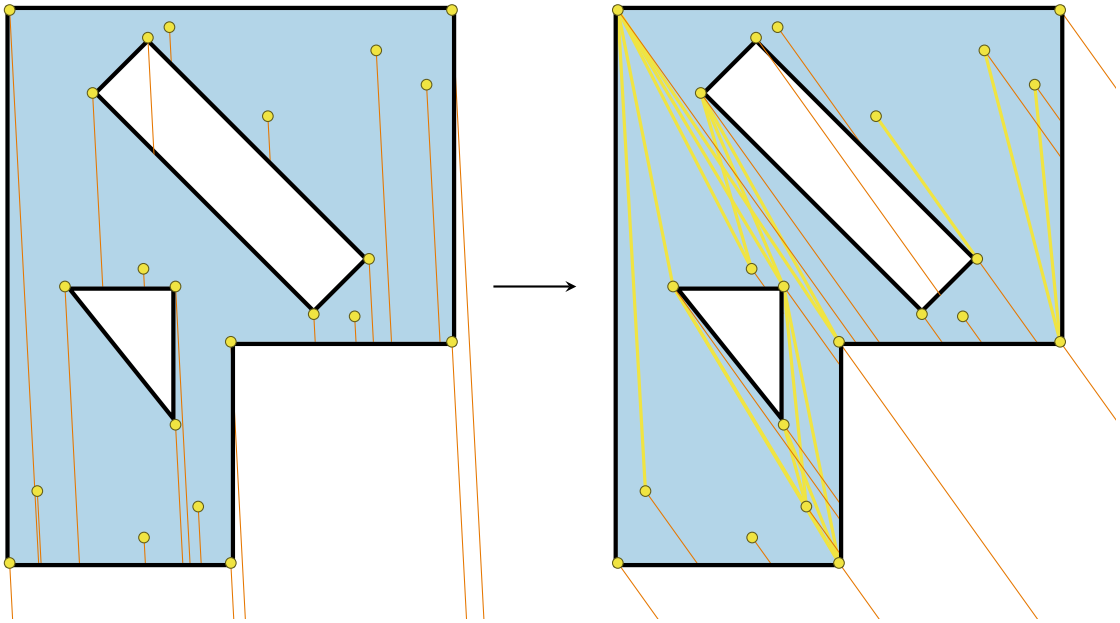


Figure A.2: Welzl's algorithm is based on the idea of extending parallel rays from each vertex (left). These rays are all rotated simultaneously and edges are added to the visibility graph when the first edge encountered by a ray changes (right).

angle between the two vertices is between $(-90°, 90°)$ and is sorted so based on the direction of pairs, so that pairs with directions closer to $-90°$ come first. If there are multiple pairs with the same direction, the tiebreaker criteria depend on how far along the ray the start and end vertices are (Algorithm A.4). This sort order guarantees the pairs are checked in the correct order so that all the required edges are added to the visibility graph and the views are updated correctly.

Next, the initial $\texttt{view}(\cdot)$ is computed (Algorithm A.5). The $\texttt{view}(v)$ is the first edge intersected by the ray leaving v in the current direction. The initial $\texttt{view}(\cdot)$ uses an initial direction which is greater than $-90°$ but less than any direction of any pair in $\texttt{pairs}(\mathcal{V})$. This choice guarantees that no ray intersects an edge at one of its endpoints. The initial $\texttt{view}(\cdot)$ is computed in $\mathcal{O}(|\mathcal{V}|^2)$ by comparing each vertex with each edge to find the closest edge, measured along the initial direction. If the ray does not intersect any edge, we set $\texttt{view}(v) = \infty$.

At this point, the main loop of the algorithm begins. It iterates over pairs

Algorithm A.2: Welzl's algorithm

Input: Environment, $\mathcal{Q} \subset \mathbb{R}^2$; and set of locations, $\mathcal{V}$
Output: set of visible edges, $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$

```
pairs ← sorted vertex pairs of V                    /* Algorithm A.3 */
view, dist ← Initial views and distances            /* Algorithm A.5 */
for pair of vertices (v_0, v_1) ∈ pairs do
    if v_1 is not part of any edge in ∂Q then
        if v_1 is in front of view(v_0) then
            E ← E ∪ {(v_0, v_1)}
    else if v_1 is part of one edge in ∂Q then
        e ← edge in ∂Q with v_1 as an endpoint
        if e = view(v_0) then
            E ← E ∪ {(v_0, v_1)}
            view(v_0) ← view(v_1)
        else if v_1 is in front of view(v_0) then
            E ← E ∪ {(v_0, v_1)}
            view(v_0) ← e
    else if v_1 is part of two edges in ∂Q then
        e_CW, e_CCW ← edges in ∂Q with v_1 as an endpoints
        if e_CW = (v_0, v_1) then
            view(v_0) ← view(v_1)
        else if e_CCW = (v_0, v_1) then
            view(v_0) ← e_CW
        else if e_CCW = view(v_0) then
            E ← E ∪ {(v_0, v_1)}
            if e_CW, e_CCW are on opposite sides of (v_0, v_1) then
                view(v_0) ← e_CW
            else
                view(v_0) ← view(v_1)
        else if v_1 is in front of view(v_0) then
            E ← E ∪ {(v_0, v_1)}
            view(v_0) ← e_CW
for e ∈ E do
    if e is fully occluded then
        E ← E \ {e}
return E
```

Algorithm A.3: Sorted vertex pairs

Input: Vertices of visibility graph, $\mathcal{V}$
Output: Sorted list of vertex pairs, $\texttt{pairs} \cong \mathcal{V} \times \mathcal{V}$

```
pairs ← empty list of vertex pairs in V × V
for vertex pairs (v_0, v_1) ∈ V × V do
    if (v_1 is somewhere left of v_0) or (directly above v_0) then
        pairs ← pairs ∪ {(v_1, v_0)}
    else
        pairs ← pairs ∪ {(v_0, v_1)}
Sort pairsusing Welzl sort order                    /* Algorithm A.4 */
return pairs
```

Algorithm A.4: Welzl sort order

Input: Two vertex pairs, $(v_i, v_i'), (v_j, v_j')$
Output: Vertex pair which should be checked first

```
if the pairs have different directions then
    return pair whose direction is closer to −90°
else if the pairs are part of different rays then
    return pair to the right when looking in their common direction
else if the pairs have different start vertices then
    return pair whose start vertex is further along the ray
else if the pairs have different end vertices then
    return pair whose end vertex is not as far along the ray
```

Algorithm A.5: Initial view

Input: Environment, $\mathcal{Q} \subset \mathbb{R}^2$; and set of vertices, $\mathcal{V}$
Output: Initial views, $\texttt{view} : \mathcal{V} \to \partial\mathcal{Q}$; and distances, $\texttt{dist} : \mathcal{V} \to \mathbb{R}_{\geq 0}$

```
θ ← angle in (−90°, 90°] smaller than any angle between two vertices of V
for vertex v ∈ V do
    view(v) ← ∞
    dist(v) ← ∞
    for edge e ∈ ∂Q do
        if ray from v in direction θ intersects e then
            d ← minimum distance from v to e
            if d < dist(v) then
                view(v) ← e
                dist(v) ← d
return view(·), dist(·)
```

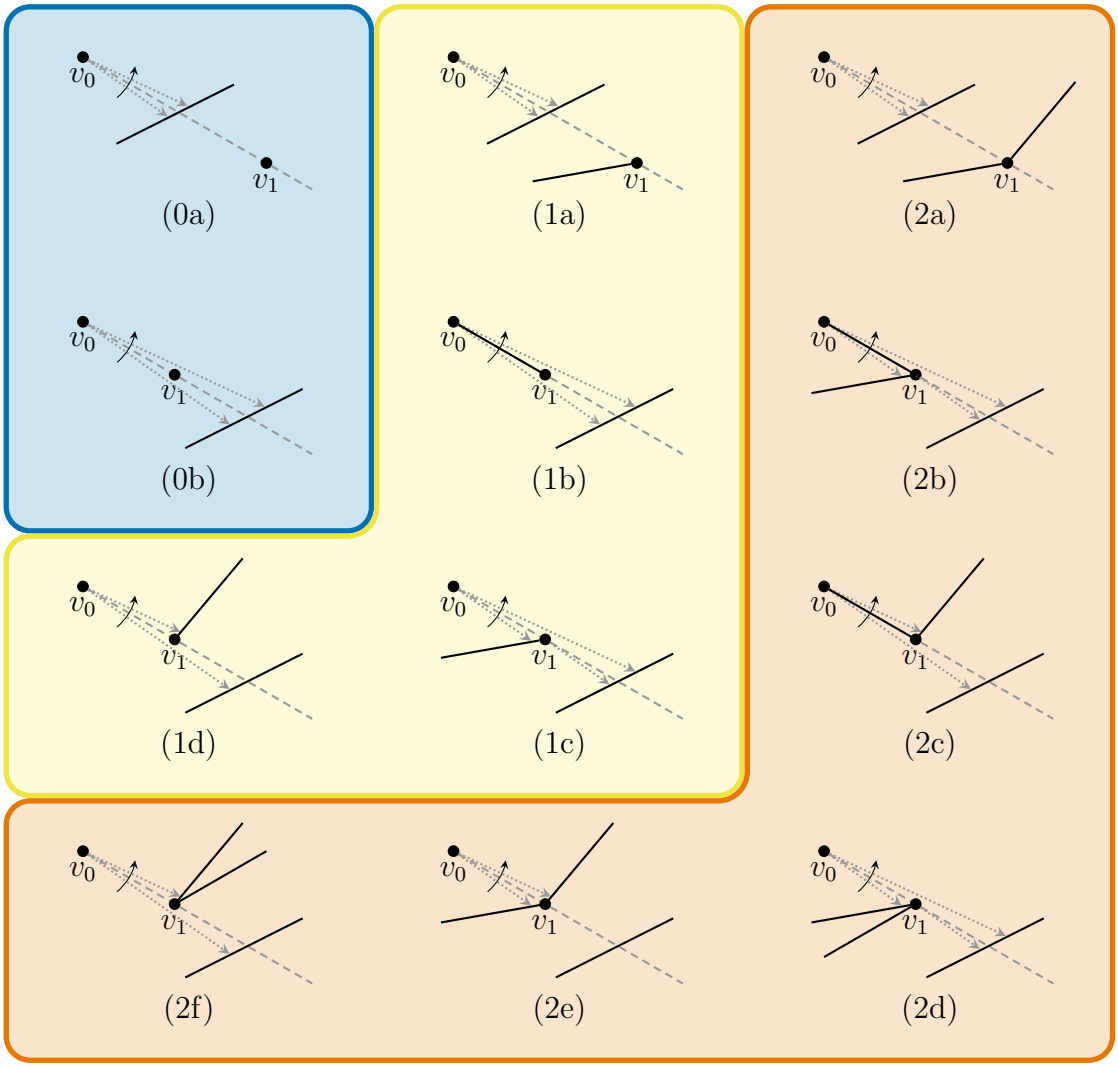


Figure A.3: Cases used when computing a visibility graph using Welzl's algorithm.

of vertices, rotating the current direction counter clockwise. When using the pair (v_0, v_1) with angle θ, the algorithm decides whether to add (v_0, v_1) to $\mathcal{E}$ and updates `view`(v). There are 12 possible cases (Figure A.3) which describe what should happen. These cases depend on whether v_1 is connected to 0, 1, or 2 edges, and how these edges are related to v_0 and its previous `view`. The previous `view` is the edge hit by a ray in direction $\theta - \epsilon$, whereas the updated `view` is the edge hit by a ray in the direction $\theta + \epsilon$ for some $\epsilon > 0$. The four cases where v_1 is part of a line segment were described by Welzl [197]. I extended the algorithm to more general geometries by adding the two point cases and six polygon cases.

The simplest case is when v_1 is a point (i.e. has no edges connected to it). These cases are in the blue portion of Figure A.3.

(0a) If v_1 is behind $\texttt{view}(v_0)$, it is not visible so we do not add anything to $\mathcal{E}$ and we do not change $\texttt{view}(v_0)$.

(0b) If v_1 is in front of $\texttt{view}(v_0)$, it is visible so we add (v_0, v_1) to $\mathcal{E}$. Since v_1 does not have any edges, the rays at $\theta - \epsilon$ and $\theta + \epsilon$ point to the same edge so $\texttt{view}(v_0)$ does not change.

The next case is when v_1 is part of a line segment (i.e. has exactly one edge). We use e to denote the edge incident to v_1. These cases are in the yellow portion of the figure and are the same cases as presented by Welzl [197].

(1a) If v_1 is behind $\texttt{view}(v)$, we do not change $\mathcal{E}$ or $\texttt{view}(v_0)$.

(1b) If v_1's neighbor is v_0, the edge between them blocks line-of-sight so we do not add anything to $\mathcal{E}$. Since the e has angle θ, the rays at $\theta - \epsilon$ and $\theta + \epsilon$ point to the same edge so $\texttt{view}(v_0)$ does not change.

(1c) If $\texttt{view}(v_0) = e$, then v_1 is visible from v_0 so we add (v_0, v_1) to $\mathcal{E}$. Since e only extends to v_1 and a ray at $\theta + \epsilon$ will not intersect with it. In this case, the rays emanating from v_0 and v_1 will intersect with the same edge so we set $\texttt{view}(v_0) = \texttt{view}(v_1)$.

(1d) If v_1 is in front of $\texttt{view}(v_0)$ but $\texttt{view}(v_0) \neq e$, then v_1 is visible so we add (v_0, v_1) to $\mathcal{E}$. At angle $\theta+\epsilon$, the ray will intersect with e instead of the previous $\texttt{view}(v_0)$ because v_1 is closer than $\texttt{view}(v_0)$. Therefore we set $\texttt{view}(v_0) = e$.

The most complex case is when v_1 is part of a polygon (i.e. has two edges). We use e_{CW} and e_{CCW} to refer to these edges (Figure A.4). These cases are in the orange portion of the figure.

(2a) If v_1 is behind $\texttt{view}(v)$, we do not change $\mathcal{E}$ or $\texttt{view}(v_0)$.

(2b) If $e_{\text{CW}} = (v_0, v_1)$ and $\texttt{view}(v_0) = e_{\text{CCW}}$, then e_{CW} blocks line-of-sight so we do not add anything to $\mathcal{E}$. As in case (1c), we set $\texttt{view}(v_0) = \texttt{view}(v_1)$.

(2c) If $e_{\text{CCW}} = (v_0, v_1)$ then e_{CCW} blocks line-of-sight so we do not add anything to $\mathcal{E}$. As in case (1d), we set $\texttt{view}(v_0) = e_{\text{CW}}$.

(2d) If $\texttt{view}(v_0) = e_{\text{CCW}}$ and e_{CW} is behind e_{CCW}, then v_1 is visible so we add (v_0, v_1) to $\mathcal{E}$. As in case (1c), we set $\texttt{view}(v_0)$ to $\texttt{view}(v_1)$.

(2d) If $\texttt{view}(v_0) = e_{\text{CCW}}$ and e_{CW} is on the opposite side of the ray, then v_1 is visible so we add (v_0, v_1) to $\mathcal{E}$. After passing over v_1, the ray will intersect with e_{CW} so we set $\texttt{view}(v_0) = e_{\text{CW}}$.

(2f) If v_1 is in front of $\texttt{view}(v_0)$ but $\texttt{view}(v_0) \neq e_{\text{CCW}}$ or e_{CW}, then v_1 is visible so we add (v_0, v_1) to $\mathcal{E}$. Similar to case (1d), the ray will now intersect with e_{CW} so we set $\texttt{view}(v_0) = e_{\text{CW}}$.

By following all of these cases, we can iterate through all of the angles while updating $\texttt{view}(\cdot)$ and adding edges to the visibility graph. The resulting graph contains all visible edges but does not distinguish between edges in free space and edges entirely in obstacles. We will need to remove all of these fully occluded edges. Since all the points of interest are in the free space, all the fully occluded edges are between two vertices of the boundary. We can find and remove all of these full occluded edges by iterating through the list of visible edges and checking the angle of the edge with the angle of the two edges in and out of the vertex. Assuming the outer polygon is oriented clockwise and the inner polygons are oriented counterclockwise, the fully occluded edges will lay between the in-edge and the out-edge of the polygon's vertex when rotating counterclockwise from the in-edge. Checking this criterion takes constant time and so we can then remove all fully occluded edges in $\mathcal{O}(|\mathcal{V}|^2)$ to get the final visibility graph which contains only edges through free space. Overall the whole algorithm takes $\mathcal{O}(|\mathcal{V}|^2)$.

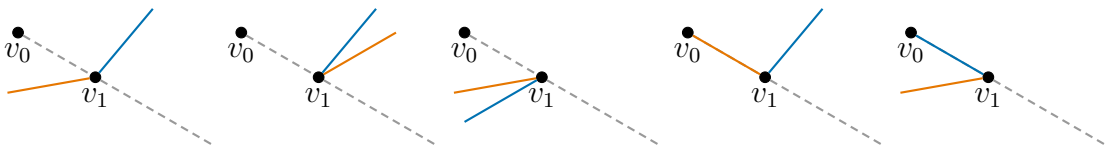


Figure A.4: Convention for which edge is labelled e_{CW} and which is labelled e_{CCW}. In most cases, e_{CW} is the first edge encountered when standing at v_1 and rotating clockwise from (v_1, v_0). When one of the edges coincides with (v_1, v_0), this definition is ambiguous. In these cases, if the other edge is in the left half plane, then the other edge is e_{CCW} and so $(v_1, v_0) = e_{\mathrm{CW}}$. Similarly, if the other edge is in the right half plane, then the other edge is e_{CW} and so $(v_1, v_0) = e_{\mathrm{CCW}}$.

Appendix B

Shortest path planning

Shortest path algorithms are an integral component of many robotic planning algorithms. In this appendix, I present three shortest path algorithms on graphs. Dijkstra's algorithm [48] and the A* algorithm [73] both solve the single-pair shortest path problem in $\mathcal{O}(|\mathcal{V}|^2)$. The Floyd-Warshall algorithm [60] solves the all-pairs shortest path problem in $\mathcal{O}(|\mathcal{V}|^3)$. All of these algorithms are guaranteed to find the exact solution in polynomial time. For robots with higher-dimensional configuration spaces or dynamic constraints on their motion, sampling-based methods, such as rapidly-exploring random trees (RRT) can be used to find short, feasible paths.

B.1 Dijkstra's algorithm

Dijkstra's algorithm computes the shortest paths from one vertex v_0 in a graph to all other vertices in the graph. It constructs a tree starting at v_0 and the shortest path to any vertex is the path along this tree (Figure B.1). Initially, the tree just contains v_0. In each round of the algorithm, a new vertex, v_{new}, is added to the tree connected by a single edge. This vertex is the closest vertex, when following edges of the graph, to v_0 which hasn't been added to the tree yet. When we add v_{new}, we connect it to v_{old}, the second-last vertex on the shortest path from v_{new}

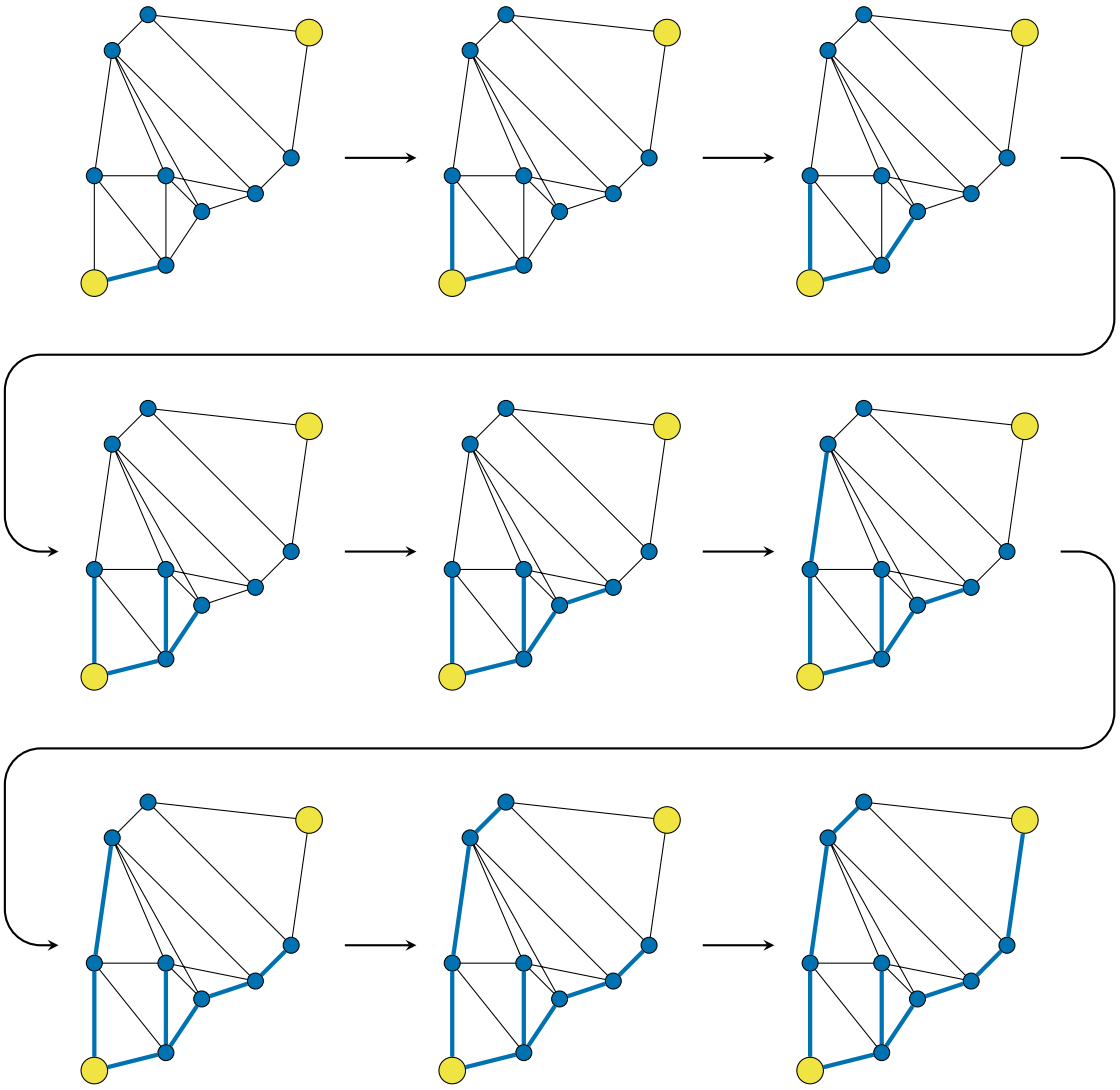


Figure B.1: Process of adding vertices to a tree during Dijkstra's algorithm. The next vertex added to the tree is the neighbor of the vertices of the tree which is closest to the start vertex when travelling along edges of the tree.

to v_{old}. This method of adding vertices guarantees that v_{old} is always already part of the tree when v_{new} is added to the tree. Computing the shortest path to v_{new} is easy because it is just the shortest path to v_{old} with v_{new} added to the end.

Since the shortest path to v_{new} is based on the shortest path to v_{old}, it is helpful to keep track of the shortest paths to each vertex in the tree and their lengths. In an efficient implementation of Dijkstra's algorithm (Algorithm B.1), we use two functions $\texttt{prev} : \mathcal{V} \to \mathcal{V}$ and $\texttt{dist} : \mathcal{V} \to \mathbb{R}_{\geq 0}$. The function $\texttt{prev}(\cdot)$ keeps track

of the shortest known path to each vertex using intermediate vertices which are all part of the tree. The function `dist`$(\cdot)$ stores the length of the shortest known path. When we add a new vertex to the tree, we check if there is a shorter path to each of the new vertex's neighbors by travelling through the new vertex. After the algorithm is complete, `prev`$(\cdot)$ encodes the structure of the tree. The shortest path from v_0 to v_1 can be recovered from `prev`$(\cdot)$ (Algorithm B.2).

Algorithm B.1: Dijkstra's algorithm

Input: Weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$; and start vertex, $v_0 \in \mathcal{V}$
Output: Shortest path, p, from v_0 to v_1

$\mathcal{V}_{\text{used}} \leftarrow \{\}$ `/* Set of already checked vertices */`
for vertex $v \in \mathcal{V}$ **do**
    `dist`$(v) \leftarrow \infty$
`dist`$(v_0) \leftarrow 0$
while $\mathcal{V}_{\text{used}} \neq \mathcal{V}$ **do**
    $v_{\text{new}} \leftarrow$ vertex in $\mathcal{V} \setminus \mathcal{V}_{\text{used}}$ which minimizes `dist`(v)
    $\mathcal{V}_{\text{used}} \leftarrow \mathcal{V}_{\text{used}} \cup \{v_{\text{new}}\}$
    for neighbor vertex $v \in$ `neighbors`(v_{new}) **do**
        if `dist`$(v_{\text{new}}) + w(v_{\text{new}}, v) <$ `dist`(v) **then**
            `dist`$(v) \leftarrow$ `dist`$(v_{\text{new}}) + w(v_{\text{new}}, v)$
            `prev`$(v) \leftarrow v_{\text{new}}$
$p \leftarrow$ path from v_0 to v_1 following `prev`$(\cdot)$ `/* Algorithm B.2 */`
return p

Algorithm B.2: Construct path (Dijkstra)

Input: Previous vertices, `prev` $: \mathcal{V} \to \mathcal{V}$; start vertex, v_0; and end vertex, v_1
Output: Path, p, from v_0 to v_1

$p \leftarrow$ path containing only v_1 `/* Constructed backwards */`
while last vertex of p is not v_0 **do**
    $v \leftarrow$ last vertex of p
    Append `prev`(v) to p
Reverse p
return p

B.2 The A* algorithm

When Dijkstra's algorithm constructs the shortest path tree, it adds every vertex to the tree. If we only care about finding the path to v_1, we could stop the algorithm as soon as the tree reaches v_1. Ideally, the algorithm would add v_1 early on so that it can be stopped early, but Dijkstra's algorithm doesn't actually use any information about v_1. It just adds the vertices in order based on their distance from v_0. If v_1 is the furthest away vertex from v_0, as was the case in Figure B.1, we can't stop the algorithm early.

A* is a variant of Dijkstra's algorithm which uses a heuristic to quickly add vertices that are likely to be close to v_1 so that we don't have to construct the whole tree. Instead of just using the distance from v_0 to v_{new} when choosing which vertex to add to the tree, it uses this distance plus a heuristic estimate of the distance from v_{new} to v_1. Although many possible heuristics could be used, we will use the Euclidean distance from v_{new} to v_1 ignoring any obstacles. With this criterion for selecting each v_{new}, A* ends up prioritizing vertices along what ends up being the path from v_0 to v_1 (Figure B.2). Compared with Dijkstra's algorithm, A* ends up adding v_1 sooner and thus needs fewer rounds of the algorithm to find the shortest path despite having the same theoretical complexity of $\mathcal{O}(|\mathcal{V}|^2)$.

The implementation of A* (Algorithm B.3) is quite similar to Dijkstra's algorithm (Algorithm B.1). The main change is that `dist`$(\cdot)$ no longer represents the distance from v_0 to v, but instead a heuristic distance which includes the Euclidean distance from v to v_1. To accommodate this change, the comparison of distances and update of `dist`$(\cdot)$ both use a new function, `heuristic`$(\cdot)$. This heuristic function is precomputed before the algorithm runs, which takes $\mathcal{O}(|\mathcal{V}|)$. For small graphs this precomputation may make A* marginally slower than Dijkstra's algorithm, but for large graphs, the cost of precomputing `heuristic`$(\cdot)$ is worth the decreased number of iterations needed to add v_1 to the tree. The

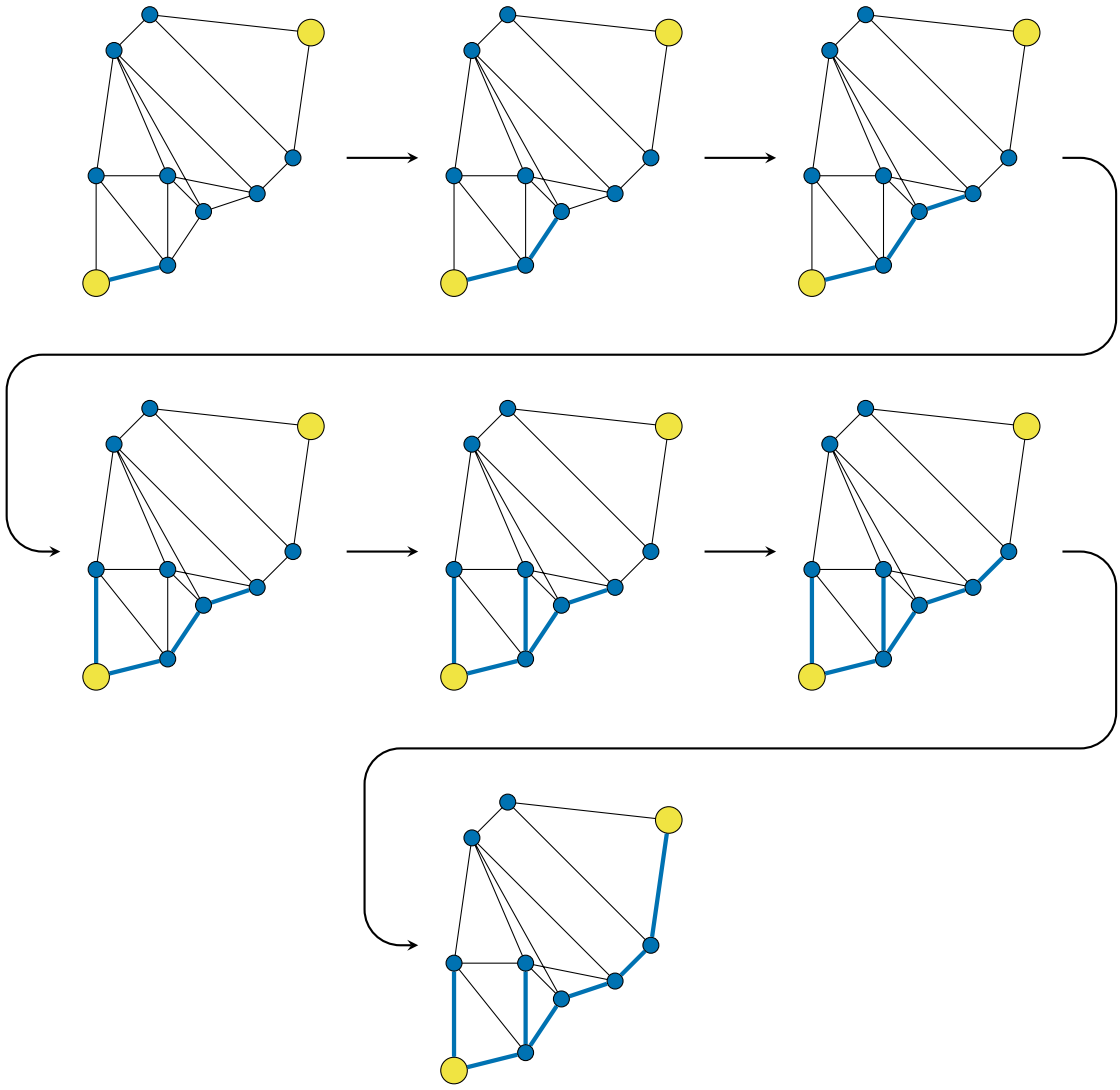


Figure B.2: Process of adding vertices to a tree during the A* algorithm. The next vertex is chosen using a heuristic which is the distance along the tree to the new vertex plus the straight-line distance from the new vertex to the end. This heuristic guides the tree towards the goal.

termination criterion for the main loop has also been changed so that it stops as soon as v_1 has been added to the tree instead of continuing until all vertices have been added. Once Algorithm B.3 has terminated, the shortest path from v_0 to v_1 can be computed from `dist`$(\cdot)$ using Algorithm B.2.

Algorithm B.3: A* algorithm

Input: Weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$; start vertex, v_0; and end vertex, v_1
Output: Shortest path, p, from v_0 to v_1

```
1  V_used ← {}                          /* Set of already checked vertices */
2  for vertex v ∈ V do
3  |  dist(v) ← ∞
4  |  heuristic(v) ← Euclidean distance from v to v_1
5  dist(v_0) ← 0
6  while v_1 ∉ V_used do
7  |  v_new ← vertex in V \ V_used which minimizes dist(v)
8  |  V_used ← V_used ∪ {v_new}
9  |  for neighbor vertex v ∈ neighbors(v_new) do
10 |  |  if dist(v_new) + w(v_new, v) + heuristic(v) < dist(v) then
11 |  |  |  dist(v) ← dist(v_new) + w(v_new, v) + heuristic(v)
12 |  |  |  prev(v) ← v_new
13 p ← path from v_0 to v_1 following prev(·)          /* Algorithm B.2 */
14 return p
```

B.3 The Floyd-Warshall algorithm

When a robot is in the same environment for a long time, it will have to find lots of shortest paths throughout that environment. Suppose there are $|\mathcal{V}|$ points of interest, and the robot may have to navigate between any two of them. Then there are $\mathcal{O}(|\mathcal{V}|^2)$ possible pairs of points and computing all the shortest paths using Dijkstra's algorithm or A* would take $\mathcal{O}(|\mathcal{V}|^4)$. Can we do better than this? Yes! The Floyd-Warshall algorithm can compute all-pairs shortest paths on a graph in $\mathcal{O}(|\mathcal{V}|^3)$.

The key idea of this algorithm is to combine the best known paths from v_i to v_k and from v_k to v_j if their combination is shorter than the best known path from v_i to v_j (Figure B.3). Initially, the shortest paths just consist of all edges of the graph. Then the algorithm iterates through all vertices of the graph, and for each vertex it tries to use it as an intermediate vertex on the shortest path between each pair of vertices (Figure B.4). After rounds $1, \ldots, k$, any path from v_i to v_j is guaranteed to optimal for the subgraph induced by the vertices $\{v_i, v_j, v_1, \ldots, v_k\}$.

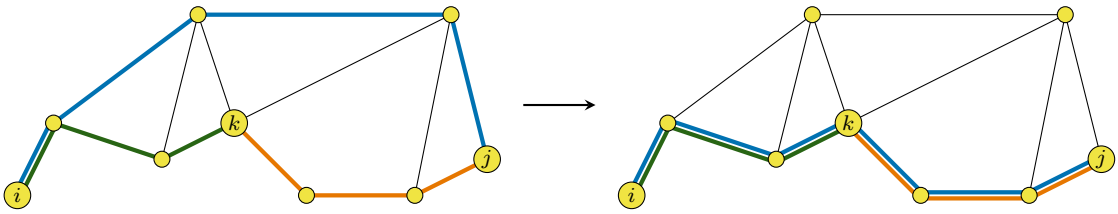


Figure B.3: The Floyd-Warshall algorithm is based on checking if the path from v_i to v_j via v_k is shorter than the previous best known path from v_i to v_j. In each round of the algorithm, a new vertex can be used as an intermediate vertex. Before v_k can be used as an intermediate vertex (left), the path from v_i to v_j is not optimal but the paths from v_i to v_k and from v_k to v_j are optimal as the necessary intermediate vertices are already allowed. After v_k is allowed as an intermediate vertex (right), the path from v_i to v_j is improved by combining the shortest paths from v_i to v_j and from v_j to v_k.

After all the rounds, all of the paths are guaranteed to be optimal on the whole graph for all pairs of vertices. Overall, the algorithm iterates over all vertices as intermediate vertices and for each possible intermediate vertex it iterates over all pairs of vertices, resulting in an algorithm that runs in $\mathcal{O}(|\mathcal{V}|^3)$.

Similar to the implementations of Dijkstra's algorithm and A*, we can implement the Floyd-Warshall algorithm (Algorithm B.4) using two data structures, $\texttt{next} : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$ and $\texttt{dist} : \mathcal{V} \times \mathcal{V} \to \mathbb{R}_{\geq 0}$. During the algorithm, $\texttt{dist}(v_i, v_j)$ stores the length of the best known path from v_i to v_j. Initially $\texttt{dist}(v_i, v_j) = w(v_i, v_j)$ if there is an edge between v_i to v_j and is infinite otherwise. In each round of the main loop, the algorithm compares the best known path from v_i to v_j, with the best known paths from v_i to v_k and from v_k to v_j. All of these best known paths only use $\{v_1, \ldots, v_{k-1}\}$ as intermediate vertices. If the path from v_i to v_k followed by the path from v_k to v_j is shorter, then their combination is the shortest path from v_i to v_j via $\{v_1, \ldots, v_k\}$ and the distance is updated accordingly. The structure of the shortest paths is encoded using the function $\texttt{next}(\cdot, \cdot)$ which, when of its second arguments is constant, works similar to $\texttt{prev}(\cdot)$ from Algorithm B.1 or Algorithm B.3. The vertex stored in $\texttt{next}(v_i, v_j)$

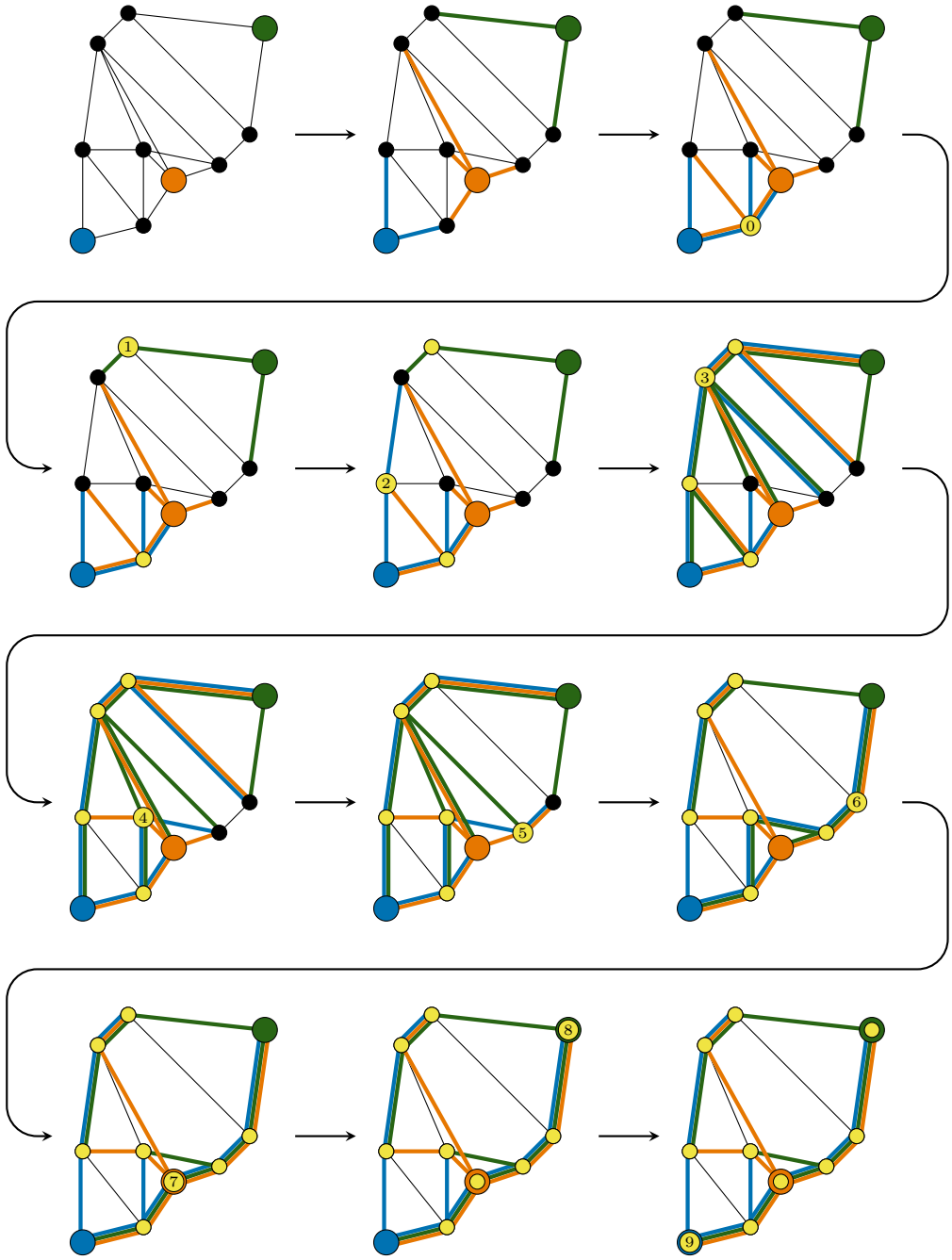


Figure B.4: In each iteration of the Floyd-Warshall algorithm, the shortest paths are rewired to include an additional vertex as an intermediate vertex between two pairs of points. In this illustration, we just show the shortest paths that connect to three of the vertices. As more vertices are allowed as intermediate vertices, these trees rearrange and become closer to optimal. When the algorithm finishes, it has produced the optimal shortest path tree for each vertex in the graph.

is the vertex immediately after v_i on the best known path from v_i to v_j. It is initially v_j if there is an edge between v_i and v_j and is undefined otherwise. When two paths are combined, $\texttt{next}(v_i, v_j)$ is updated to contain the first vertex of the first of the two paths that are combined. After the Floyd-Warshall algorithm has finished, the shortest paths between any pair of vertices can be reconstructed in $\mathcal{O}(|\mathcal{V}|)$ just using $\texttt{next}(\cdot, \cdot)$ (Algorithm B.5). Note that I have presented the Floyd-Warshall algorithm for a symmetric graph, although it is easy to modify for asymmetric graphs.

Algorithm B.4: Floyd-Warshall algorithm

Input: Weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$
Output: Map used to construct shortest paths, $\texttt{next} : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$

for pair of vertices $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$ **do**
    $\texttt{dist}(v_i, v_j) \leftarrow \infty$
for edge $(v_i, v_j) \in \mathcal{E}$ **do**
    $\texttt{dist}(v_i, v_j) \leftarrow w(v_i, v_j)$
    $\texttt{dist}(v_j, v_i) \leftarrow \texttt{dist}(v_j, v_i)$
    $\texttt{next}(v_i, v_j) \leftarrow v_j$
    $\texttt{next}(v_j, v_i) \leftarrow v_i$
for vertex $v_k \in \mathcal{V}$ **do**
    for pair of vertices $(v_i, v_j) \in \mathcal{V} \times \mathcal{V}$ **do**
        if $\texttt{dist}(v_i, v_k) + \texttt{dist}(v_k, v_j) < \texttt{dist}(v_i, v_j)$ **then**
            $\texttt{dist}(v_i, v_j) \leftarrow \texttt{dist}(v_i, v_k) + \texttt{dist}(v_k, v_j)$
            $\texttt{dist}(v_j, v_i) \leftarrow \texttt{dist}(v_j, v_i)$
            $\texttt{next}(v_i, v_j) \leftarrow \texttt{next}(v_i, v_k)$
            $\texttt{next}(v_j, v_i) \leftarrow \texttt{next}(v_j, v_k)$
return $\texttt{next}(\cdot, \cdot)$

Algorithm B.5: Construct path (Floyd-Warshall)

Input: Map used to construct shortest paths, `next` : $\mathcal{V} \times \mathcal{V} \to \mathcal{V}$; start vertex, v_0; and end vertex, v_1

Output: Path, p, from v_0 to v_1

```
1 p ← path containing only v_0                    /* Constructed forwards */
2 while last vertex of p is not v_1 do
3 |   v ← last vertex of p
4 |   Append next(v, v_1) to p
5 return p
```

Appendix C

Minimum spanning trees

A spanning tree is a subgraph of a connected graph which is a tree and visits every vertex of the graph (Figure C.1). A minimum spanning tree (MST) has the shortest length of any spanning tree for that graph. Although MSTs are not used directly in path planning, several solutions to the travelling salesperson problem use MSTs which makes them relevant to path planning. In this appendix, I present two exact algorithms for computing MSTs: Kruskal's algorithm [112] which runs in $\mathcal{O}(|\mathcal{E}|\log(|\mathcal{V}|))$ and Prim's algorithm [154] which runs in $\mathcal{O}(|\mathcal{V}|^2)$. Kruskal's algorithm produces a set of edges sorted from shortest to longest and is faster on sparse graphs. Prim's algorithm produces a tree rooted at a specific vertex and is faster on dense graphs.

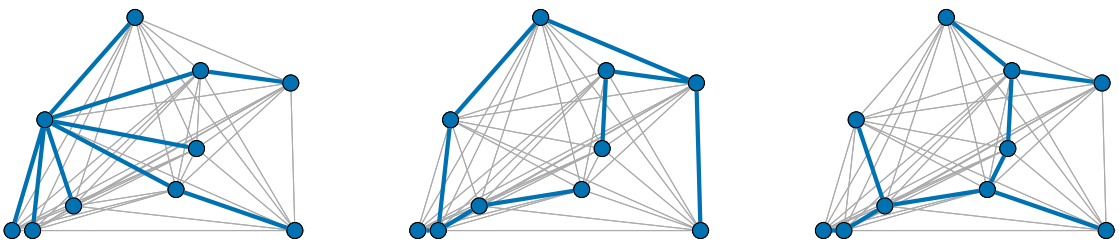


Figure C.1: A spanning tree (left) is a subtree which connects every vertex of a graph. A minimum spanning tree (right) is the shortest length spanning tree.

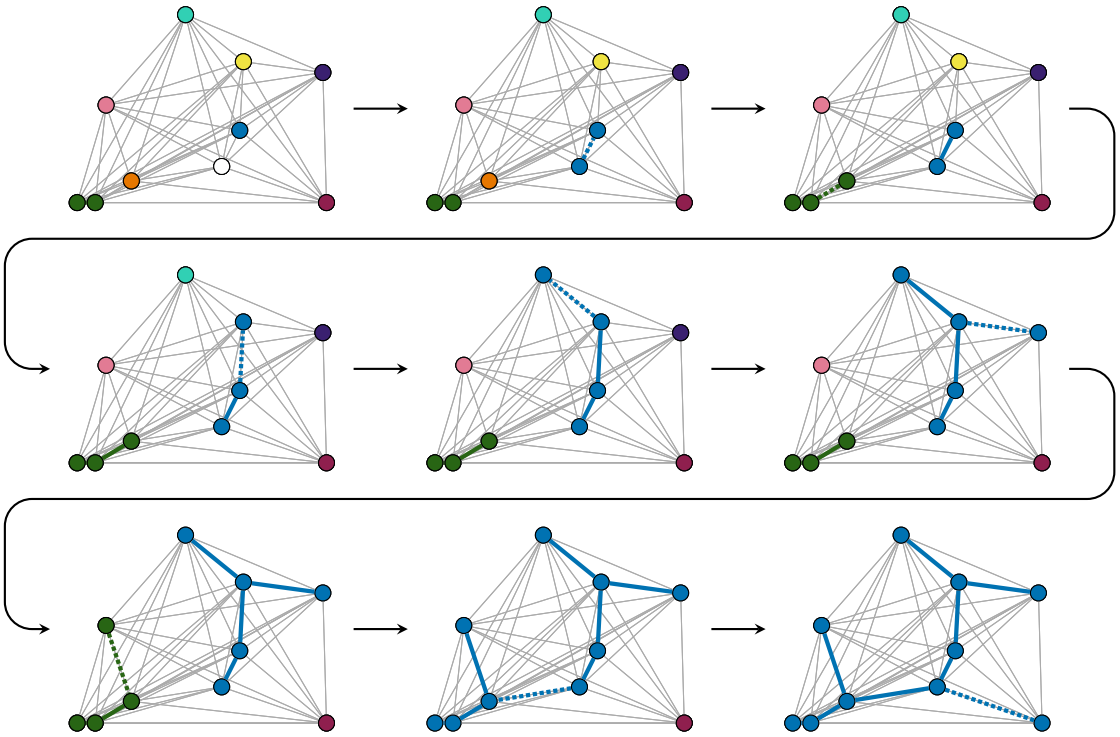


Figure C.2: Kruskal's algorithm builds an MST by adding the shortest edge of the graph which would not result in a cycle to the set of edges. After each round, the number of subtrees is reduced by 1.

C.1 Kruskal's algorithm

An alternate definition of a spanning tree is a collection of $|\mathcal{V}|-1$ edges which does not contain any cycles. Kruskal's algorithm (Figure C.2) constructs a spanning tree by adding edges which would not create a cycle with the edges already added. At each step of the algorithm, the current set of edges creates a set of subtrees. Any new edge which is added must have endpoints in different subtrees so that it does not produce a cycle. When this new edge is added, it merges the two subtrees. After $|\mathcal{V}|-1$ merges, all of the subtrees have been merged into a single acyclic graph which is a spanning tree. By always adding the shortest edge that connects two subtrees, the set of subtrees at each step is a minimum spanning forest, and the final spanning tree is an MST.

When implementing Kruskal's algorithm (Algorithm C.1), the edges must first

be sorted by length and then we can iterate through the sorted list of edges to add the shortest edges to the MST. Sorting the edges takes $\mathcal{O}(|\mathcal{E}|\log(|\mathcal{E}|)) = \mathcal{O}(|\mathcal{E}|\log(|\mathcal{V}|))$ which uses the most computation of any step of the algorithm. After the edges have been sorted, the MST is constructed by iterating through the edges and adding an edge if it connects distinct subtrees. Initially, all vertices have their own subtree. When a new edge is added, all vertices in one of the edge's vertex's subtree get relabelled so they are in the other vertex's subtree. In this way, the two subtrees get merged. Once $|\mathcal{V}|-1$ edges have been added, all subtrees have been merged and so the main loop can be terminated as no more edges can be added.

Algorithm C.1: Kruskal's algorithm

Input: Weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$
Output: Minimum spanning tree edges, $\mathcal{E}_{\text{MST}} \subset \mathcal{E}$

```
1  for vertex v ∈ V do
2      subtree(v) ← v
3  Sort E from shortest to longest
4  E_MST ← {}                                   /* Edges of the MST */
5  for edge (v_i, v_j) ∈ E do
6      if subtree(v_i) ≠ subtree(v_j) then
7          Add e to E_MST
8          for vertex v ∈ V \ {v_j} do
9              if subtree(v) = subtree(v_j) then
10                 subtree(v) ← subtree(v_i)
11             subtree(v_j) ← subtree(v_i)
12 return E_MST
```

C.2 Prim's algorithm

Another property of a spanning tree is that it contains exactly one path from a root vertex to any other vertex in the tree. Prim's algorithm (Figure C.3) constructs a spanning tree by adding an edge which connects one more vertex to the root vertex in each iteration. It maintains a single tree and several isolated vertices. When

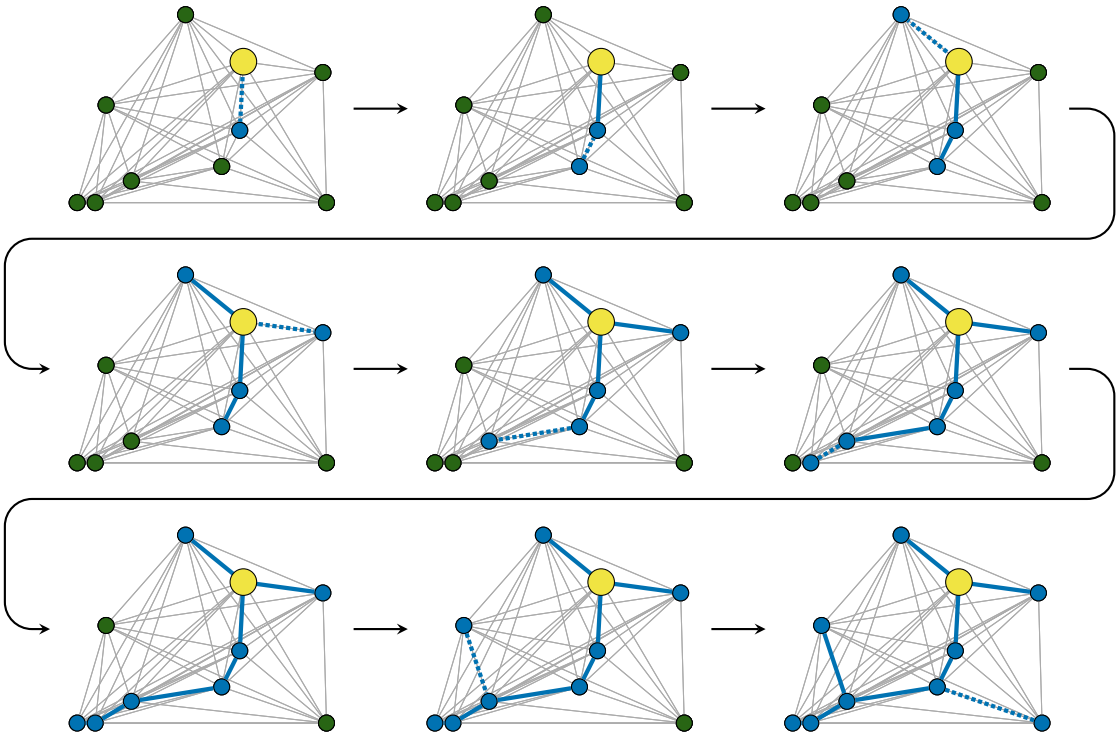


Figure C.3: Prim's algorithm builds an MST by adding the shortest edge which connects the current non-spanning tree with a vertex outside of this tree. This construction results in an MST which is stored as a tree rooted at the initial vertex (yellow vertex).

adding an edge it finds the shortest edge which connects the existing non-spanning tree with any of the vertices not in the tree. At any time, the tree is the shortest tree that reaches its current set of vertices. After the final round, the tree spans the entire graph and is therefore an MST.

When implementing Prim's algorithm (Algorithm C.2), the tree is stored by keeping track of the parent of each tree vertex and the shortest distance from each non-tree vertex to the tree. Initially, the root vertex is added to the tree and the shortest distances for all the other vertices (all non-tree vertices) is its distance from the root. In each round, we add the closest vertex to the tree and then update the shortest distances for all remaining non-tree vertices by checking if they are closer to the just-added vertex than to the previous closest vertex of the tree. When updating the shortest distance, we also store the tree vertex minimizing the

distance as the parent of the non-tree vertex. The parent of a non-tree vertex can therefore change during the algorithm, but once a vertex is added to the tree, its parent can no longer change. Once all vertices have been added to the tree, each vertex is connected to the root vertex by a chain of parent vertices and the tree formed by the vertex-parent edges is the MST.

Algorithm C.2: Prim's algorithm

Input: Weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$
Output: Minimum spanning tree edges, $\mathcal{E}_{\text{MST}}$

$v_0 \leftarrow$ random vertex of $\mathcal{V}$ `/* Root vertex of MST */`
$\mathcal{V}_{\text{tree}} \leftarrow \{\}$ `/* Vertices already in tree */`
$\mathcal{E}_{\text{MST}} \leftarrow \{\}$ `/* Edges of the MST */`
for $v \in \mathcal{V}$ **do**
    $\texttt{dist}(v) \leftarrow \infty$ `/* Distance to nearest vertex of` $\mathcal{V}_{\text{tree}}$ `*/`
$\texttt{dist}(v_0) = 0$
while $\mathcal{V}_{\text{tree}} \neq \mathcal{V}$ **do**
    $v_{\text{new}} \leftarrow$ vertex in $\mathcal{V} \setminus \mathcal{V}_{\text{tree}}$ which minimizes $\texttt{dist}(v)$
    $\mathcal{V}_{\text{tree}} \leftarrow \mathcal{V}_{\text{tree}} \cup \{v_{\text{new}}\}$
    if $|\mathcal{V}_{\text{tree}}| > 1$ **then**
        $\mathcal{E}_{\text{MST}} \leftarrow \mathcal{E}_{\text{MST}} \cup \{(v_{\text{new}}, \texttt{parent}(v_{\text{new}}))\}$
    for $v \in \texttt{neighbors}(v_{\text{new}}) \cap \mathcal{V} \setminus \mathcal{V}_{\text{tree}}$ **do**
        if $w(v, v_{\text{new}}) < \texttt{dist}(v)$ **then**
            $\texttt{dist}(v) \leftarrow w(v, v_{\text{new}})$
            $\texttt{parent}(v) \leftarrow v_{\text{new}}$
return $\mathcal{E}_{\text{MST}}$

Appendix D

Travelling salesperson algorithms

Algorithms that solve the travelling salesperson problem (TSP) are very valuable in robotics. They provide near optimal plans for many different problems where the order of a robot's spatially distributed tasks needs to be determined. In particular, for my thesis, the TSP is important because I use a constrained version of it to plan coverage paths. As the TSP is NP-hard [150], there are no known polynomial time algorithms for solving it exactly and we instead rely on heuristics. In this appendix, I present three heuristics which, in my opinion are the most important TSP algorithms. Christofides' algorithm is a deterministic algorithm which runs in $\mathcal{O}(|\mathcal{V}|^3)$ and is guaranteed to produce a cycle which is at most one-and-a-half times longer than the optimal cycle. The other two heuristics, 2-opt [44] and the Lin–Kernighan (LK) heuristic [122], make incremental changes to improve a cycle and are randomized so that they can produce many high quality cycles.

D.1 Christofides' algorithm

The objective of the TSP is to find a spanning cycle—a closed path which visits every vertex of a graph. A spanning cycle can also be described as a spanning subgraph where each vertex has exactly 2 edges. Suppose, instead, we have a spanning subgraph where each vertex has an even number of edges. We can easily convert

such a graph into a spanning cycle by a process called shortcutting (Figure D.1). In this process, we follow a path which visits every edge of the graph exactly once and add each vertex to the cycle the first time we encounter it on this path (Algorithm D.1). For a graph which satisfies the triangle inequality, shortcutting always results in a shorter cycle than the original even spanning subgraph. If we can find a low-weight even spanning subgraph, then we can use this shortcutting algorithm to get a short spanning cycle which is an approximate solution to the TSP.

Algorithm D.1: Shortcutting

Input: Spanning subgraph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, with even vertex degrees
Output: Spanning cycle, c

```
v ← random vertex of V
c ← path containing only v
while E ≠ {} do
    e ← edge of E with v as endpoint
    E ← E \ {e}
    v ← other endpoint of e
    if v ∉ c then
        Append v to c
Append first vertex of c to c
return c
```

Christofides algorithm is a deterministic algorithm for the TSP based on the construction of an even spanning subgraph from an MST (Figure D.2, Algorithm D.2). The MST is already a spanning subgraph, but it is not guaranteed to have all even-degree vertices (in fact, it always has at least two degree-one vertices). To convert the MST into an even spanning subgraph, one edge must be added for each odd-degree vertex. Let $\mathcal{V}_{\text{odd}}$ be the set of odd-degree vertices in the MST. This set is guaranteed to contain an even number of vertices, and so it is possible to pair them up so that exactly one new edge is connected to each vertex. Such a set of edges is called a *perfect matching* and the minimum perfect matching (MPM) can be found in polynomial time. There are several algorithms for computing MPMs [43], such as Lawler's algorithm [118] which runs in $\mathcal{O}(|\mathcal{V}|^3)$.

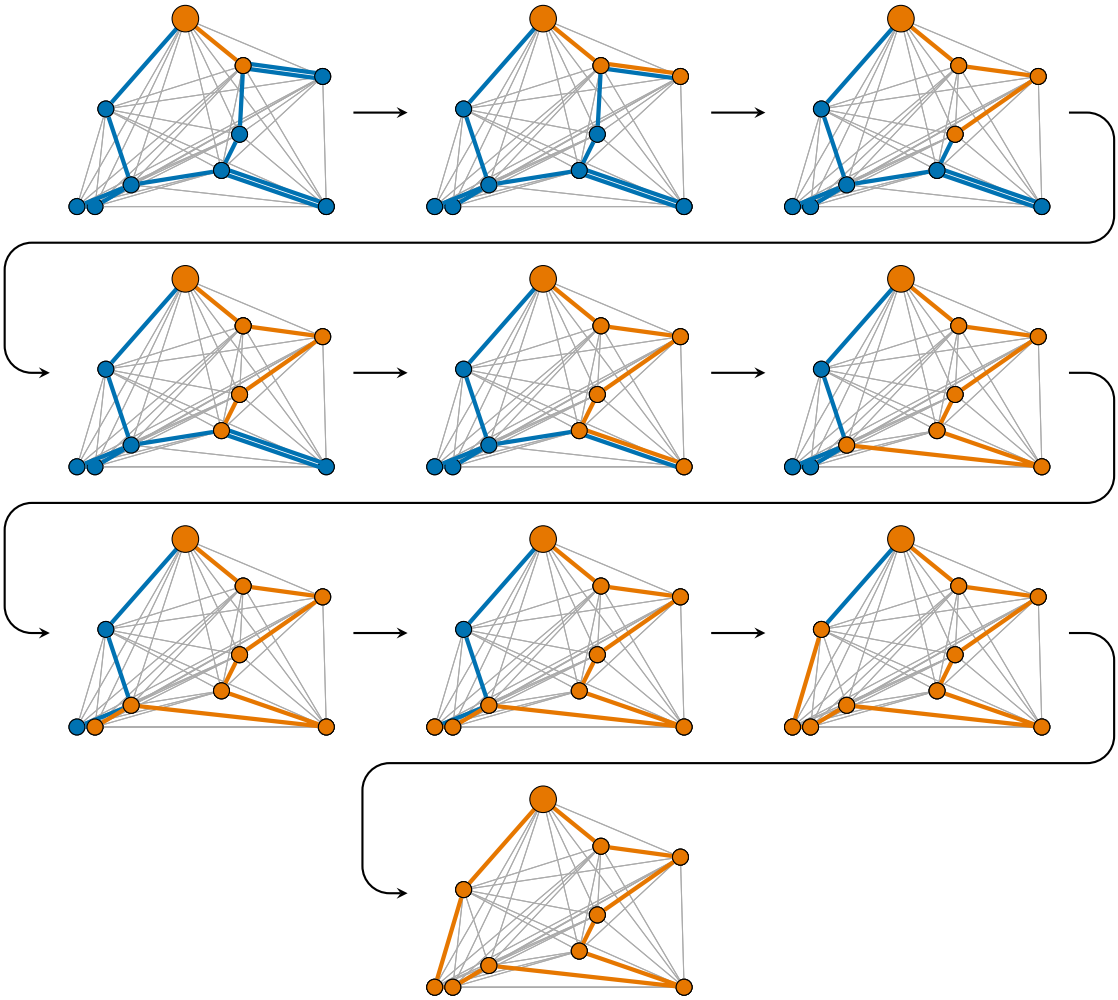


Figure D.1: The final step of Christofides algorithm is a shortcutting procedure which constructs a tour from a spanning graph whose vertices all have even degree. Starting at any vertex, it follows unused edges of the original graph until it reaches an unused vertex. It then replaces all of the edges needed to travel between those vertices with a direct edge, effectively reducing the degree of all intermediate vertices by 2. Once this procedure reaches the original vertex, all vertices will have degree 2 and the graph is a spanning cycle.

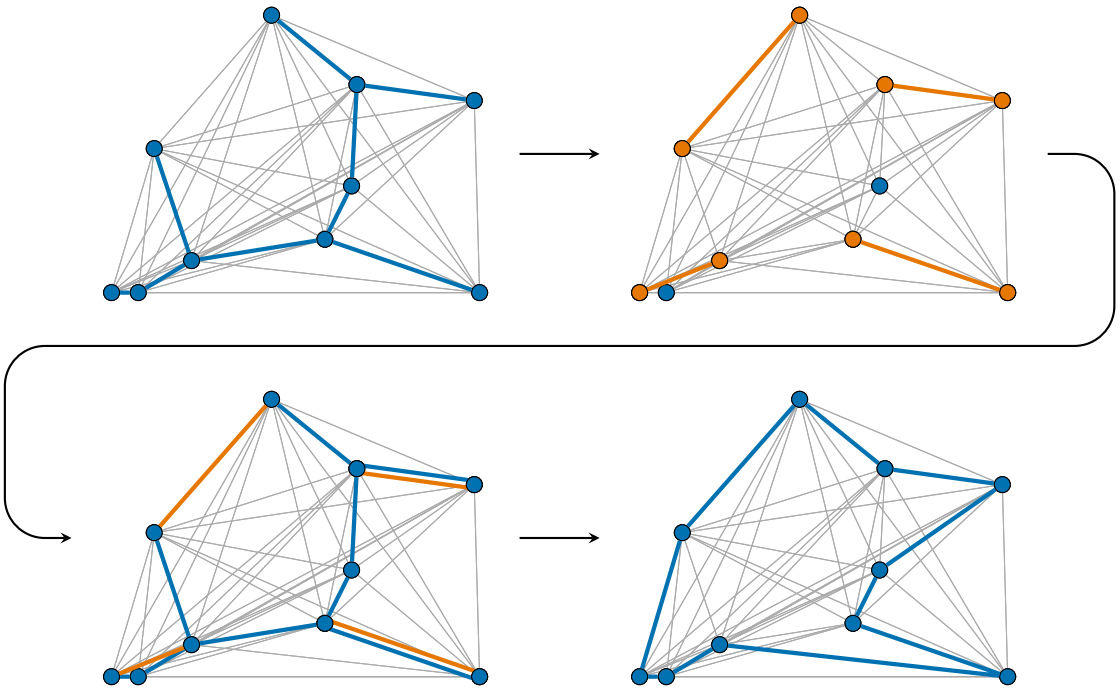


Figure D.2: Christofides' algorithm for solving the TSP uses an MST (top left), and a MPM (top right) to construct a spanning graph where each vertex has even degree (bottom left). This spanning graph can be converted to a cycle by "shortcutting" (Figure D.1) which decreases the node of a single vertex by 2.

These algorithms are based on linear programming and I am not aware of an intuitive combinatorial explanation for how they work. Combining the MST with the MPM on $\mathcal{V}_{\text{odd}}$ results in an even spanning subgraph which is turned into a spanning cycle by shortcutting.

Algorithm D.2: Christofides' algorithm

Input: Complete weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$
Output: Spanning cycle, c

1 $\mathcal{E}_{\text{MST}} \leftarrow$ minimum spanning tree of $\mathcal{G}$ `/* Algorithm C.1 or C.2 */`
2 $\mathcal{V}_{\text{odd}} \leftarrow$ vertices of $\mathcal{G}$ with odd degree in the MST
3 $\mathcal{E}_{\text{MPM}} \leftarrow$ edges of the minimum perfect matching of $\mathcal{V}_{\text{odd}}$
4 $c \leftarrow$ spanning cycle on $\mathcal{G}' = (\mathcal{V}, \mathcal{E}_{\text{MST}} \cup \mathcal{E}_{\text{MPM}})$ `/* Algorithm D.1 */`
5 **return** c

The result of Christofides' algorithm is a spanning cycle whose length is guaranteed to be at most one-and-a-half times the length of the TSP solution. To

understand this bound, we first bound the MST and MPM. Removing an edge from the TSP solution results in a spanning tree whose length is less than the TSP length. The MST is shorter than any other spanning tree, such as the one obtained by removing an edge from the TSP solution and therefore

$$\ell(\text{MST}) \leq \ell(\text{TSP}). \tag{D.1}$$

For an even subset of vertices, such as $\mathcal{V}_{\text{odd}}$, we can use the TSP solution to get a spanning cycle on these vertices which is at most the same length as the TSP solution. This spanning cycle contains an even number of edges and can be split into two subsets by taking every other edge of the cycle. Both of these subsets are perfect matchings on $\mathcal{V}_{\text{odd}}$. The shorter of these matchings is half the length of the TSP solution or shorter and, as the MPM is the shortest perfect matching,

$$\ell(\text{MPM}) \leq \frac{1}{2}\ell(\text{TSP}). \tag{D.2}$$

Combining the bounds (D.1)–(D.2), we get the bound for the spanning cycle produced by Christofides' algorithm:

$$\ell(c) \leq \ell(\text{MST}) + \ell(\text{MPM}) \leq \ell(\text{TSP}) + \frac{1}{2}\ell(\text{TSP}) = \frac{3}{2}\ell(\text{TSP}).$$

In many cases, $\ell(c)$ will actually be much closer to the length of the TSP solution.

Overall, the complexity of Christofides' algorithm is $\mathcal{O}(|\mathcal{V}|^3)$. The MST can be computed by Prim's algorithm in $\mathcal{O}(|\mathcal{V}|^2)$ and the MPM by Lawler's algorithm in $\mathcal{O}(|\mathcal{V}|^3)$. As these algorithms are performed consecutively, the complexity of computing the even spanning subgraph is $\mathcal{O}(|\mathcal{V}|^3)$. Shortcutting to get the final spanning cycle is linear in the number of edges. In this case, there are $|\mathcal{V}| - 1$ edges in the MST and at most $\frac{1}{2}|\mathcal{V}|$ edges in the MPM and so shortcutting will take $\mathcal{O}(|\mathcal{V}|)$ and not change the overall complexity of Christofides' algorithm.

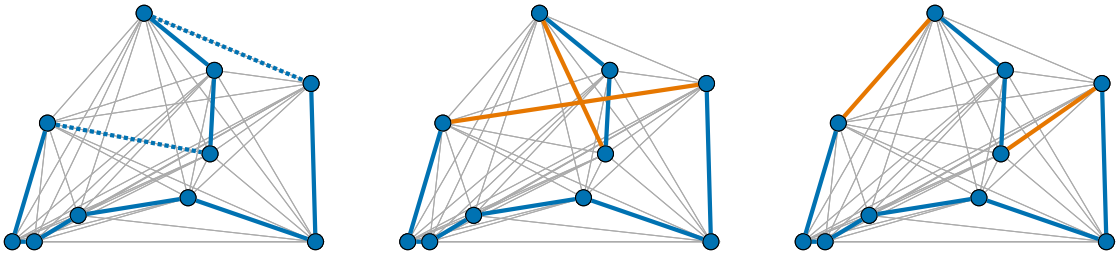


Figure D.3: In 2-opt, 2 edges of a cycle are removed (left). There are two possible ways to reconnect the partial paths. One choice results in a disconnected path (center), whereas the other results in a new cycle (right).

D.2 2-opt

Combinatorial optimization problems can, in general be described as "from a finite set of possible solutions, find a feasible solution that minimizes a cost function". A basic heuristic method for solving any combinatorial optimization is using a transformation which can convert a feasible solution into one of several "nearby" feasible solutions. Starting with an initial feasible solution, we repeatedly apply the transformation and replace the current solution with the transformed one if the transformed one has a lower cost. The TSP is a combinatorial optimization problem where the feasible solutions are spanning cycles and the cost function is the length of the cycle.

2-opt is a transformation that can be used to convert one spanning cycle into another spanning cycle by replacing two edges of the cycle with two new edges (Figure D.3). When the two edges are removed, there are two possible ways to reconnect the two paths but only one results in a spanning cycle. Therefore each pair of edges in the spanning cycle results in a unique 2-opt transformation and so there are $\mathcal{O}(|\mathcal{V}|^2)$ possible transformations from each cycle.

Repeatedly transforming a spanning cycle by 2-opt to reduce its length is a simple heuristic which can quickly improve a spanning cycle (Figure D.4, Algorithm D.3). In each round a pair of edges is replaced with a shorter pair of edges.

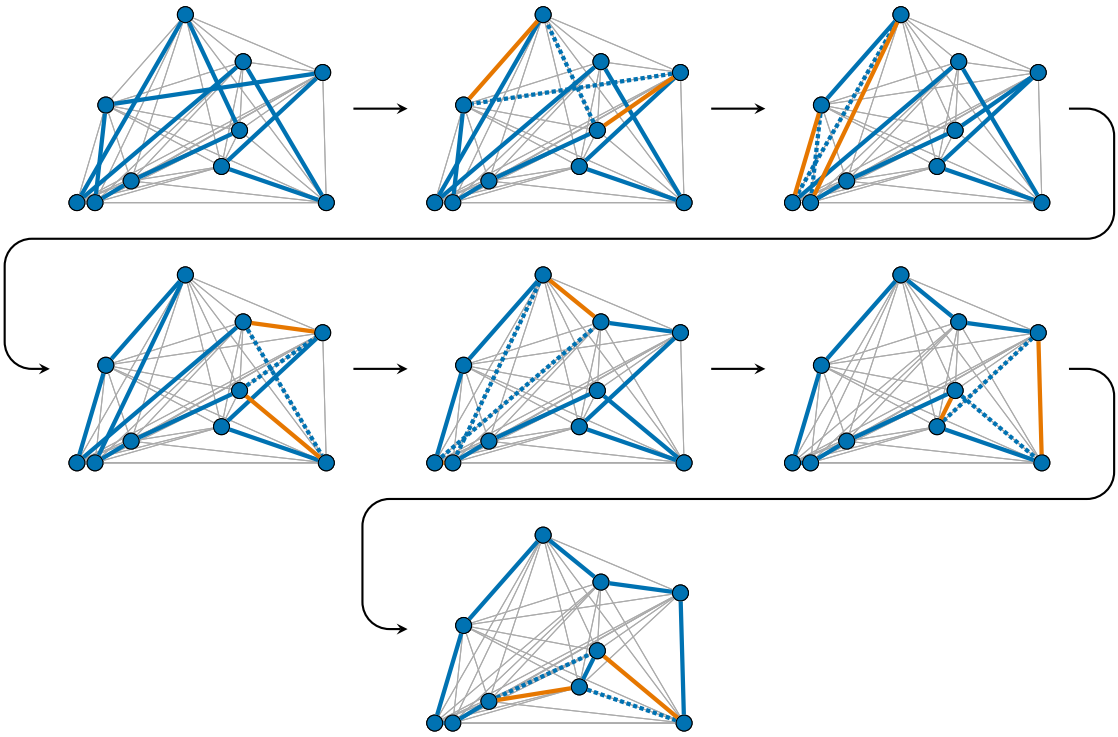


Figure D.4: 2-opt is a heuristic for reducing the length of a spanning cycle by repeatedly removing 2 edges and replacing them with 2 new edges. When a pair of edges is removed, there is only one choice of new edges which results in another spanning cycle (Figure D.4).

Eventually, the algorithm will obtain a cycle which cannot be improved by any of the 2-opt moves—a local minimum—and it returns this cycle. As the heuristic is randomized, it can be run multiple times to find different local minima. For small problems, there are few enough local minima that it can find the global minimum after a few runs. However, for large problems, there are too many global minima and 2-opt is unlikely to find the optimal cycle.

D.3 Lin–Kernighan heuristic

In combinatorial optimization, local minima occur whenever a solution is better than any *adjacent* solution. What do I mean by adjacent? Two solutions are adjacent if there is a single step transformation which converts one solution into the other. For 2-opt, two solutions are adjacent if they differ by exactly 2 edges.

Algorithm D.3: 2-opt heuristic

Input: Complete weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$
Output: Spanning cycle, c

```
1 c ← random spanning cycle of G
2 while true do
3     for edges e_1, e_2 of c do
4         e'_1, e'_2 ← edges obtained by interchanging endpoints of e_1 and e_2 to
            maintain cycle
5         if w(e'_1) + w(e'_2) < w(e_1) + w(e_2) then
6             Replace e_1, e_2 of c with e'_1, e'_2
7             break
8     if c has not been improved then
9         return c
```

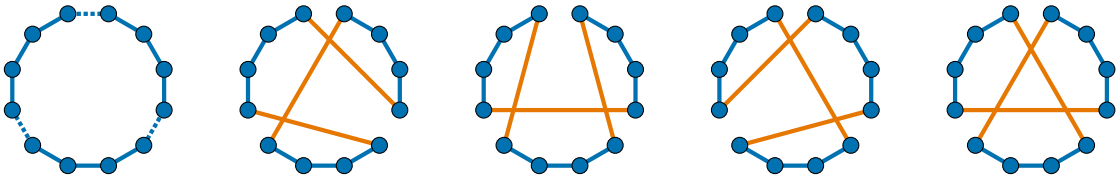


Figure D.5: In 3-opt, 3 edges of a cycle are removed (left). There are four possible ways to reconnect the partial paths into cycles (right 4).

If I were to use a different transformation to modify spanning cycles, there would be a different notion of adjacency, and the set of local minima could change.

Ideally, we would use a transformation which results in very few local minima. Perhaps the easiest way to decrease the number of local minima is to increase the number of adjacent solutions. 2-opt can be extended to 3-opt, 4-opt, or higher by allowing more edges to be interchanged simultaneously. If we use 3-opt, the number of local minima will decrease but in each round, we will have to check $\mathcal{O}(|\mathcal{V}|^3)$ triples of edges to break. Furthermore, when 3 edges are broken, there are 4 possible ways to reconnect the partial paths into a cycle (Figure D.5). In 4-opt, the situation is even worse with $\mathcal{O}(|\mathcal{V}|^4)$ possible quadruples of edges to break and 20 different ways to reconnect the partial paths.

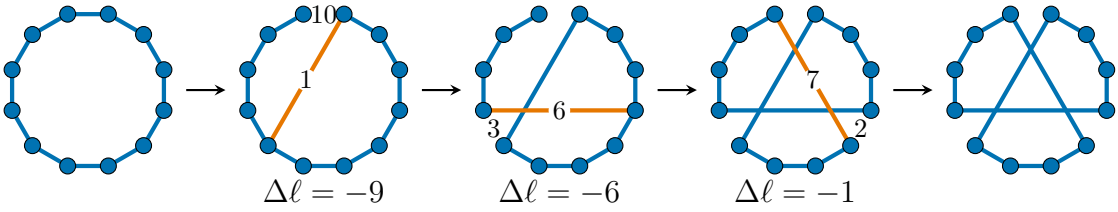


Figure D.6: Sequential exchange of edges in the LK algorithm which each result in a decrease in the total edge length. After each step, the edges do not necessarily a cycle. The edge exchange is only accepted when the resulting edges form a cycle.

Lin and Kernighan [122] proposed a simple way around this search space explosion. When performing k-opt, we swap k pairs of edges (each pair shares a single vertex) and for each pair, we can compute the difference in their lengths. In the original LK algorithm, pairs of edges are swapped sequentially—with adjacent pairs of edges sharing a vertex—to obtain a new cycle. The first and last pair of edges will also always share a vertex and so we can perform the sequence of exchanges starting with any vertex. If the net result of the k exchanges is a shorter tour, there will always be a way to order the exchanges so that the effect of exchanging the first j pairs (for any j between 1 and k) is a set of edges which is shorter than the original cycle. This result is very useful! We can sequentially search for pairs of edges to swap and only consider possibilities which result in a shorter total edge length. When we swap two edges, the resulting set of edges is not usually a cycle, so we keep swapping edges—as long as the net result is a shorter total edge length—until the set of edges is a tour (Figure D.6). This approach effectively lets us find a k-opt move without having to search over all the $\mathcal{O}(|\mathcal{V}|^k)$ possibilities.

I found that the main difficulty in implementing the LK algorithm is guaranteeing that the exchange of edges results in a valid cycle. When I originally read Lin and Kernighan's paper, I was very confused about how to implement the algorithm as there did not seem to be any simple way to check how close we

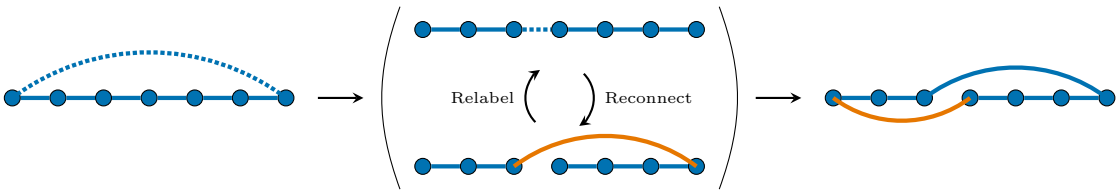


Figure D.7: In Karapetyan's interpretation of the LK heuristic, one edge is removed to form a path. Then, edges are sequentially exchanged by connecting the last vertex of the path to one in the middle. Finally, the first and last vertices are connected if the resulting cycle is shorter than the original one.

are at any given time to having a cycle. Then I found Karapetyan and Gutin's explanation [98] which clarified my understanding of the algorithm and made it much easier to make sure we end up with a cycle. Their insight is to first remove a single edge before swapping pairs of edges (Figure D.7). After removing this edge, the edges form a linear path. Then each additional exchange is equivalent to connecting the last vertex of the path to some vertex in the middle of the path and breaking the edge immediately after that vertex. The result of this exchange is still a linear path, and so the same technique can be applied again and again to perform exchanges of more edges. At any point, this path can be converted back into a cycle by adding a single edge from its start to its end. This move, closing up the cycle, is performed if the resulting cycle is shorter than the original one (i.e. before removing an edge to create the path). The edge removed at the beginning and the edge added at the end can be viewed as the final pair of edges.

Although Karapetyan's explanation of the LK heuristic is very simple, and thus easy to understand and implement, it has a fatal flaw: many of the k-opt moves cannot be built sequentially in this way. For example, the symmetric 3-opt move, which can be obtained by the original LK heuristic, and the double-bridge 4-opt move, which cannot be obtained by the LK heuristic, both aren't possible using Karapetyan's version (Figure D.8). Without these moves possible, the method is not guaranteed to find all beneficial 3-opt or 4-opt moves and might not find the

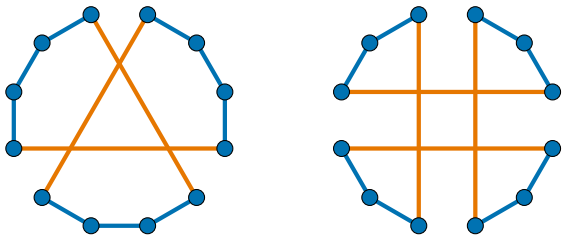


Figure D.8: Two edge exchanges which are difficult to implement using the LK heuristic: the symmetric 3-opt exchange (left) and the double-bridge 4-opt exchange (right). Both moves can be performed by certain variants of the LK heuristic but not others.

locally optimal solution with respect to these moves.

Rearranging the path by connecting the final vertex to some middle vertex is equivalent to performing 2-opt. If we close the paths before and after rearranging once, the two cycles differ by exactly two edges. The paths, on the other hand, differ by a single edge. The additional edges that get swapped when viewing the path as a cycle are the edges that connect to the start of the path. When we rearrange the path for a second time, this edge to the start that we just added gets removed and replaced by a new edge to the start of the path. Therefore by repeatedly rearranging the end of the path, we are effectively performing multiple 2-opts where each time we remove one of the edges that we had previously just added. In other words, we can perform k-opt by repeatedly performing 2-opt!

Is there a maximum number of 2-opts needed to perform any k-opt? Yes! It turns out that at most k 2-opts are needed to perform k-opt [129]. Suppose that all successive 2-opts each have one edge in common. The first 2-opt adds one 2 edges and each 2-opt after that will just add a single edge. After k 2-opts we'll have added $k + 1$ edges, but k-opt only results in k new edges. What's going on here? There must be one edge which got added and later removed. It turns out this "dummy edge" is essential to actually implementing sequential k-opt. Whenever all segments of the path between the exchanged edges get traversed in the same direction in both cycles before and after k-opt, we will need a dummy edge. The simplest case where we need a dummy edge is the symmetric 3-opt,

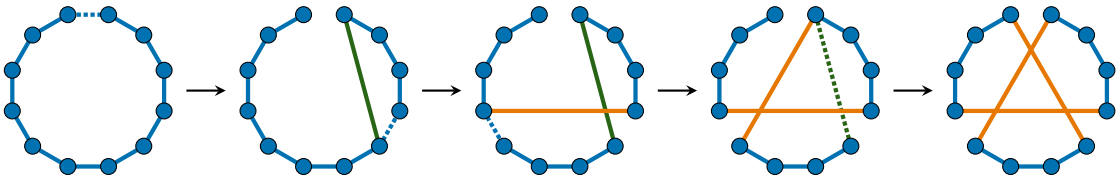


Figure D.9: By using a dummy edge, which will not be part of the final cycle, it is possible to perform the symmetric 3-opt as a sequential exchange. Here each move is simply connecting the last vertex to some middle vertex.

which previously wasn't possible. By adding a dummy edge in the first move, and removing it in the last move, this exchange is now possible (Figure D.9). Notice that all the operations on the path—including the ones involving the dummy edge—just involve connecting the current last vertex of the path to some vertex in the middle of the path.

Even if we use dummy edges to increase the number of possible sequential exchanges, some are still not possible. In particular, the double-bridge exchange (Figure D.8 right) is notoriously difficult to implement in the LK heuristic. The best way that I've found is the modified LK heuristic [128] which slightly relaxes the sequential requirement. Instead of requiring that the next edge added shares a vertex with the last edge removed, it can share a vertex with *any* edge that has been removed. However, each vertex of an edge that has been removed can still only be shared with one edge added. When the first edge is removed, there are two possible vertices that a new edge can share. The first edge added will share one of these vertices and will share a vertex with the next edge removed, resulting in two possible vertices that the next edge can be added next to. For every edge added, there will now always be two vertices of removed edges that haven't been used yet. The new edge must be adjacent to one of these vertices (or both if closing up the cycle). With this change, it is finally possible to implement the double-bridge as a sequential exchange (Figure D.10).

This change is much more intuitive if we use the Karapetyan's interpretation

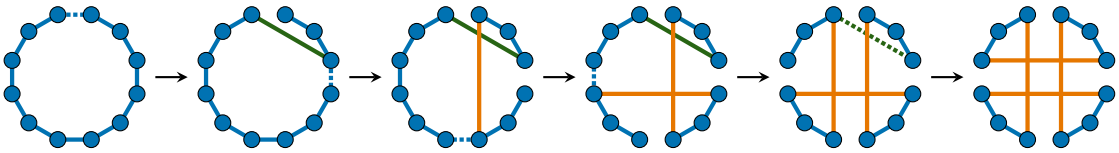


Figure D.10: The double-bridge 4-opt can be implemented as a sequential move by using a dummy vertex and performing some exchanges on the front of the path and some on the back of the path.

[98] where the partially modified cycle is viewed as a path. First, one edge is removed from the cycle to create a path. Then this path is rearranged, either by connecting the last vertex of the path to some vertex in the middle or by connecting the first vertex of the path to some vertex in the middle. These two kinds of rearrangement can alternatively be viewed as reversing the first $j < n$ vertices of path or reversing the last $j < n$ vertices of the path. Up to k successive rearrangements may be performed. Finally, to re-obtain a cycle, the first vertex is connected to the last vertex. Since only one dummy edge is needed to obtain any k-opt, we also add the constraint that we can only remove one previously added edge in a later round. I think this explanation is the simplest, most intuitive way to explain the LK heuristic and it is capable of producing any k-opt move as a sequence of 2-opt moves (including ones described by Lin and Kernighan as "non-sequential").

The LK heuristic can be viewed as a branch and bound algorithm. The initial cycle is the root vertex of a search tree. New branches are added by modifying this cycle according to three modifications:

1. Removing an edge of a cycle to create a path. This modification only happens at the first level of the tree as all lower levels contain paths and not cycles.

2. Rearranging a path by reversing its first or last $j < n$ vertices.

3. Connecting the first and last vertices of a path to create a cycle. This move is forbidden at the second level of the search tree because it would result in

the initial cycle. If, at any lower level, it results in an improved cycle, the search stops.

The key insight of Lin and Kernighan is that this tree can be searched very quickly by bounding most branches. In their formulation, the bounding criterion is that the total length is shorter than that of the initial solution. This criterion doesn't quite work when we include dummy edges, as the dummy edges could be arbitrarily long. Therefore, we use a modified branching criterion if we haven't removed a previously added edge yet. The modified branching criterion is the total length minus the longest edge added plus the shortest edge that could still be added. It may be possible to use a slightly more aggressive bounding criterion which speeds up the search while still being able to find any beneficial exchange. Usually, the solutions are also bounded by limiting the maximum depth of the search tree (i.e. having a maximum value for k).

The number of times that the path gets rearranged is not fixed so the LK algorithm can be implemented using a recursion (Algorithm D.4). The recursive function receives a path, and set of recently added edges and tries to rearrange the path without breaking any of these edges. The set of edges contains all edges that have been added to the path in all rounds of the recursion except the first. The edge added in the first round is allowed to be broken as it can serve as the dummy edge needed to achieve exchanges such as the double bridge. The recursive function will try to break any of the allowed edges. If breaking an edge results in a path which is shorter than the original cycle (the algorithm knows this length as it receives $\Delta\ell$, the difference in length between the original cycle and the current path), the path is rearranged by breaking this edge and the rearranged path is used as the input to the next level of recursion. If at any time the rearranged path can be closed back into a cycle which is shorter than the original, this path will be returned all the way through the chain of recursive function calls and will therefore be returned where the recursion was called for the first time. If no better cycle

can be found (up to a maximum recursion depth) the original path is returned to the previous level. At this point, the algorithm does not travel up to the initial function call immediately, but instead tries every rearrangement at the previous level before returning the original path provided to the level of recursion before that. Ultimately, if there are no improvements to the original cycle, the original path will eventually be returned wherever the recursion was first called. Note that the criterion used here that $\Delta\ell + \Delta\ell_{\text{back}} < 0$ is the criterion used in the original version of the LK heuristic and does not account for the dummy edge.

Algorithm D.4: Lin–Kernighan recursion

Input: Spanning path, p; cumulative change in length, $\Delta\ell$; set of new edges, $\mathcal{E}$; `recursion_depth`; and `max_recursion_depth`
Output: Improved spanning path, p_{new}

```
if recursion_depth ≥ max_recursion_depth then
    return p
for edge e of p not in 𝓔 do
    Δℓ_back ← change in length from reversing partial path after e
    if Δℓ + Δℓ_back < 0 then
        if recursion_depth ≠ 1 then
            e_new ← edge connecting end of p to vertex of e
            𝓔_new ← 𝓔 ∪ {e_new}
        p_new ← path obtained by reversing partial path after e
        Δℓ_close ← change in length by connecting ends of p_new
        if Δℓ + Δℓ_back + Δℓ_close < 0 then
            return p_new
        else
            p_new ← Lin–Kernighan recursion (p_new, Δℓ + Δℓ_back, 𝓔_new,
                    recursion_depth + 1, max_recursion_depth)
            Δℓ_close ← change in length by connecting ends of p_new
            if Δℓ + Δℓ_back + Δℓ_close < 0 then
                return p_new
    repeat the same procedure except reversing the partial path before e
return p
```

This recursive implementation of a single LK move can then be used to implement the entire algorithm (Algorithm D.5). Starting with an initial random cycle, the cycle is rearranged by opening it up to a path by removing an edge. This path

is then rearranged by calling the recursion which will either return the original path or one that can be closed into a shorter cycle. If the recursion returns a different path than it was given, this new path must be better and so it is closed up into a cycle and the process repeats. If the recursion returns the same path it was given, the algorithm attempts to rearrange the cycle by breaking a different edge. Once all of the edges of a cycle have been tried without any rearrangements resulting in an improvement, the current cycle is returned as it is a local minimum with respect to the LK transformation.

Algorithm D.5: Lin–Kernighan algorithm

Input: Complete weighted graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w)$
Output: Spanning path, c

```
1  c ← random spanning cycle on G
2  improved ← true
3  while improved do
4      improved ← false
5      for edge e of c do
6          p ← path obtained by removing e from c
7          p′ ← improved path by rearranging p        /* Algorithm D.4 */
8          if p′ ≠ p then
9              c ← cycle obtained by connecting ends of p′
10             improved ← true
11             break for
12 return c
```