Any-Com Multi-Robot Path Planning

by

Michael Wilson Otte

B.S. Computer Science, Clarkson University, 2005

B.S. Aeronautical Engineering, Clarkson University, 2005

M.S. Computer Science, University of Colorado at Boulder, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2011

This thesis entitled:
Any-Com Multi-Robot Path Planning
written by Michael Wilson Otte
has been approved for the Department of Computer Science

Prof. Nikolaus Correll

Prof. Michael Mozer

Prof. Richard Han

Prof. Jason Marden

Prof. Eric Frew

Prof. Gaurav Sukhatme

Prof. Richard Voyles

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Otte, Michael Wilson (Ph.D., Computer Science)

Any-Com Multi-Robot Path Planning

Thesis directed by Prof. Nikolaus Correll

Robust autonomous robotic systems must use complete or probabilistically complete path planning algorithms when less expensive methods fail (where *completeness* is the algorithmic property of being able to find a solution to a problem when one exists). However, these methods are sometimes so computationally complex that a solution cannot be found within a reasonable amount of time. Communication between robots tends to increase completeness and reduce computational complexity; however, communication quality is environmentally dependent and often beyond control of the user or system. Previous approaches to the multi-robot path planning problem have each been tailored to a single point within the completeness vs. computational vs. communication state space, and are often ill-equipped to solve problems outside their design envelope. In contrast, I believe that truly robust multi-robot navigation can only be achieved by algorithms that automatically tune their performance within this state space to maximize performance vs. each problem the system faces.

My personal bias is to maximize algorithmic completeness while respecting the computational resource and communication quality that is currently available. In order to be useful, the resulting solutions must be calculated within a reasonable amount of time. I also believe that it makes sense to divide the computational effort of finding multi-robot path planning solutions among all robots that the solution will benefit. This can be accomplished by recasting a networked team of robots as an ad-hoc distributed computer—allowing the team's computational resources to be pooled increases the complexity of problems that can be solved within a particular amount of time. However, distributed computation in an ad-hoc framework must respect the fact that communication between computational nodes (i.e., robots) is usually unreliable.

I propose the thesis "*Sharing Any-Time search progress over an ad-hoc distributed computer*

that is created from a dynamic team of robots enables probabilistically complete, centralized, multi-robot path-planning across a broad class of instances with varied complexity, communication quality, and computational resources." The work presented in this dissertation in support of my thesis can be divided into three related areas of focus. (1) I propose a new distributed planning concept called Coupled Forests Of Random Engrafting Search Trees (C-FOREST), and demonstrate that it has parallelization efficiency greater than 1 for many problems. (2) I propose using a robotic team as an ad-hoc distributed computing cluster, and demonstrate that when C-FOREST is run on this type of architecture it is able to exploit perfect communication when it exists, but also has graceful performance declines as communication quality deteriorates. I coin the term "Any-Com" to describe algorithms with the latter property. (3) I propose a dynamic team version of Any-Com C-FOREST that allows multiple robotic teams to form and then re-form as robots move about the environment. Each team acts as an ad-hoc distributed computer to solve its composite robots' communal path planning problem. Limiting teams to include only conflicting robots improves algorithmic performance because it significantly reduces the computational complexity of the problem that each team must solve. Replanning through *only* the subset of the configuration space in which conflicts occurs has similar computational benefits.

Dedication

This thesis is dedicated to my mechanical pets—on account of their integral role in its creation:

Scooter (#1)

Theodore "Teddy" (#2)

Bill (#3)

Jeeves (#4)

Johny (#5)

Geoffery (#6)

As well as my flesh and blood pets:

Luna

Jewels

Stormey

And old friends:

Roger (#22)

Marven (#42)

Acknowledgements

I would like to thank the residents of Andrew's Hall for allowing me to conduce experiments in their home. I would also like to thank Sam Edwards for helping map Andrew's Hall, Erik Komendera for helping run the Large Andrew's Hall Experiment, my advisor Nikolaus Correll and all of the other members of my thesis committee for the valuable feedback and advice that they have provided, as well as Dustin Reishus for his feedback. Finally, I would like to thank the members of the Andrews Hall Robotics Initiative who helped me instrument the building with infrared localization tags.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

The proliferation of inexpensive computers, high-resolution depth-sensing devices, and accurate localization systems has significantly shifted the economics of using robotics. It has both increased the number of applications where robotic solutions are affordable and reduced the barrier-to-entry of the robotic research field. Thus, the demand for mainstream robotic products is growing in parallel with the wealth of expertise and knowledgeable that is necessary to deliver them. This suggests a significant increase in robotic deployment in the near future. Autonomous navigation is a key capability for enabling both industrial and consumer robotics to perform their work effectively [31]. As robot traffic becomes more congested, tomorrow's systems must be capable of coordinated interaction within a multi-robot society. This imposes a need for multi-robot navigation solutions that can plan efficient, coordinated, and collision-free paths for all robots operating within a common environment.

The *multi-robot navigation problem* is to find a coordinated set of collision-free paths for all robots moving within a common area. It is an instance of the piano mover's problem—for which *complete* solutions are exponentially difficult to calculate in the number of robots involved [113]. For instance, *centralized* multi-robot path-planning algorithms view individual robots as separate pieces of a single conglomerate robot. These algorithms provide the best completeness guarantees of any tool in the multi-robot navigation toolbox; however, they are also the most expensive to use—sometimes prohibitively so. Many less-expensive *incomplete* methods have proven to be extremely useful for all but the most challenging navigation instances (see Chapter 2). However, challenging

problem instances do exist, and robust autonomous systems must use complete algorithms when less expensive methods fail. I believe that multi-robot navigation algorithms should automatically adjust their level of completeness to maintain tractability vs. the particular problem they are currently facing.

Communication between robots tends to increase completeness and reduce computational complexity. For example, teams without communication may fail to solve simple bottleneck problems, and it is much easier to learn about other robots' intentions via messages than to infer them from sensor observations and/or behavior models. Unfortunately, communication quality is environmentally dependent and often beyond control of the user or system. I believe that multi-robot navigation algorithms should adjust to use whatever communication is available, while also attempting to maximize completeness and maintain tractability .

Previous approaches to the multi-robot path planning problem have each been tailored to a single point within the completeness vs. computational vs. communication state space, and are often ill-equipped to solve problems outside their design envelope. In contrast, I believe that truly robust multi-robot navigation can only be achieved by algorithms that automatically tune their performance within this state space to maximize performance vs. each problem the system faces.

I believe that if each robot is equipped with its own computer and the ability to communicate, then it makes sense to divide computational effort of finding a centralized path planning solution among all robots that the solution will benefit. In particular, a networked team of robots can be re-cast as a distributed computer to solve the path planning problem encountered by its composite robots. While the focus of this dissertation is on path planning, I believe that the general idea of ad-hoc distributed computing should generalize well to other multi-robot problems. Solving communal problems in this manner is, arguably, elegant on a philosophical level because it enables the composite robots to become a single entity in both body and mind.

Any distributed computation over a robotic team must respect the fact that wireless bandwidth is environment dependent, and often beyond the control of the user or system. Current algorithms for coordinating networked robot systems usually rely on a minimum quality of service

and fail otherwise. I am interested in distributed algorithms that maximize the utility of unreliable communication channels, but also take full advantage of high-quality networks. I use the term "Any-Com" to describe this type of algorithm. Any-Com algorithms should exploit perfect communication and have gracefully performance declines otherwise.

I propose the thesis "*Sharing Any-Time search progress over an ad-hoc distributed computer that is created from a dynamic team of robots enables probabilistically complete, centralized, multi-robot path-planning across a broad class of instances with varied complexity, communication quality, and computational resources.*" The central idea explored in my research is to have a robotic team find a sub-optimal solution as quickly as possible, then refine that solution subject to both communication and time constraints. The idea is inspired by previous work on Any-Time algorithms by [15] and [43]. However, the use of distributed computation over an unreliable dynamic network presents unique challenges that require new solutions. In general, the work presented in this dissertation can be divided into three related and progressing areas of focus.

(1) I propose a new distributed planning concept called Coupled Forests Of Random Engrafting Search Trees (C-FOREST). Although C-FOREST was designed with the multi-robot path planning problem and ad-hoc distributed computing in mind, it has proven remarkably useful for a variety of path planning applications (e.g., manipulator arm problems) and distributed computing architectures. Indeed, parallelization efficiencies significantly greater than 1 (e.g., 20) have been demonstrated using a traditional distributed computing cluster.

(2) I explore the concept of using a robotic team as an ad-hoc distributed computing cluster, and demonstrate that when C-FOREST is run on this type of architecture it is able to exploit perfect communication when it exists, but also has graceful performance declines as communication quality deteriorates. I coin the term "Any-Com" to describe algorithms with the latter property, and call the resulting algorithm Any-Com C-FOREST.

(3) I extend Any-Com C-FOREST to accommodate dynamic teams of robots, where teams are formed and re-formed to accommodate path conflicts as robots move about the environment. Each team acts as an ad-hoc distributed computer to solve the communal path planning problem of its

composite robots. Limiting teams to include only conflicting robots improves performance because it significantly reduces the computational complexity of the problems that each team must face. Replanning through only the subset of the configuration space in which conflicts occur has similar computational benefits. The resulting algorithm is called Dynamic-Team Any-Com C-FOREST.

In the current version of this work I assume that some mechanism exists *a priori* with the ability to propagate information throughout whatever distributed architecture is being used (e.g., a message passing protocol). I also assume that the time it takes to propagate information from one CPU to all others is small compared to the time required to solve the centralized multi-robot path planning problem. I believe that in (1) it is a fair assumption for the majority of standard distributed architectures that are used in practice; however it may not apply in extreme cases. In the case of the ad-hoc distributed computing clusters that I investigate in (2) and (3), teams are encouraged to form in ways that are conducive to communication, and I experimentally investigate the effect that dropped messages have on the algorithm (e.g., due to data corruption and/or sending conflicts). That said, I do not explicitly investigate the effects that different underlying message passing protocols have on the algorithms' performance.

The contributions of the work presented in this dissertation can be summarized as follows:

(1) Proposal of a new method for distributed single-query path planning called C-FOREST.

- Theoretical proof that C-FOREST can have super linear speedup vs. number of CPUs.
- Experimental validation that C-FOREST can have super linear speedup in practice.
- Proposal of a modified Sequential C-FOREST for use on a single CPU.
- Experimental validation that Sequential C-FOREST is beneficial on a single CPU.
- Experimental validation that C-FOREST and Sequential C-FOREST can be used with a variety of distance metrics, configuration spaces, robots, and random search-trees.

(2) Proposal of the Any-Com C-FOREST algorithm for use on an ad-hoc distributed computer.

- Coinage of the term "Any-Com" to describe distributed and/or multi-agent algorithms that have graceful performance declines vs. decreasing communication quality.
- Demonstration that a team of robots can be used as an ad-hoc distributed computer.
- Experimental validation that Any-Com C-FOREST performs well in practice, even when communication is relatively poor.

(3) Proposal of Dynamic-Team Any-Com C-FOREST algorithm, in which robots form teams and re-plan in sub-regions of the configuration space as necessary to avoid collisions.

- Observation that increasing team size can cause practical planning failure due to increased problem complexity.
- Theoretical insight into how the size of the configuration space can be reduced while maintaining probabilistic completeness.
- Experimental validation that Dynamic-Team Any-Com C-FOREST allows more difficult planning instances to be solved than Any-Com C-FOREST, with respect to planning time.
- Experimental validation that both team size and configuration space diameter independently contribute to problem complexity.

The rest of this dissertation is organized as follows: The second half of this chapter contains a high level overview of the work presented in the rest of the thesis (Section 1.1), as well as a nomenclature reference (Section 1.2). Related work is presented in Chapter 2. The meat of the dissertation, in which I defend my thesis, is presented in Chapters 3 through 5, where each chapter includes the technical details, experiments, and discussion pertaining to one of the three major focuses mentioned above. Chapter 3 is on the C-FOREST distributed path planning algorithm. Chapter 4 explores the concept of using a robotic team as an ad-hoc distributed computer, and the Any-Com properties exhibited when C-FOREST is used on this type of architecture. The dynamic team extension to Any-Com C-FOREST is presented in Chapter 5. Final thoughts and conclusions

are presented in Chapter 6.

There are also three appendices containing additional content that I consider relevant, but that is noncritical to the defense of my thesis. Appendix A includes a brief introduction to path planning concepts that is intended to provide enough background information to make the rest of the dissertation accessible to non-experts. Appendix B describes the Shortest-Path Random Tree algorithm—one of the underlying random-tree algorithms that is tested with C-FOREST, and the particular random tree algorithm used in the Any-Com C-FOREST and Dynamic Team C-FOREST experiments. Appendix C contains preliminary work I have done on the expected lengths of greedy paths through random graphs. This theoretical work has guided the parameter selection used for the underlying random trees in C-FOREST.

1.1 Overview

I believe there are two different ways to motivate the multi-robot path planning algorithms presented in this dissertation. One is top-down argument, while the other uses a bottom-up line of reasoning. Both ways of thinking have played an important role in shaping how I personally approach the multi-robot problem, and therefore provide two different lenses through which my work can be viewed. While neither is inherently better than the other, it is likely that certain readers will prefer one over the other. I personally believe that the full value of my work can only be understood by considering both, and I now summarize each in turn.

1.1.1 Top-down motivation

A vast number of approaches to solve the multi-robot planning have previously been proposed (see Chapter 2). One reason that so many ideas have been suggested is that none is capable of solving every multi-robot planning problem that exists. The trade-off between algorithmic completeness and computational complexity is painfully present in the multi-robot path planning domain. Fast algorithms tend to work well in most cases, but they are unable to deduce when a solution does not exist—and occasionally leave entire robot populations crippled in dead-lock traffic-jams even when

a solution *does* exist. On the other hand, more powerful algorithms exist that do not suffer from the latter two problems; however, they are so computationally complex that calculating a solution may be impossible within a practical amount of time—allowing entire robot populations to rust into dirt before a movement plan can even be devised.

While these two caricatures represent opposite ends of the completeness vs. computation spectrum, there are a multitude of other algorithms located somewhere in the middle that attempt to find an 'optimal' trade-off between the two (where optimality is usually defined by the person that came up with a particular algorithm—hopefully due to the constraints imposed by whatever application domain they were working in at the time).

I believe that truly robust multi-robot planning algorithms must be able to automatically adjust their operation along the completeness vs. computation spectrum based on the requirements of the current problem they are faced with. This will allow inexpensive incomplete methods to be used when it is advantageous to do so, and more expensive algorithms to be used when they are necessary and/or tractable.

In real world environments there is another axes to the algorithmic state-space that I have been considering—namely, communication. Another reason that so many different multi-robot planning algorithms exist is that it is also possible to optimize different algorithms for different levels of robot-robot or robot-environment communication. There are obvious trade-offs between computation and communication (it is easier for robots to exchange information about each-other's locations and intentions over a wireless network then to deduce these things from environmental sensors and/or abstract behavioral models). Likewise, there are trade-offs between communication and completeness (robots without communication may not even solve a simple bottleneck problem).

Multi-robot planning algorithms will be more useful if they can automatically adjust their performance within the completeness vs. computation vs. communication space to handle whatever situations arise. The work presented in this dissertation is a step in that direction, and to the best of my knowledge it is the first multi-robot planning algorithm that attempts to automatically tune itself vs. computation, completeness, and communication. In general, the algorithms I present

attempt to find actionable solutions within a reasonable amount of time while doing their best to maximize completeness given the available computational resources and communication quality.

Given these goals, it makes sense to distribute the computational load of solving the problem. On the incomplete side of the spectrum this has traditionally been done passively (e.g., by having desirable distributed behavior emerge as a function of each robot following its own agenda). While it is usually done actively on the complete side of the spectrum (e.g., by having robots explicitly share data and agree on a mutual plan). Since sharing information raises the upper limits of completeness without inherently increasing computation (at least, substantially), it seems desirable that planning algorithms maximize the utility of whatever communication is available.

Assuming that some communication is available, it makes sense to design algorithms that enable all robots to contribute their computational resources to solving a mutual problem. I believe that this can be accomplish by creating an ad-hoc distributed computer from the networked robotic team, and then designing planning algorithms that can take advantage of an ad-hoc distributed computing framework. This idea is investigated in Chapter 4.

It is well known that problem complexity is dependent on both the size of the robotic team, and also on the size of the environment in which the team must operate. Therefore, I believe that it (1) makes sense to keep teams as small as possible, and (2) have teams plan in the smallest portion of the environment necessary to find a solution. Both (1) and (2) should be done while simultaneously attempting to maintain algorithmic completeness. Chapter 5 is devoted to exploring this concept using a dynamic-team approach. All robots start in their own team, and teams are combined (only) when doing so is necessary to guarantee safety. The combined team plans through the smallest subset of the environment necessary to safely avoid the conflict. I call the subset of the environment used for team planning the *conflict region*. All robots can use their original plans to move to and from the conflict region, while navigational coordination within the conflict region is given by the combined solution.

1.1.2 Bottom-up motivation

Centralized multi-robot path planning is the most complete and computationally expensive type of multi-robot path planning algorithm. In centralized planning all robots are viewed as component pieces of a single robot. Given infinite computational and communication resources, centralized algorithms would always be used—since they provide the best solutions available. However, practical use is limited by finite computation time and communication resources. Increasing speed and decreasing dependence on communication will increase the number of problems for which centralized algorithms can be used. Further, the same algorithms are also used to solve other high-dimensional path-planning problems—in fact, one can argue a strong case that they are actually used more frequently to solve single-robot manipulator-arm problems than multi-robot problems. Therefore, increasing the speed of centralized algorithms will benefit many other high-dimensional planning problems beyond multi-robot navigation.

I believe that one of the best ways the speed of centralized algorithms can be increased is by harnessing the power of distributed computation. In Chapter 3 I present a new high-dimensional path planning framework called C-FOREST that uses distributed computation. C-FOREST has proven to be both very powerful and very general. I prove that it has super linear speedup—that is, with n CPUs it is possible to find a similar solution in *less than* $1/n$ the time required by a single CPU. In fact, I observe speedups an order of magnitude higher than any previously observed in the robotic path planning domain. Further, C-FOREST can be extended to use any random tree algorithm that has optimal convergence, and any planning space that obeys the triangle inequality.

As discussed in the previous section, it is possible to view a networked robotic team as an ad-hoc distributed computer. Given that powerful distributed centralized path-planning algorithms exists, it seems natural to employ them on an ad-hoc distributed computer to solve the multi-robot path planning problem faced by its composite robots (nodes). Chapter 4 focuses on the modifications that are necessary to adapt C-FOREST to an architecture that has unreliable communication between its computational nodes.

Finally, forming an ad-hoc distributed computer only makes sense when doing so is necessary to avoid collisions. If non-overlapping sub-problems exist, then multiple ad-hoc distributed computers should be used to minimize computational complexity faced by each team. As previously discussed, it is also important to have each computer select an appropriately sized chunk of the environment to plan through. These ideas are addressed in Chapter 5.

1.2 Nomenclature

The purpose of this section is to provide a reference for the many variables used in Chapters 3-6. All variables presented here are also described as they are introduced later in the dissertation. I anticipate that the casual reader will skip this section on an initial read-through—but then use is as an occasional reference to lookup variable definitions.

Let $time_1$ and $time_T$ be the computation time required to solve a particular problem with 1 or T CPUs, respectively. Speedup is defined as $S = time_1/time_T$, and measures the benefit derived from using T CPUs in parallel. Parallelization efficiency is defined as $E = S/T$ and measures the amount of speedup per CPU. Efficiency is inversely proportional to the power that must be consumed to solve a problem.

A tree $\mathbf{t}$ is a directed, acyclic, connected graph consisting of nodes (way-points) and edges (path segments). It is standard practice to 'grow' a tree from a root node out toward leaf nodes such that edges are always directed away from the root and toward the leaves. Each node in the tree has exactly one incoming edge—except for the root, which has no incoming edges. By construction it is possible to reach the root node from any other node in the tree by iteratively moving counter to edge direction.

A forest $\mathbf{T}$ is a set of $T = |\mathbf{T}|$ trees ($\mathbf{t} \in \mathbf{T}$). A forest containing only one tree (i.e, a stand-alone tree) is denoted $\mathbf{t}'$. The prefix '$\mathbf{t}$.' (note the dot) is used to indicate association with the particular tree $\mathbf{t}$. A particular state (or point) in the configuration space is denoted v. Let s and g represent the start and the goal states, respectively. A solution (or path) $\mathbf{P} \subseteq \mathbf{t}$ is composed of a sequence of states beginning at s and ending at g. By construction it is safe for the system to

move from the i-th state to the $(i+1)$-th state of a particular solution. The current best solution is denoted $\mathbf{P}_{bst}$ and contains $|\mathbf{P}_{bst}|$ points labeled $\mathbf{P}_{bst,i}$ for $1 \leq i \leq |\mathbf{P}_{bst}|$. I assume that a distance metric $\|\cdot\|$ is defined over the configuration space, and that it obeys the triangle inequality. L_{bst} is the measure of $\mathbf{P}_{bst}$ with respect to this metric, $L_{bst} = \|\mathbf{P}_{bst}\|$. For example, L_{bst} may represent the distance traveled by a robot following $\mathbf{P}_{bst}$.

$h(v_1, v_2)$ is admissible heuristic function that returns an (under) estimate of the distance between two states v_1 and v_2 (in the case of multiple goal states $h(v_1, g)$ is assumed to be admissible over the entire set of goals). $h_s(v) = h(s, v)$ and $h_g(v) = h(v, g)$. The actual distance from the start to v through tree $\mathbf{t}$ is $\mathbf{t}.d_s(v)$. The tree that discovered $\mathbf{P}_{bst}$ is $\mathbf{t}_{bst}$ and $L = \mathbf{t}_{bst}.d_s(g)$. Geometrically, $h_s(v) + h_g(v) = L$ describes an enclosed region $\mathbf{A}_L$ in the search space, in Euclidean space $\mathbf{A}_L$ is bounded by an ellipsoid.

The Lebesgue measure of a space is denoted $\|\cdot\|_l$. If the space contains 1,2, or 3, dimensions then $\|\cdot\|_l$ corresponds to the length, area, and volume of that space, in higher dimensions $\|\cdot\|_l$ measures the hyper-volume of that space. The workspace and configuration spaces are denoted $\mathbf{W}$ and $\mathbf{C}$, respectively. When different numbers of robots are used, a configuration space containing R robots is denoted $\mathbf{C}_R$.

When message passing is explicitly used r is used to denote the particular robot with ID r, and $\mathbf{t}.r$ holds the id of the robot that is building tree $\mathbf{t}$. The set of all robots is denoted $\mathbf{R}$, and the number of robots is $R = |\mathbf{R}|$. The portions of the combined start and goal vector associated with robot r are denoted s_r and g_r, respectively. All robots keep track of what they believe to be the current state of the team. The following data is defined with respect to the current belief of the robot on which it is located (host robot). $\mathbf{D}$ is a set of data fields, where each data field $d_r \in \mathbf{D}$ contains data about a particular robot r that the host robot knows about, including its start (or current location) $d_r.s$ and goal $d_r.g$. If the host robot does not know about a particular robot r then $d_r \notin \mathbf{D}$. the ID of the agent that created $\mathbf{P}_{bst}$ is r_{bst}. The list $\mathbf{V}$ contains robots that the host robot believes also support $\mathbf{P}_{bst}$ as the best solution. The list $\mathbf{F}$ contains all robots, known to the host robot, that have submitted a final solution (i.e., have finished planning). $\mathbf{m}.M$ is a movement

flag that is true if the host robot has started moving. B is the amount of time that the host robot is behind schedule, with respect to movement along the solution.

When dynamic teams are used, a particular dynamic team is denoted $\mathbf{\Delta}$, and contains $\Delta = |\mathbf{\Delta}|$ robots. Also, $d_r.\mathbf{P}_{nav}$ contains the entire path between the current location $d_r.s$ and the goal $d_r.g$ that robot r is currently using to navigate—including its time parameterization, and $d_r.\varepsilon$ contains the current planning epoch of robot r.

A message is denoted $\mathbf{m}$ and its subfields are denoted by the prefix '$\mathbf{m}$.'. The subfields reflect the sending robot's current belief about the state of its team and are analogous to the values described in the previous paragraph. They include $\mathbf{m}.\mathbf{D}$, $\mathbf{m}.\mathbf{P}_{bst}$, $\mathbf{m}.L_{bst}$, $\mathbf{m}.r_{bst}$, $\mathbf{m}.\mathbf{P}_{bst}$, $\mathbf{m}.\mathbf{V}$, $\mathbf{m}.\mathbf{F}$, $\mathbf{m}.M$ and $\mathbf{m}.B$. If dynamic teams are being used then $\mathbf{m}.\mathbf{\Delta}$ is also included.

The rate at which messages are sent is $1/\omega$ (i.e., the amount of time between each outgoing message is ω). The amount of planning time allotted during the planning phase is μ. The probability that a message is successfully transmitted is denoted τ.

Chapter 2

Related work

The major focus of this dissertation is on the distributed implementation of centralized single-query multi-robot path planning algorithms. However, the C-FOREST algorithm that provides the path planning foundation of my work can also be used for other single-query high dimensional path planing problems that are neither multi-robot nor Any-Com in nature (e.g., manipulator arms). This chapter is broken into two major sections in order to address the diverse body of related work. The first is concerned with multi-robot path planning in general, and the second surveys single query path planning algorithms with a bias toward those that use distributed computation and/or some notion of a forest.

2.1 Multi-robot path planning

Here I briefly discuss a few multi-robot algorithms located along the communication, computation, and completeness spectra. Recall that a *complete* algorithm is guaranteed to find a solution when one exists and will also report failure in finite time if a solution does not exist. A *resolution complete* algorithm is an algorithm that is complete to within a predefined granularity of the world representation. A *probabilistically complete* algorithm is an algorithm that will find a solution, if one exists, in finite time with probability approaching 1.

2.1.1 Cocktail party

At the low end of the communication and completeness spectra lie reactive algorithms, in which all robots are relatively ignorant of other agents' intentions. In the multi robot context 'reactive' means that robots react to the movements of each-other by planning new paths (as opposed to single-robot 'reactive' algorithms that may not use paths at all). This idea is often called the *cocktail party* model because it resembles the navigation method used by guests at a cocktail party [84]. Each agent maintains its own world-view, goals, and navigation function. Other robots are viewed as obstacles. Each agent alternates between sensing, planning, and movement. The control loop is assumed to run fast enough to prevent collisions. There is no direct coordination between robots, but nothing prevents them from using passive sensors to detect each other. When robots use the same algorithm (and can be differentiated them from other moving bodies) the estimated movement of other robots can be refined [123, 128]. Cocktail party algorithms are incomplete (e.g. they can fail when two robots must move in opposite directions through a narrow corridor), but they are popular due to their simplicity, scalability, and communication free architecture.

2.1.2 Traffic rules

Often a strict set of *traffic rules* is used to facilitate multi-robot navigation similar to the way automobiles (theoretically) interact via traffic laws [1, 2, 65, 106, 107]. These methods assume each robot knows the rules, agrees to follow them, and can sense required environmental cues (e.g. stoplights). A robot is allowed to use any planning method that respects the rules, so world knowledge can often be restricted to a local subset of the environment. Traffic rules are relatively simple, distributed, and scalable. However, they assume highly structured environments, and may contain rules that prohibit optimal solutions from being found (e.g. taking a short-cut by going the wrong way down a one-way street). Most research in this area is focused on designing the rules and/or the environment to guarantee minimum performance. In a game theoretic context, this problem is known as game design [85].

2.1.3 Prioritized planning

Prioritized planning forces robots to respect the movement constraints imposed by higher priority robots [22, 37, 40, 126]. The highest priority robot plans first, then the next-highest, and so forth. Priorities may be assigned *a priori* or on-line via a bidding mechanism or other process. Robots can use whatever underlying path-planning method they want, but it is assumed they can communicate an environmental space-time reservation to lower priority robots. On-line versions exist that alternate sensing, planning, and movement [26, 27, 49, 50, 129]. Prioritized methods are greedy and inherently incomplete. Higher priority robots follow optimal to near-optimal trajectories while lower priority robots may be unable to find a solution. Prioritized planning has also been used to periodically create a line-of-sight communication chain while performing the somewhat related coverage task [53].

2.1.4 Decoupled planning

Decoupled planning works in two phases [6, 48, 62, 63, 79, 117]. In phase-1, each robot calculates its own path to the goal. In phase-2, the space-time positions of the robots along these paths are calculated such that no collisions occur. Usually either a descendant of A* [52] and/or PRM [95] is used as the underlying search algorithm. Priorities may be assigned for the phase-2 calculation [8, 13], and special cases for two robots exist [24, 77, 89, 116]. Phase-1 is completely distributable, but phase-2 must be performed on a single agent (or in parallel on each robot)—and communication with that agent must exist. Although decoupled planning can be distance-optimal, it is incomplete because each robot's path is completely determined after phase-1 (and may pathologically conflict) [109]. That said, decoupled planning is arguably more complete than cocktail party, traffic rule, or prioritized methods.

2.1.5 Centralized planning

In *Centralized planning* all robots are considered to be individual pieces of a single composite robot. Paths are calculated in the resulting high dimensional composite configuration space [16,

17, 28, 29, 38, 39, 97, 102, 108, 109, 113, 115, 127]. The high dimensional solution is then projected down into the relevant subspaces for each robot. In previous research, the path has been calculated on a single agent or at the same time on each robot independently. It is assumed robots can communicate with this agent or each other, respectively. Centralized planning is theoretically optimal and complete, but practical algorithms are usually probabilistically or resolution complete and optimal [55, 102]. Regardless, centralized planning provides the best solution quality of any multi-robot planning method, but is also the most computationally expensive.

2.1.6 Dynamic teams

A *dynamic team* is a temporary confederation of robots formed as a result of environmental or other factors [28]. Dynamic teams are often used to solve robotic problems that require (or can benefit from) using multiple coordinating agents. For example, coverage (e.g., for search-and-rescue) [7, 124], surveillance [56], remote sensing [3], exploration [125], and maintaining a communication link [35, 36].

The most closely related previous work on using dynamic teams for path planning is [28], in which teams are encouraged to form as soon as robots are within communication range. In contrast, I investigate using dynamic teams to solve non-overlapping sub-problems—such that robots with conflicting solutions are placed in the same team in order to find a non-conflicting solution, but non-conflicting robots/teams remain separate.

2.1.7 Any-Com

I coin the term "Any-Com" (along with my advisor, Nikolaus Correll) in [90, 91, 92], to refer to the class of distributed algorithms that are able to maximize the utility of perfect communication and have graceful performance declines as communication deteriorates. However, many other Any-Com algorithms exits. For instance, in [105] and [5] Any-Com properties are observed in coverage algorithms, and in [54] algorithms with Any-Com properties are used for search.

One reason that the Any-Com idea was not previously labeled is that most practical *net-*

working algorithms (e.g., algorithms for data transmission) must have Any-Com properties in order to be useful [61]. While these are undoubtedly important (indeed, one could argue that any other type of Any-Com algorithm must rely on at least one Any-Com communication algorithm as a subroutine), listing all of them would require hundreds of citations. As a result, it may not make sense to use the term "Any-Com" in the networking domain; however, I believe that the term is useful for distinguishing algorithms in other domains that are capable of utilizing unreliable and/or shifting network quality for explicit distributed computation or coordinated multi-agent tasks.

2.2 Single-query path planning

While efficient grid-based methods exist for 2- and 3-dimensional problems [34, 44, 52, 68, 118], the PSPACE-hardness of complete planning causes complete algorithms to be impractical in higher dimensions [55, 103]. State-of-the-art higher-dimensional algorithms randomly sample the environment to create a graph that is then searched using standard graph techniques. In general, these algorithms are probabilistically complete. Unlike low-dimensional grid-based methods that spend most of their time searching the graph, randomized search algorithms use the bulk of their computational effort building the graph itself. Depending on the intended application, high-dimensional sampling based planners tend to come in one of two flavors: multi-query and single-query.

2.2.1 Single-query vs. multi-query algorithms

Multi-query planners are used when many searches are expected to be performed in the same environment. A detailed graph through the configuration space is created, stored, and possibly improved over time. Paths are calculated by connecting start and goal states to the graph and then searching for a path between them [28, 69, 95, 108, 109].

Single-query planners are used when a different environment is encountered every time a system plans [121]. A detailed graph is not saved, since each graph is only used once, so the planner builds the best graph possible within the allotted planning time. Single-query planners

usually take the form of random tree algorithms that fuse graph creation and search. Newly sampled random points are immediately inserted into the tree if they can be connected to the existing graph. Points that cannot be connected to the current graph are forgotten. C-FOREST builds directly on single-query random trees.

One of the earliest and most widely used single-query planners is the *Rapidly Exploring Random Tree* or *RRT* [75, 76]. Re-planning versions also exist [41, 45]. While RRT provides probabilistic coverage guarantees, the resulting paths tend to wander—for instance, [64] prove that RRT will almost surly converge to a sub-optimal solution. In contrast, *RRT** attaches new nodes in a way that minimizes cost with respect to a carefully choose subset of old nodes such that the resulting algorithm almost surly converges to the optimal solution [64]. The tree in [91] is similar, except that the set of potential neighbors always includes all nodes, and random remodeling guarantees optimal convergence.

2.2.2 Distributed single-query algorithms

This dissertation pulls a significant amount of material from my previous papers on Any-Com [90, 91, 92] and C-FOREST [93]. [90] is a preliminary look at the Any-Com concept, and also investigates using Any-Com for coverage problems. [91] outlines a multi-robot search algorithm that is solved in a distributed manner by a six robot team. [92] extends this work to include dynamic teams. The algorithm used in both [91] and [92] is similar to C-FOREST presented in [93], except that the underlying trees are only assumed to be SPRT trees (SPRT trees are described in Appendix B). Although super-linear speedup was observed in [91], there was no theoretical explanation for why it existed. Proving that the super linear speedup is algorithmically based is a main contribution of [93]. Without such a proof, skeptics may attribute such speedup to the well known hardware phenomena encountered when using multiple CPUs (better cache alignment, less communal overhead per tree, etc. [112]), or even chance. [93] also investigates using distributed path planning for manipulator arm problems and in non ad-hoc architectures.

A closely related work is [23], where a cluster of CPUs is used, and each CPU independently

builds a random tree. However, a major difference is that no data is exchanged between CPUs during the search. Although super-linear speedup is observed ($E = 1.2$) on clusters of two CPUs, larger clusters have $E < 1$. In contrast, C-FOREST exchanges data during the search process, achieves super-linear speedup for much larger clusters (e.g., 64 CPUs), and the efficiency observed is an order of magnitude larger ($E > 9$).

Probabilistic Road-Map (PRM), a popular multi-query planning algorithm, has been shown to be 'embarrassingly' parallel [4, 120]. Parallelization is achieved by having each CPU randomly sample and connect new points to the graph. Results show approximately linear speedup vs. the number of CPUs. While single-query planners can be parallelized in the same way, doing so requires each CPU to have memory access to the graph. In contrast, in C-FOREST each CPU maintains its own memory footprint, and knowledge transfer is achieved via message passing. This allows C-FOREST to be distributed between processes that do not necessarily have access to shared global memory (e.g., a networked computing cluster).

The PRM and Expansive Space Tree (EST) algorithms have also been parallelized in a message passing architecture by [100, 101]. Here a multi-query planner grows multiple trees at *different* locations in the configuration space, and trees are connected if they grow close together. Master CPUs pick the root node of each tree and check for tree combinations. Slave CPUs each grow a single tree rooted at a different place in the configuration space. Although the use of multiple trees is similar to C-FOREST, a fundamental difference is that each tree only spans a small portion of the environment—i.e. a path from start to goal will necessarily move through multiple trees that have been connected. In general, the idea in [100, 101] achieves powerful but usually sub-linear speedup. While super linear speedup is observed on a few trials (up to $E = 1.12$), no theoretical explanation is given as to why it occurs. In contrast, a main contribution of our work is the theoretical justification for super linear speedup. Further, C-FOREST trees are rooted at the *same* location, and so 1 to $T - 1$ of T CPUs can simultaneously fail (e.g., power off), and C-FOREST will still find a solution (although not as quickly). Other differences include the fact that C-FOREST is a single-query algorithm, C-FOREST CPUs are homogeneous with one phase

of operation, and we observed efficiencies up to an order of magnitude larger ($E > 9$).

2.2.3 Other forest based algorithms

The Reconfigurable Random Forest algorithm or RRF [81] also uses multiple random trees. RRF is a replanning algorithm where old trees, disconnected by obstacle movement, are saved and tested for connection vs. the current tree. The assumption is that most free-space remains unchanged, and so nodes from old trees will be useful if they can be reconnected to the current tree. Updated versions of the idea are explored by [46] and [130] and called "Lazy Reconfiguration forest" and "Multipartite RRTs," respectively. The two major distinctions between all of these ideas and C-FOREST are: (1) previous work only grows one tree at a time, and (2) previous work has not used a parallel architecture.

The closest work to sequential C-FOREST is *Any-Time RRT*, which builds new trees while time remains, such that each new tree is guaranteed to be better than its predecessor [43, 45]. However, the algorithm runs on a single CPU and the next tree is not started until after the previous tree has been completed and destroyed. In contrast, C-FOREST builds all trees simultaneously. Since multiple trees exist at one time, cooperation between them contributes to search progress.

It is also worth mentioning that, in the field of machine learning, forests of decision trees have proven to be much more powerful than a single decision tree for the problems of classification and regression [21]. While the tasks of regression and classification are quite different from the path planning problem I am concerned with, this body of work is an interesting analogue.

2.3 Contributions of this dissertation

A major difference between my work and previous work is that previous algorithms have been targeted at a single point in the completeness vs. computation vs. communication algorithm-space. The algorithms explored in this dissertation automatically adjust their location within that space to accommodate the problem that the system is currently solving and the resources that are available. Another important difference is that the algorithms in my work leverage the distributed-computing

power of the robotic team to help find better solutions more quickly. In contrast, the vast majority of previous work that uses a team has required *each* agent to calculate an entire solution completely on its own. For instance, in prioritized planning each robot can calculate its own path (assuming it respects robots of higher priority), and in decoupled planning each robot can individually calculate its own phase-1 solution (although these must be assembled by a single agent in phase-2). However, both prioritized planning and decoupled planning are incomplete, while the algorithms I present are probabilistically complete.

With respect to dynamic teams, I delay team formation/combination until it becomes necessary to prevent collisions. By keeping team sizes small, I hope to minimize problem complexity per team. I also perform experiments in a much larger workspace that subjects robots to actual (i.e., not simulated) wireless communication disturbances.

In general, the C-FOREST component of this dissertation differs from previous work on single-query path planning in the following ways: C-FOREST has provably super linear speedup vs. the number of CPUs. C-FOREST uses trees that have identical root and goal, grow concurrently, and span the entire configuration space. The trees in C-FOREST actively cooperate to help one other find better solutions, including the engrafting of beneficial branches onto other actively planning trees in the forest. C-FOREST has 1 phase of operation, 1 tunable parameter (beyond those of the underlying tree), and no dedicated scheduler or master node(s). C-FOREST has relatively little overhead (e.g., no pre-calculation of the partitioning problem).

This differs from previous ideas that place roots at different locations in the c-space then connect trees [100, 101], sample from deleted trees [46, 81, 130], delete old trees and regrow new trees from scratch [43], use multiple cores that require shared memory [4, 120], or have no theoretical explanation for the observed super linear speedup [23, 100].

Chapter 3

C-FOREST: distributed path planning with super linear speedup

Path planning algorithms calculate a sequence of actions that cause a system to transition from an initial state to a goal state. By abstracting the task of maneuvering a robotic system (e.g., manipulator arm, single rover, or robotic team), path planning provides basic functionality that facilitates many autonomous or semi-autonomous robotic applications.

As robots become integrated into everyday life, they must perform increasingly complex real-time path planning tasks, in order to ensure safe and efficient operation. Practical real-time solutions to simple problems are found using *Any-Time* algorithms that find a suboptimal actionable solution quickly, then refine it as time remains. Recent breakthroughs have provided algorithms with almost surly optimal convergence [64, 91]. However, the rate of convergence is dependent on problem complexity. Increasing the convergence rate will allow harder problems to be solved in real-time and facilitate the transition of autonomous robots from industrial technology to mainstream consumer goods. In addition to enabling movement that is intuitive and predictable to nearby humans, better path quality will decrease power consumption, decrease task time, and increase safety.

Meanwhile, parallelization is becoming established in computer hardware. As current manufacturing technologies reach their physical limitations with respect to clock-speed, computational power is increased by adding more processing cores to a computational unit (CPU), adding more CPUs to a computer, and linking computers in high performance clusters. Algorithms that leverage the power of parallel processing will have a significant advantage over those designed to run on a

single CPU.

I present a parallel algorithm for single-query high-dimensional path planning called *Coupled Forest Of Random Engrafting Search Trees* (C-FOREST). C-FOREST is designed to be run on T CPUs that communicate (e.g., a distributed computer with T CPUs, a networked cluster of T computers, or even a T-core processor). In the basic version, each CPU builds a *probabilistically independent* search tree between the *same* start and goal states. Message passing enables new exploration and pruning (of all trees) to be a function of the current best solution known to any tree in the forest. Solution branches are also exchanged so they can be engrafted onto and improved by other trees.

I assume that some mechanism exists *a priori* with the ability to propagate information throughout whatever distributed architecture is being used (e.g., a message passing protocol). I also assume that the time it takes to propagate information from one CPU to all others is small compared to the time required to solve the centralized multi-robot path planning problem. While this assumption may not apply in extreme cases, I believe that it is a fair assumption for the majority of standard distributed architectures that are used in practice.

I prove that parallel C-FOREST can have super linear speedup vs. the number of CPUs (i.e., trees). That is, $S > T$, and $E > 1$. Given the potential for super linear speedup, C-FOREST can also benefit non-parallel path planning. I propose a modified version that sequentially divides computation between trees on a single CPU. Although I believe that parallel algorithms represent the future of path planning, sequential C-FOREST provides a simple way to increases the power of existing hardware—it also highlights the power of the C-FOREST framework.

C-FOREST can utilize any underlying random tree algorithm and configuration space, as long as two conditions are met: (1) the configuration space obeys the triangle inequality, (2) the underlying tree is expected to converge to an optimal solution given infinite time.

3.1 Parallel C-FOREST

In the parallel version of C-FOREST each CPU builds a single *probabilistically independent* search tree between the *same* start and goal states. Message passing enables random exploration and pruning (of all trees) to be a function of the current best solution known to any tree. Thus, all trees avoid exploring regions of the configuration space that cannot produce globally better solutions, and all trees prune themselves of globally outdated nodes. This significantly reduces the time required to insert new nodes because node insertion time is dependent on the number of nodes already in a tree. Message passing also allows the current best solution to be engrafted onto and then improved by any other tree. This increases the chance that new exploration will yield better solutions by expanding the tree into regions of the configuration space that are known to be beneficial.

3.1.1 Algorithm description

Let $h(v_1, v_2)$ be an admissible heuristic function that returns an estimate of the distance between two states v_1 and v_2. Because the function is admissible, it will never overestimate the distance between v_1 and v_2. In the case of multiple goal states $h(v_1, g)$ is assumed to be admissible over the entire set of goals. In other words, the distance returned is less than or equal to the distance to any member of the goal set. More accurate distance estimates returned by $h(v_1, v_2)$ will tend to increase algorithmic performance. However, if no suitable heuristic can be found then it is possible to define $h(v_1, v_2) \equiv 0$.

Let $h_s(v) = h(s, v)$ and $h_g(v) = h(v, g)$. Let $\mathbf{t}.d_s(v)$ be the actual distance from the start to v through tree $\mathbf{t}$. The tree that discovered $\mathbf{P}_{bst}$ is $\mathbf{t}_{bst}$ and $L = \mathbf{t}_{bst}.d_s(g)$. Assuming that at least one path to the goal has been found, any point v for which $h_s(v) + h_g(v) \geq L$ cannot possibly lead to a better $\mathbf{P}_{bst}$. Geometrically, $h_s(v) + h_g(v) = L$ describes an enclosed region in the search space (see Figure 3.1). Let the space within this region be denoted $\mathbf{A}_L$. In Euclidean space $\mathbf{A}_L$ is bounded by an ellipsoid.

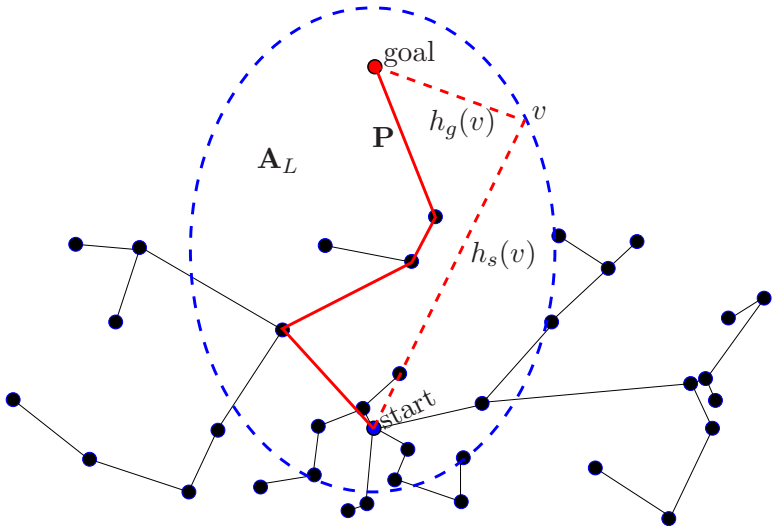


Figure 3.1: Boundary beyond which new points cannot lead to better solutions (dashed-blue). The space within the region is denoted $\mathbf{A}_L$ and its boundary is defined by $h_s(v) + h_g(v) = L$, where L is the length of the current best path $\mathbf{P}_{bst}$ (red).

The parallel C-FOREST algorithm is displayed in Figure 3.2-Left-Top. $\mathbf{P}_{bst}$ is initialized to the empty set and its length to ∞ (lines 1-2). Next, T trees are started, each on their own CPU (lines 3-4). Each tree is a separate, probabilistically independent, version of a random search tree (e.g., RRT*). The subroutine **TimeLeft**() returns *true* while planning time remains, otherwise it returns *false*. Once the allotted planning time has been exhausted, the best solution is returned (lines 5-7).

The bulk of the algorithm takes place within each individual tree $\mathbf{t}$ in **RandomTree**($\mathbf{t}$) (Figure 3.2-Right-Top). **RandomTree**($\mathbf{t}$) initializes $\mathbf{t}.\mathbf{P}_{bst}$ to the empty set and $\mathbf{t}.L$ to ∞ (lines 1-2). Search happens by picking a random point v from the configuration space using **RandomPoint**(L) and then inserting it into $\mathbf{t}$ with $\mathbf{t}$.**Insert**(v) (lines 4-5). **RandomPoint**(L) and **Insert**(v) are assumed to incorporate any specific logic required by the underlying tree and/or configuration space. If adding v leads to a (globally) better path, then the new solution is distributed to the other trees (lines 7-8). This can be accomplished in shared memory or via messaging passing. If a better solution is found by another tree, then it is added to the local tree using **AddPath**($\mathbf{P}_{bst}$) (lines 11-15). The local tree is pruned based on the global value of L using $\mathbf{t}$.**Prune**(L) (lines 9 and 14),

ParallelCFOREST()
1: $L = \infty$
2: $\mathbf{P}_{bst} = \emptyset$
3: **for** $\forall \mathbf{t} \in \mathbf{T}$ **do**
4: **RandomTree**($\mathbf{t}$) on its own CPU
5: **while TimeLeft**() **do**
6: sleep
7: **Return** $(L, \mathbf{P}_{bst})$

t.**AddPath**($\mathbf{P}_{bst}$)
1: **for** $i = 2$ to $|\mathbf{P}_{bst}|$ **do**
2: **if not** $\mathbf{t}$.**InTree**($\mathbf{P}_{bst,i}$) **then**
3: $\mathbf{t}$.**Insert**($\mathbf{P}_{bst,i}, \mathbf{P}_{bst,i-1}$)

$v =$ **RandomPoint**(L)
1: **repeat**
2: $v = (\mathbf{Rand}(0,1) * (c - b)) + b$
3: **until** $h_s(v) + h_g(v) < L$

$\mathbf{t}$.**Prune**(L)
1: **for** $\forall$ nodes $n \in \mathbf{t}$ **do**
2: **if** $h_s(n) + h_g(n) \geq L$ **then**
3: remove n and its descendants

SetSampleBounds(L)
1: $a = (L - |s - g|)/2$
2: $b = \max\{\min\{s, g\} - a, \mathbf{MinBounds}()\}$
3: $c = \min\{\max\{s, g\} + a, \mathbf{MaxBounds}()\}$

RandomTree($\mathbf{t}$)
1: $\mathbf{t}.L = \infty$
2: $\mathbf{t}.\mathbf{P}_{bst} = \emptyset$
3: **while TimeLeft**() **do**
4: $v =$ **RandomPoint**(L)
5: $\mathbf{t}$.**Insert**(v)
6: **if** $\mathbf{t}.L < L$ **then**
7: $\mathbf{P}_{bst} = \mathbf{t}.\mathbf{P}_{bst}$
8: $L = \mathbf{t}.L$
9: $\mathbf{t}$.**Prune**(L)
10: $\mathbf{t}$.**SetSampleBounds**(L)
11: **else if** $L < \mathbf{t}.L$ **then**
12: $\mathbf{t}$.**AddPath**($\mathbf{P}_{bst}$)
13: $\mathbf{t}.L = L$
14: $\mathbf{t}$.**Prune**(L)
15: $\mathbf{t}$.**SetSampleBounds**(L)

$\mathbf{t}$.**Insert**(v)
1: $p_v =$ prospective parent of v according to the random tree algorithm being used
2: **if** $\mathbf{t}.d_s(p_v) + h(p_v, v) + h_g(v) < L$ **then**
3: insert v according to the random tree algorithm

$\mathbf{t}$.**Insert**(v, p_{bst})
1: $p_v =$ prospective parent of v according to the random tree algorithm being used
2: **if** $\mathbf{t}.d_s(p_v) + h(p_v, v) < \mathbf{t}.d_s(p_{bst}) + h(p_{bst}, v)$ **then**
3: **if** $\mathbf{t}.d_s(p_v) + h(p_v, v) + h_g(v) < L$ **then**
4: insert v according to the random tree algorithm
5: **else if** $\mathbf{t}.d_s(p_{bst}) + h(p_{bst}, v) + h_g(v) < L$ **then**
6: insert v according to the random tree algorithm with p_{bst} as its parent

Figure 3.2: Algorithm for parallel C-FOREST (Left-Top), and forest tree (Right-Top), and selected subroutines. Note that any random tree algorithm can be used, as long as it provides the necessary subroutines. **MinBounds**() and **MaxBounds**() return the minimum and maximum coordinates of the configuration space along each dimension.

and the sampling bounds are updated using $\mathbf{t}$.**SetSampleBounds**(L) (lines 10 and 15).

The subroutine **AddPath**($\mathbf{P}_{bst}$) is shown in Figure 3.2-Left-Center. Line 1 iterates over nodes in $\mathbf{P}_{bst}$ from start to goal (the start node does not need to be inserted, since it is guaranteed to exist in $\mathbf{t}$). The subroutine $\mathbf{t}$.**InTree**($\mathbf{P}_{bst,i}$) checks if $\mathbf{P}_{bst,i}$ is already in $\mathbf{t}$ to avoid duplicating points (line 2). If $\mathbf{P}_{bst,i}$ does not exist in $\mathbf{t}$ then it is inserted on line 3. **Insert**($v, \mathbf{P}_{bst,i-1}$) is a modified version of **Insert**(v) that explicitly includes $\mathbf{P}_{bst,i-1}$ in the possible neighbor set, but is

otherwise identical. This ensures that $\mathbf{P}_{bst,i-1}$ can be the parent of $\mathbf{P}_{bst,i}$, but allows better nodes to be used if they exist.

Points not in $\mathbf{A}_L$ can be ignored for random sampling (**RandomPoint**(L), line 6, Figure 3.2). Nodes not in $\mathbf{A}_L$ can also be pruned (as is done in **prune**(L), line 2). Sampling directly from $\mathbf{A}_L$ can be difficult in practice. Instead, an initial sampling is taken from the hypercube described by $h_s(v) + h_g(v) \leq L$ per each dimension (**SetSampleBounds**(L), lines 1-3), and then points are disregarded if they are outside $\mathbf{A}_L$ (**RandomPoint**(L), lines 1-3).

I have also found it useful to disregard any points for which $\mathbf{t}.d_s(v) + h_g(v) \geq L$ (**Insert**(v), line 2). That is, points that cannot lead to a better solution given their current distance-to-root through the tree plus the heuristic estimate of the distance to goal. This is a greedy strategy, since it does not account for the fact that future tree-remodeling may decrease $\mathbf{t}.d_s(v)$, and is similar to the priority heap weight used in the A* algorithm. in a similar greedy approach, the descendants of pruned nodes are also themselves pruned (**prune**(L), line 3).

Coupling the sampling and pruning mechanisms of all trees enables the entire forest to grow based on the best solution found so far. Sharing the current best solution gives all trees a chance to improve it.

3.1.2 Sampling analysis

The rational for growing T trees in parallel relies on the assumption that trees are probabilistically independent—since it is desirable that each CPU grow a unique tree. I now formally address this issue.

Theorem 3.1: *Assuming trees are probabilistically independent and built over a continuous configuration space, a C-FOREST almost surly does not suffer from repeated exploration by multiple trees.*

Proof. The space is continuous, so the chance any two trees sample the same point is 0. Thus, the probability different trees generate identical paths is also 0. □

3.2 Super linear speedup analysis

I now prove that C-FOREST can have super linear speedup due to its design. I begin by calculating the probability a newly sampled point immediately leads to a better solution in a single tree. The time required to insert a new node into a tree is dependent on the number of nodes already in the tree (e.g., $O(f(n))$ if the tree contains n nodes). Assuming we know the runtime of the node insertion function $f(n)$, it is possible to calculate the expected time until a better solution is found using a single tree. Similar analysis can be used to calculate a bound on the expected time to find a better solution using a T tree parallel C-FOREST. Comparing the expectations shows when super linear speedup can occur for the specific task of finding a *better* solution. The rest of the proof follows using induction over the task of finding each additional better solution. Super linear speedup can only occur after the first solution has been found—therefore, the task of finding the first solution does not have super linear speedup. However, efficiency > 1 after the first solution will eventually cause efficiency > 1 for the entire run.

It is important to note that I assume information sharing between trees is instantaneous. In future work I hope to investigate the effects of variable message transmission time. However, I believe that the current analysis can be used to predict performance in many practical situations for a number of reasons. First, data sharing is practically instantaneous in distributed architectures that use shared memory. Second, in most standard message based distributed architectures that are currently used, the time required to find a better solution to a multi-dimensional path-planning problem is likely to be much greater (by orders of magnitude) than the time required to propagate a message from one node to the rest of the architecture. Finally, in the following analysis I make a number of assumptions that tend to underestimate the performance of C-FOREST with respect to practical vs. the theoretical performance and, while it is technically incorrect to make any claims based on this, they tend to mitigate the assumption of perfect communication.

Consider the search of $\mathbf{t}'$, a single stand-alone tree. $\mathbf{A}_{\mathbf{P}_{now}}$ is the union of all points not in $\mathbf{t}'$ that would immediately result in a better solution if added, $\mathbf{A}_{\mathbf{P}_{now}} \subseteq \mathbf{A}_L$, where $\mathbf{A}_L$ is described

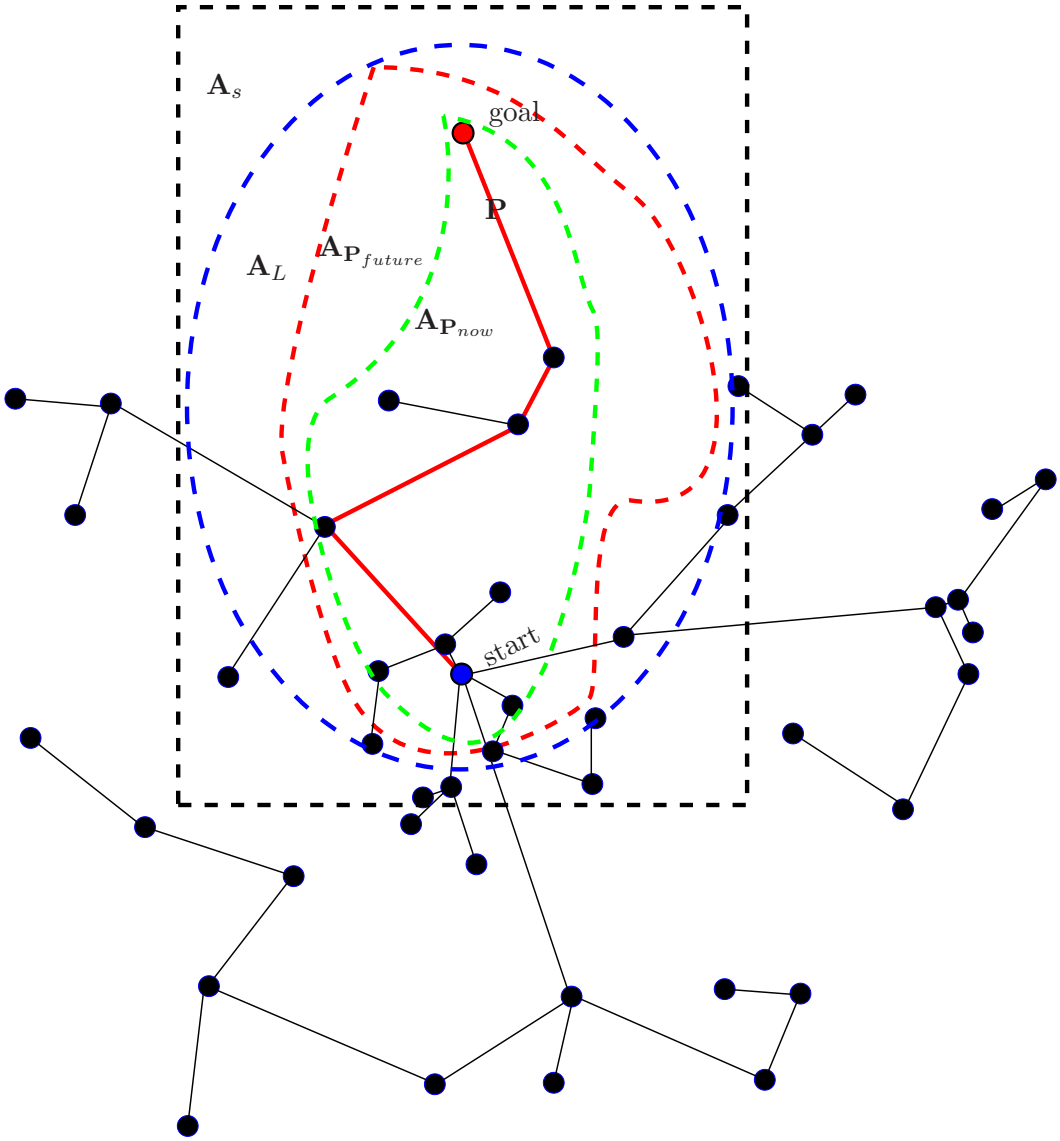


Figure 3.3: Subspaces $\mathbf{A}_{\mathbf{P}_{now}}$ (dashed-green), $\mathbf{A}_{\mathbf{P}_{future}}$ (dashed-red), $\mathbf{A}_L$ (dashed-blue), and $\mathbf{A}_s$ (dashed-black).

in Section 1.2. Points in $\mathbf{A}_{\mathbf{P}_{now}}$ are visible to $\mathbf{t}'$ (e.g., not blocked by collisions) and fulfill all other requirements necessary to be connected to $\mathbf{t}'$ (or cause some other point v_2 to be connected).[1] Verification of $v \in \mathbf{A}_{\mathbf{P}_{now}}$ is trivial, assuming we are alerted when better solutions are found.

The subspace $\mathbf{A}_{\mathbf{P}_{future}}$ is the union of all points not in the tree that, if *eventually* added,

[1] In some algorithms sampling v will cause a different point v_2 to be added to $\mathbf{t}'$ (e.g., in RRT if v is more than δ away from $\mathbf{t}'$ then v_2 is the point δ away from $\mathbf{t}'$ along the line from v to the closest node in $\mathbf{t}'$). For the purposes of this proof, I assume that v is the point sampled from the configuration space, even if v_2 is added to $\mathbf{t}'$.

would enable a better path to *eventually* be found. $\mathbf{A}_{\mathbf{P}_{now}} \subseteq \mathbf{A}_{\mathbf{P}_{future}}$ and $\mathbf{A}_{\mathbf{P}_{future}} \subseteq \mathbf{A}_L$. $\mathbf{A}_{\mathbf{P}_{future}}$ can be broken into two disjoint subspaces: $\mathbf{A}_{\mathbf{P}_{future,see}}$ contains points visible to the current tree, and $\mathbf{A}_{\mathbf{P}_{future,blind}}$ contains points not visible due to obstacle collisions. By construction $\mathbf{A}_{\mathbf{P}_{now}} \subseteq \mathbf{A}_{\mathbf{P}_{future,see}}$. I assume $\mathbf{A}_{\mathbf{P}_{future,see}}$ cannot be calculated in practical time. This is a reasonable assumption assuming the dimensionality of the configuration space is greater than three. Note that if $\mathbf{A}_{\mathbf{P}_{future,see}}$ can be calculated quickly, then non-randomized algorithms such as gradient descent should be used instead.

Let $\mathbf{A}_s$ denote the sample space. Ideally $\mathbf{A}_s = \mathbf{A}_L$; however, in practice $\mathbf{A}_s \supseteq \mathbf{A}_L$. Figure 3.3 depicts the relationship between $\mathbf{A}_{\mathbf{P}_{now}}$, $\mathbf{A}_{\mathbf{P}_{future}}$, $\mathbf{A}_L$, and $\mathbf{A}_s$. Let the Lebesgue measure (i.e., volume) of a subspace be denoted $\|\cdot\|_\ell$.

Lemma 3.1: *Using a single search tree, the probability a point sampled uniformly at random leads to a better solution is given by:* $\mathrm{P}(v \in \mathbf{A}_{\mathbf{P}_{now}}) = \frac{\|\mathbf{A}_{\mathbf{P}_{now}}\|_\ell}{\|\mathbf{A}_s\|_\ell}$.

Proof. Assuming points are sampled uniformly at random from $\mathbf{A}_s$, and a subspace $\mathbf{A} \subseteq \mathbf{A}_s$, it is possible to express the probability of choosing a point in $\mathbf{A}$ as $\mathrm{P}(v \in \mathbf{A}) = \|\mathbf{A}\|_\ell / \|\mathbf{A}_s\|_\ell$. Replacing $\mathbf{A}$ with $\mathbf{A}_{\mathbf{P}_{now}}$ finishes the proof. □

Lemma 3.2: *Using a single tree and sampling uniformly at random from* $\mathbf{A}_s$*, the probability of picking a point in* $\mathbf{A}_{\mathbf{P}_{now}}$ *will not decrease until a better path is found.*

Proof. If $\mathbf{A}_{\mathbf{P}_{now}}$ changes due to a node insertion that does not result in a better path being found, then it will not get smaller (and may get bigger). Thus, $\mathbf{A}_{\mathbf{P}_{now}}$ will not get smaller until a better path is found. Applying Lemma 1 finishes the proof. □

Let p_j be the probability that the j-th sampled point immediately results in a better solution after the algorithm has sampled $j-1$ nodes (either successfully or unsuccessfully) $p_j = \mathrm{P}(v \in \mathbf{A}_{\mathbf{P}_{now}})$. Assuming a better solution can actually be found, $0 < p_j \leq 1$. Let $f(n)$ represent the insertion *time* required to add a new node to a tree, assuming n nodes are already in the tree (e.g., $f(n) = \log(n)$

or $f(n) = n$). In all non-trivial random trees $f(n) = \Omega(c)$, for a constant c, since attaching new nodes in a useful way requires performing a search over the set of nodes already in the tree.

Assuption 3.1: *The runtime of $f(n)$ is less than exponential in n.*

The reason assumption 3.1 is introduced will become clear later. This is a valid assumption given that, for all popular randomized search algorithms that I am aware of, $f(n) = O(n)$ or $f(n) = O(log(n))$.

Lemma 3.3: *Using a single stand-alone tree $\mathbf{t}'$, the expected time to find a better solution is:*

$$\mathrm{E}_{\mathbf{t}'} = \sum_{j=1}^{\infty} \left[(1 - p_j)^{j-1} p_j \left(\sum_{i=0}^{j-1} f(n+i) \right) \right] \tag{3.1}$$

Proof. Assuming every sampled point is added to the tree, the expected time to insert j nodes is given by the cumulative insertion time of nodes 1 through j. This is given by $\sum_{i=0}^{j-1} f(n+i)$. The probability the j-th node yields the first better solution is $(1 - p_j)^{j-1} p_j$. The expected time $\mathrm{E}_{\mathbf{t}'}$ required to find a better solution is calculated by summing over the total time until each node j is inserted, weighted by the probability that j yields the *first* better solution. □

Now assume that a C-FOREST $\mathbf{T}$ contains T random trees, such that all trees simultaneously search for a path between the same start and goal locations in the configuration space. Let $p_{\mathbf{t},j}$ denote the probability the j-th new insertion into tree $\mathbf{t} \in \mathbf{T}$ leads to a better solution. If all trees simultaneously add a new point, then the probability that at least one tree immediately finds a better solution is:

$$\mathrm{P}_{\mathbf{T},j} = 1 - \prod_{t=1}^{T} (1 - p_{\mathbf{t},j}) \tag{3.2}$$

An upper bound on the expected time for C-FOREST to find a better solution can be calculated using a similar process to the one employed in Lemma 3.

Lemma 3.4: *Using a C-FOREST $\mathbf{T}$, an upper bound on the expected time to find a better solution is:*

$$\mathrm{E}_{\mathbf{T}} \leq \sum_{j=1}^{\infty} \left[(1 - \mathrm{P}_{\mathbf{T},j})^{j-1} \mathrm{P}_{\mathbf{T},j} \left(\sum_{i=0}^{j-1} f(n_{max} + i) \right) \right] \tag{3.3}$$

Proof. If all trees require the same amount of time to insert a new node (i.e., they are all the same size n), then the expected number of iterations until a better path is found is $1/\mathrm{P}_{\mathbf{T},j}$. In practice, different trees may have different n and different values for $f(n)$. An upper bound on the expected time to find a new better path can be calculated using the maximum $f(n)$ of any $\mathbf{t} \in \mathbf{T}$ per each iteration. Assuming the insertion function is non-decreasing, $f(n+1) \geq f(n)$ for all $n > 0$, then it follows that $f(\max_{\mathbf{t}\in\mathbf{T}}(n_{\mathbf{t}}) + i) = \max_{\mathbf{t}\in\mathbf{T}} f(n_{\mathbf{t}} + i)$. Substituting $n_{max} = \max_{\mathbf{t}\in\mathbf{T}}(n_{\mathbf{t}})$ finishes the proof. □

Comparing Lemmas 3.3 and 3.4 demonstrates how a single search tree is expected to perform vs. parallel C-FOREST, respectively. We would like to know what conditions, if any, allow super linear speedup. This is also of interest because it shows when sequential C-FOREST will theoretically outperform a single tree (e.g., using a single processor and splitting computation time between each tree). In particular, we want to see when Equation 3.3 holds for popular search-tree algorithms (e.g., when $f(n)$ is linear, logarithmic, etc.). Let $p_{min,j} = \min_{\mathbf{t}\in\mathbf{T}}(p_{\mathbf{t},j})$.

Lemma 3.5: *Assuming $n_{max} \leq n$ and $p_{min,j} \geq p$, the expected time required to find a better path is less for C-FOREST $\mathbf{T}$ than for a single tree $\mathbf{t}'$.*

Proof. This follows directly from Lemmas 3 and 4, since $0 < p_{min,j} \leq 1$ and $0 < p_j \leq 1$, and $T > 1$. □

Assuming the probability a new node leads to a better path is the same for all C-FOREST trees and the single stand-alone tree gives the following:

Lemma 3.6: *Assuming $n_{max} \leq n$ and $p_{\mathbf{t},j} = p_j$ for all $\mathbf{t} \in \mathbf{T}$, parallel C-FOREST with $T > 1$ is expected to find a better path before a single tree $\mathbf{t}'$.*

Proof. $p_{\mathbf{t},j} = p_{min,j}$ because all $p_{\mathbf{t}}$ are the same. The rest follows from Lemma 5. □

Both Lemmas 3.5 and 3.6 make intuitive sense, since drawing more samples from a random distribution increases the probability of finding what we are looking for. Note that

$\lim_{T\to\infty} 1 - \prod_{t=1}^{T} (1 - p_{min,j}) = 1$, and as a result, the expected time to a new solution approaches 0 as T approaches infinity.

Substituting $n_{max} = n$ in Lemma 3.5 shows that the expected time to a solution is less for parallel C-FOREST than for a stand alone tree, regardless of super linear speedup or not. Therefore, the number of nodes in a C-FOREST tree $\mathbf{t} \in \mathbf{T}$ is likely to be less than that in a stand-alone tree $\mathbf{t}'$ (i.e., $n_{max} \leq n$)—especially after a few solutions have already been found—due to pruning and reduced sample space caused by decreasing L. Smaller tree size translates into a smaller insertion time per new node, which reinforces a reduced expected time until the next better path is found by $\mathbf{t} \in \mathbf{T}$ vs. $\mathbf{t}'$. It is desirable to know how large C-FOREST trees can get (n_{max}), relative to a stand-alone tree (n), while still facilitating super linear speedup.

The basic idea is to compare Equations 3.1 and 3.3, assume that all trees are equally likely to find a solution, and then solve for a bound on n_{max} in terms of n. In order to reduce mathematical complexity I will make a couple of assumptions that favor the expected runtime of the stand-alone tree vs. C-FOREST. Therefore, the resulting bound on n_{max} will be looser than what one might expect in practice. For notational ease, let $q = 1 - p$, where $0 \leq q < 1$.

Theorem 3.2: *C-FOREST will have super linear speedup for finding the next best solution when:*

$$\sum_{i=0}^{\infty} q^{Ti} f\left(n_{max} + i\right) < \frac{1}{T} \sum_{i=0}^{\infty} q^{i} f\left(n + i\right) \tag{3.4}$$

Proof. By definition, speedup is super linear when $\mathrm{E}_{\mathbf{T}} < \mathrm{E}_{\mathbf{t}'}/T$. Substituting Equations 3.1 and 3.3 yields:

$$\sum_{j=1}^{\infty} \left[(1 - \mathrm{P}_{min,j})^{j-1} \mathrm{P}_{min,j} \left(\sum_{i=0}^{j-1} f(n_{max} + i) \right) \right] < \frac{1}{T} \sum_{j=1}^{\infty} \left[(1 - p_j)^{j-1} p_j \left(\sum_{i=0}^{j-1} f(n + i) \right) \right]$$

I assume that for insertion j all trees are equally likely to find a better solution, $p_j = p_{\mathbf{t},j} = p_{min,j}$. This favors the stand-alone tree vs. C-FOREST—in practice we expect $p_{\mathbf{t},j} > p_j$ for some $\mathbf{t} \in \mathbf{T}$—and so the inequality is maintained. Using the definition of $\mathrm{P}_{\mathbf{t},j}$ from Equation 3.2 and substituting $p_{\mathbf{t},j} = p_j$ gives:

$$\sum_{j=1}^{\infty}\left[\left(\prod_{t=1}^{T}(1-p_j)\right)^{j-1}\left(1-\prod_{t=1}^{T}(1-p_j)\right)\left(\sum_{i=0}^{j-1}f(n_{max}+i)\right)\right] < \frac{1}{T}\sum_{j=1}^{\infty}\left[(1-p_j)^{j-1}p_j\left(\sum_{i=0}^{j-1}f(n+i)\right)\right]$$

The dependency of p_j on j is mathematically challenging. Observe that p_j is expected to increase as a function of j as the search tree grows into the configuration space. If p_j were to remain fixed at its $j = 1$ value until the next best solution was found, then C-FOREST would suffer a greater hit to its expected time than a stand-alone tree. This is because the increased sampling power of T trees gives C-FOREST a greater chance of increasing p_j vs. the stand alone tree. Therefore, the inequality continues to be maintained if we substitute $p = p_j = p_1$. Performing this substitution and then simplifying gives:

$$\sum_{j=1}^{\infty}\left[(1-p)^{T(j-1)}\left(1-(1-p)^{T}\right)\left(\sum_{i=0}^{j-1}f(n_{max}+i)\right)\right] < \frac{1}{T}\sum_{j=1}^{\infty}\left[(1-p)^{j-1}p\left(\sum_{i=0}^{j-1}f(n+i)\right)\right]$$

Assumption 3.1 implies $f(n) = o(c^n)$, note the little 'o'. Thus, $\lim_{m\to\infty}(1-p)^m f(n+m-1) = 0$ and $\lim_{m\to\infty}(1-p)^{Tm}f(n+m-1) = 0$. Using these facts to simplify the previous equation, and then substituting $1-q = p$ finishes the proof. □

Theorem 3.2 leads to the following corollaries about linear and logarithmic insertion functions, respectively:

Corollary 3.1: *Parallel C-FOREST with trees using a linear insertion function will have super linear speedup finding a better solution when:*

$$n_{max} < \frac{(1-q^T)(pn+q)}{Tp^2} - \frac{q^T}{1-q^T} \tag{3.5}$$

Proof. With a linear insertion function $f(n) = c_1 n$, where c_1 is a constant greater than 0. Substituting into Equation 3.4, c_1 cancels from either side:

$$\sum_{i=0}^{\infty}q^{Ti}\left(n_{max}+i\right) < \frac{1}{T}\sum_{i=0}^{\infty}q^{i}\left(n+i\right) \tag{3.6}$$

Solving for the limit of the sum as the number of terms approaches infinity finishes the proof. □

Corollary 3.2: *Parallel C-FOREST with trees using a logarithmic insertion function will have super linear speedup finding a better solution when:*

$$\sum_{i=0}^{\infty} q^{Ti} \log_2 (n_{max} + i) < \frac{1}{T} \sum_{i=0}^{\infty} q^i \log_2 (n + i) \tag{3.7}$$

Proof. With a logarithmic insertion function $f(n) = c_2 \log_2(n)$, where c_2 is a constant greater than 0. Substituting into Equation 3.4, c_2 cancels from either side. Although I am unable to find a more elegant closed-form solution, it is still possible to evaluate the inequality numerically using partial summations to obtain an estimate that is arbitrarily accurate. Note that the size of the terms rapidly decreases due to the exponentiation q^i vs. the slow growth of $\log_2(n + i)$. □

The final insertion function I examine is the insertion function used by RRT*.

Lemma 3.7: *Parallel C-FOREST with RRT* trees will have super linear speedup finding a better solution when:*

$$\sum_{i=0}^{\infty} q^{Ti} \log_2 (n_{max} + i) < \frac{1}{T} \sum_{i=0}^{\infty} q^i \log_2 (n + i)$$

Proof. In RRT* all nodes within a particular d-ball are evaluated for possible connection to a new node, where d is the dimensionality of the configuration space. The radius of the d-ball is calculated as $\min\left\{c_3((\log n)/n)^{1/d}, c_4\right\}$, where c_3 and c_4 are constants defined in terms of d (see [64] for more details). The volume of the d-ball is $\mathbf{A}_{ball} = \min\{c_5((\log_2 n)/n), c_6\}$, where c_5 and c_6 are constants dependent on d. Assuming nodes are evenly distributed in $\mathbf{A}_s$, then the expected number of nodes evaluated per insertion is $nr\|\mathbf{A}_{ball}\|_\ell / \|\mathbf{A}_s\|_\ell$, where r is the ratio between the number of nodes in $\mathbf{A}_s$ and the number of nodes in the tree. We can assume r is a constant if most nodes are in $\mathbf{A}_s$, and $r = 1$ if the tree is pruned as described in Figure 3.2. This gives an expected node insertion time of $f(n) = \min\{c_7 \log_2(n), c_8 n\}$, where c_7 and c_8 are constants. While it is possible to substitute this result into Equation 3.4 and then solve numerically as before, it should be noted that the logarithmic case wins out whenever $\log_2(n) < c_9 n$ and $\log_2(n_{max}) < c_9 n_{max}$, where $c_9 = c_8/c_7$. Therefore, Corollary 2 can be used for RRT*, assuming trees are sufficiently large. □

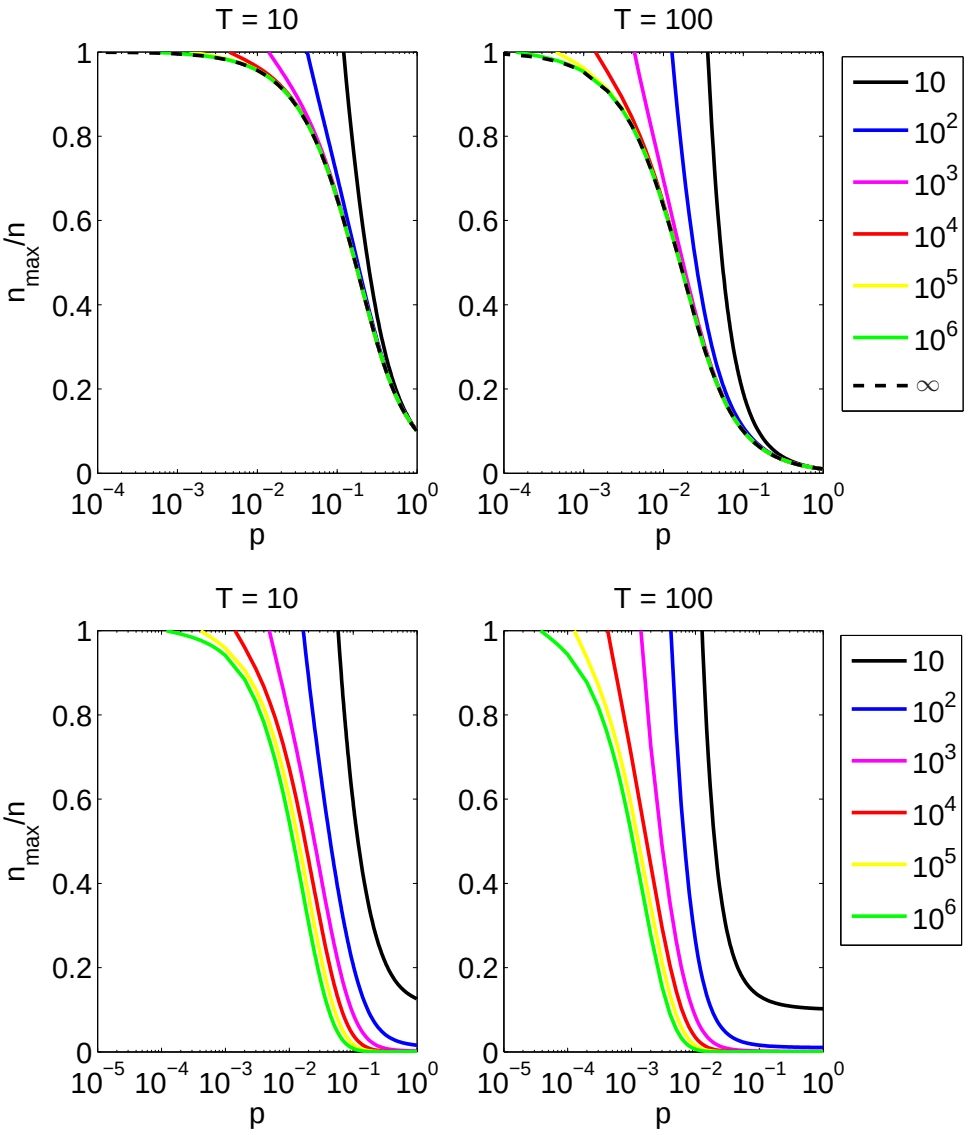


Figure 3.4: Maximum ratio n_{max}/n that allows super linear speedup of parallel C-FOREST vs. a single tree for various values of n, T, p, and $f(n)$. Linear and logarithmic $f(n)$ (Top and Bottom, respectively) for $T = 10$ and $T = 100$. Color indicates n. Super linear speedup occurs when environment and insertion function cause the performance of C-FOREST vs. a single tree to be beneath the curve.

Figure 3.4 displays the maximum ratio of n_{max}/n per p that will allow super linear speedup, given various values of n, T, $p = 1 - q$, and $f(n)$. The area beneath this curve represents the region in which the expected speedup of C-FOREST is super linear. For example, given $T = 10$ trees and $p = 10^{-1}$, if a single tree would have $n = 10^3$ nodes given a particular L, then super linear speedup is expected when C-FOREST trees have less than about $0.6n$ nodes. Recall that C-FOREST trees

require fewer nodes per L because they are expected to find solutions faster—regardless of super linear speedup or not. For linear $f(n)$ the ratio n_{max}/n approaches $(1-q^T)/(Tp)$ as n approaches infinity (represented by the black dashed line). Logarithmic $f(n)$ do not appear to have a (non-zero) limiting function as n approaches infinity. This means that, in the logarithmic case, it is possible C-FOREST trees may get too large for a particular p. Fortunately, if trees get this big, it is a good indication that p is relatively small. Also, the chances that a tree gets too large may be practically unlikely, since each order of magnitude larger n only reduces the super linear speedup range of p by a relatively small amount.

Theorem 3.2, Corollaries 3.1 and 3.2, and Lemma 3.7 show that C-FOREST can have super linear speedup with respect to the task of finding a better solution. However I have neglected the case of finding the initial solution. Estimating the time to initial solution is less straightforward than estimating the time to an improved solution because planning in nontrivial environments requires discovering paths that necessarily contain a minimum number of points (i.e., to avoid obstacles). Therefore, the probability a search tree finds an initial solution before the minimum number of points have been sampled is 0. Further complications arise due to the fact that both n and n_{max} start at 0 and will grow at the same rate—so it is too early to capitalize on the advantages provided by coupled pruning and sampling in C-FOREST. On the other hand, there is still a significant statistical advantage to using multiple trees in parallel—although definitely not super linear. Let the expected time to initial solution for a single stand-alone tree and C-FOREST to be defined as $\mathbf{E}_{\mathbf{t}',0}$ and $\mathbf{E}_{\mathbf{T},0}$, respectively. We know that $\mathbf{E}_{\mathbf{t}',0} \geq \mathbf{E}_{\mathbf{T},0}$ (since $\mathbf{t}'$ and $\mathbf{T}$ draw 1 and T samples from the distribution of all trees, respectively), and that the inequality increases with T.

Lemma 3.8: *If C-FOREST has super linear performance while finding better solutions and runs long enough, then it will eventually exhibit super-linear performance over its entire run.*

Proof. Theorem 3.2, Corollaries 3.1 and 3.2, and Lemma 3.7 show that C-FOREST can have efficiency > 1 finding better solutions. If efficiency remains greater than 1 long enough, then it will

eventually compensate for the sub-linear performance of the initial solution. □

Corollary 3.3: *C-FOREST may not exhibit super-linear performance if it does not run long enough.*

Proof. If efficiency is not greater than 1 long enough, then it cannot compensate for the sub-linear performance of the initial solution. □

Theorem 3.3: *C-FOREST can have super linear speedup over its entire run due to algorithmic properties.*

Proof. This is proved by induction. Lemma 8 implies that as long as Inequalities 3.5 or 3.7 hold for long enough, then there will eventually be a time at which performance becomes super linear with respect to the entire run. Let this improvement iteration be denoted τ_0 and represent the base case. The inductive step is given by the fact that for any improvement iteration τ_b beyond this, the algorithm will be super linear as long as the algorithm was super linear at τ_{b-1} and Inequalities 3.5 or 3.7 continue to hold (assuming their respective insertion functions are being used). □

It is important to note that Theorem 3.3 is merely an existence proof, it does not guarantee performance *will* be super linear. However, it is significant because it is the first proof demonstrating that super linear speedup in parallelized path planning can be attributed to algorithmic properties (i.e., instead of the hardware phenomena that sometimes enable super linear speedup for small numbers of CPUs—such as better cache alignment, etc.). It also highlights the specific properties responsible for super linear speedup in C-FOREST, and suggests the super linear speedup in C-FOREST may be applicable to large clusters.

Theorem 3.4: *Sharing* $\mathbf{P}_{bst}$ *can increase the chances that C-FOREST finds a better solution relative to only sharing* L*.*

Proof. Sharing L allows all trees in the forest to have identical $\mathbf{A}_L$ and $\mathbf{A}_s$. However, they still have different $\mathbf{A}_{\mathbf{P}_{now}}$. Sharing the points in $\mathbf{P}_{bst}$ may increase $\mathbf{A}_{\mathbf{P}_{now}}$ and will never decrease it.

SequentialCFOREST()
1: $L = \infty$
2: $\mathbf{P}_{bst} = \emptyset$
3: **for** $\forall \mathbf{t} \in \mathbf{T}$ **do**
4: $\quad \mathbf{t}.L = \infty$
5: $\quad \mathbf{t}.\mathbf{P}_{bst} = \emptyset$
6: **while TimeLeft**() **do**
7: $\quad$ **for** $\forall \mathbf{t} \in \mathbf{T}$ **do**
8: $\quad\quad$ **RandomTree**($\mathbf{t}$)
9: **Return** $(L, \mathbf{P}_{bst})$

RandomTree($\mathbf{t}$)
1: **if** $L < \mathbf{t}.L$ **then**
2: $\quad \mathbf{t}$.**AddPath**($\mathbf{P}_{bst}$)
3: $\quad \mathbf{t}.L = L$
4: $\quad \mathbf{t}$.**prune**(L)
5: $\quad \mathbf{t}$.**SetSampleBounds**(L)
6: **while TreeTimeLeft**() **and TimeLeft**() **do**
7: $\quad v = \mathbf{t}$.**RandomPoint**(L)
8: $\quad \mathbf{t}$.**Insert**(v)
9: $\quad$ **if** $\mathbf{t}.L < L$ **then**
10: $\quad\quad \mathbf{P}_{bst} = \mathbf{t}.\mathbf{P}_{bst}$
11: $\quad\quad L = \mathbf{t}.L$
12: $\quad\quad$ **Return**

Figure 3.5: Sequential C-FOREST (left), and individual tree (right). Note that any random-tree algorithm can be used, as long as it provides the necessary subroutines. Subroutines are described in Figure 3.2 in Section 3.1.

This is due to the fact that points in $\mathbf{P}_{bst}$ may increase the visibility of the receiving tree, and will be connected to the latter by construction. Lemma 3.2 shows that sharing points in $\mathbf{P}_{bst}$ can only increase the probability that the receiving tree finds a better solution. □

A consequence of Theorem 3.4 is that we expect $p_j \leq p_{\mathbf{t},j}$ if a C-FOREST $\mathbf{T}$ and a stand alone tree $\mathbf{t}'$ have the same L. Although this technically violates the assumption $p_j = p_{\mathbf{t},j} = p_{min,j}$ used in Theorem 3.2, it is not a problem due to the fact that using the inequality instead of the equality can only improve the performance of $\mathbf{T}$ vs. $\mathbf{t}'$. Thus, sharing $\mathbf{P}_{bst}$ should increase the advantage C-FOREST has over a single tree.

Given the analysis presented above, it is possible to predict the types of problems expected to exhibit the most speedup. Each new value of L significantly affects p and/or $p_{\mathbf{t}}$ via $\mathbf{A}_s$, and a particular path length reduction reduces $\|\mathbf{A}_s\|_\ell$ by an amount exponential in d. Since $\|\mathbf{A}_s\|_\ell$ correlates directly to random sampling and pruning, C-FOREST is expected to become more useful with increasing dimensionality d. Similarly, examining Figures 3.4 it is expected that harder search problems (e.g., those with smaller p) are more likely to exhibit super linear speedup.

3.3 Sequential C-FOREST

Given the potential for super linear speedup vs. T, I propose a sequential version of C-

FOREST that is allotted $1/T$-th of computation time on a single CPU for each tree $\mathbf{t} \in \mathbf{T}$. The algorithm is presented in Figure 3.5. The subroutine **TreeTimeLeft**() returns *true* if there is still time for tree $\mathbf{t}$ to plan during the current planning iteration. The amount of time allotted to each tree per iteration is small (e.g., on the order of 0.01 second), so that many loops through the forest occur over the course of the search. The rest of the subroutines are identical to those described in Figure 3.2 in Section 3.1.

The algorithm moves to the next tree as soon as the previous tree has found a solution, even if time still remains for the previous tree (**RandomTree**($\mathbf{t}$), line 12). I have found this to help during early phases of search, since it enables the next tree to focus a disproportional amount of effort on improving the current best solution. This effect is diminished once the forest is established. However, quickly reducing the search space at the beginning of the search has positive effects that propagate through the rest of the runtime.

The analysis on super linear speedup in Section 3.2 is also applicable to the sequential C-FOREST algorithm presented here. The main difference is that speedup and efficiency are equivalent for sequential C-FOREST, since only one CPU is used. Thus, while parallel C-FOREST may still offer a speed advantage when efficiency < 1, sequential C-FOREST will only be beneficial when efficiency > 1. Figure 3.4 shows the potential trade-offs between tree size n, forest size T, and speedup.

3.4 C-FOREST experiments

I perform three sets of experiments to evaluate the performance of C-FOREST of various sizes vs. a single random tree. The first experiment involves a seven degree-of-freedom manipulator arm. The second and third involve a multi-robot team in which all robots are viewed as pieces of a single robot. The sequential C-FOREST is tested on all three environments, and parallel C-FOREST is tested on the multi-robot problem using clusters containing $T = \{1, 2, 4, 8, 16, 32, 64\}$ computers. $T = 1$ is equivalent to planning with the underlying tree algorithm. The manipulator arm experiment uses C-FOREST composed of a search tree included in the OpenRAVE [33] environment

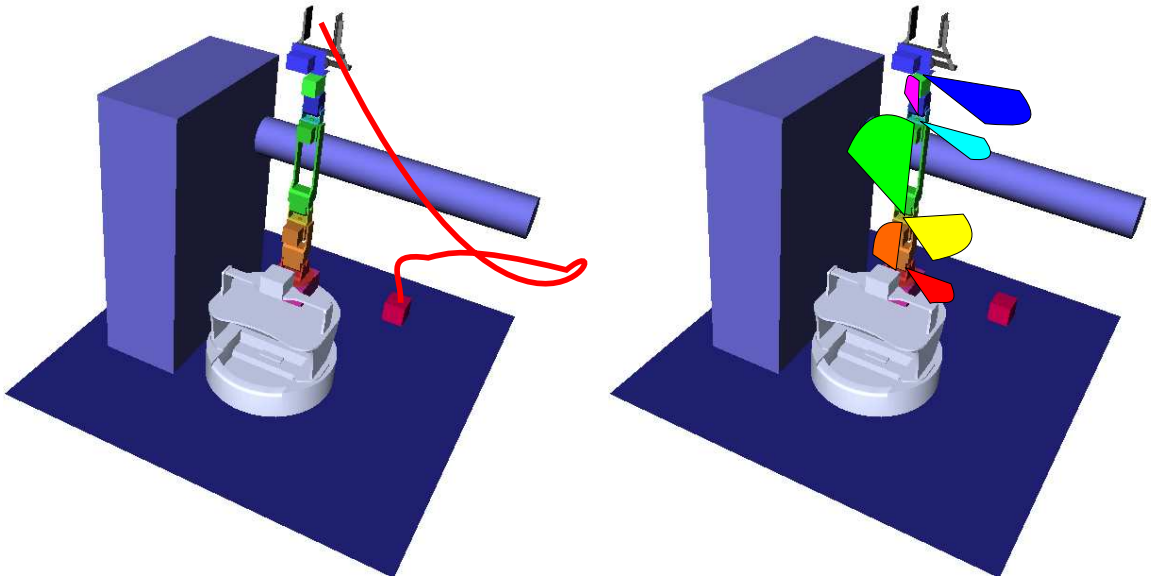


Figure 3.6: 7 DOF arm. The arm must grasp a block while avoiding the obstacles. The end-effector and summed swept area distance metrics (left and right, respectively).

that has been modified to perform a bi-directional SPRT, where SPRT is the random tree path planning algorithm described in Appendix B. The multi-robot experiments evaluates C-FORESTs composed of RRT* and SPRT trees. SPRT has a linear node insertion function, and RRT* [64] has a logarithmic insertion function in most cases. The parameter that defines the maximum distance between nodes in RRT* is set according to the analysis presented in appendix C. In short, the maximum distance is set to the diameter of the configuration space so that new nodes can rapidly spread the search-tree through the configuration space during early search stages. RRT* automatically shrinks this distance as more nodes are added to the tree to guarantee logarithmic node insertion runtime while maintaining convergence to the optimal solution.

3.4.1 Seven degree of freedom manipulator arm

A bi-directional planner in OpenRAVE [33] has been modified to use a sequential C-FOREST of SPRT trees instead of a single tree and test it on a 7-DOF manipulator arm (Figure 3.6). Each 'tree' corresponds to a particular pair of forward and backward SPRT trees. Two experiments are run, each using a different distance metric over the configuration space. The first defines solution

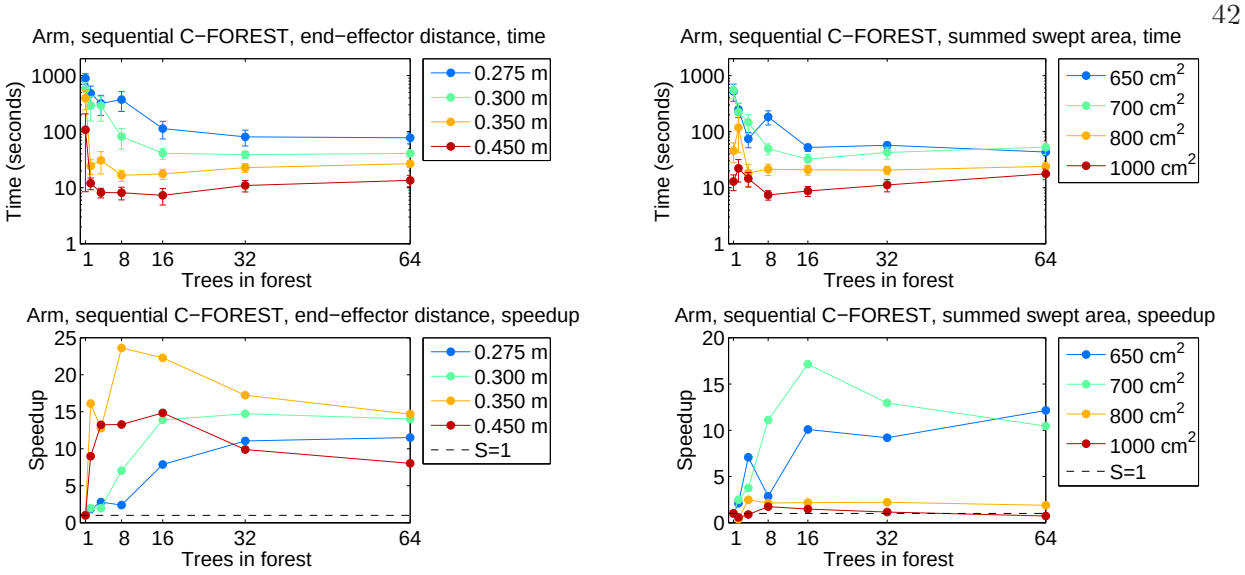


Figure 3.7: Sequential C-FOREST using bidirectional SPRT on a 7-DOF arm. Solution quality is the distance (m) traveled by the end effector or the summed swept area (cm^2) of arm segments (Left or Right, respectively). Color denotes solution quality. Top: Time to find a solution of a particular quality (mean and standard error over 20 trials)—note the log scale. Bottom: the resulting efficiency. Larger forests find better solutions more quickly. Many data-points have speedup > 1.

length as the total distance traveled by the end-effector of the arm. The second defines length as the sum of areas swept out by each arm section. The first metric minimizes the distance traveled by whatever the arm is holding, and the second minimizes the arm's interference with the rest of the environment. There are two main purposes of this experiment. The first is to demonstrate that existing hardware and software can be modified to use C-FOREST with relative ease—and that it is beneficial to do so. The second is to demonstrate that the benefits provided by C-FOREST are not dependent on a particular cost-metric.

Figure 3.7 displays how long it takes the forest to obtain solutions of varying lengths. Each data-point represents the mean and standard error over 20 trials. Left and right sub-figures correspond to using the first and second distance metrics, respectively. Top and Bottom display the average solution lengths and the speedup observed for C-FOREST of various size T.

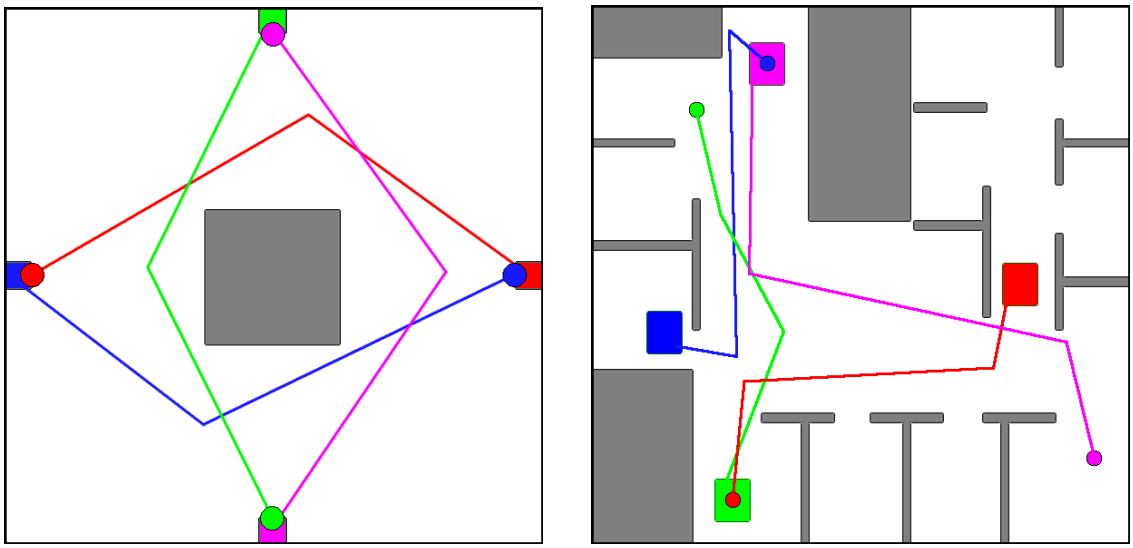


Figure 3.8: Toy and Office environments used in experiment groups 2 and 3 with sample paths. Goal regions appear rectangular and robot starting locations are circular.

3.4.2 Centralized multi-robot planning: toy problem

Four experiments are performed to evaluate the performance of C-FOREST on a simple multi-robot problem where four robots must change places around a central obstacle (Figure 3.8). The centralized multi-robot planning framework is used, so all robot are viewed as individual pieces of a single larger robot. The dimensionality of the configuration space is found by summing degrees-of-freedom of all individual robots. Four holonomic robots are used, and each adds two dimensions to the configuration space, for a total of eight degrees-of-freedom. Experiments are performed using sequential and parallel C-FORESTs compose of SPRT and RRT* trees. This experiment is designed to test the abilities of C-FOREST in a multi-robot problem where tree-growth is relatively unhindered by robot-obstacle collision checking. Obstacle collision checking may require significant computational power and may indirectly influence the runtime of the node insertion function (e.g., by causing more memory or disk reads). By minimizing collision checking I hope to push the runtime of the node insertion function toward its theoretical value. The experiment is also designed to demonstrate that C-FOREST can work on a variety of underlying random tree algorithms with different node insertion function run-times.

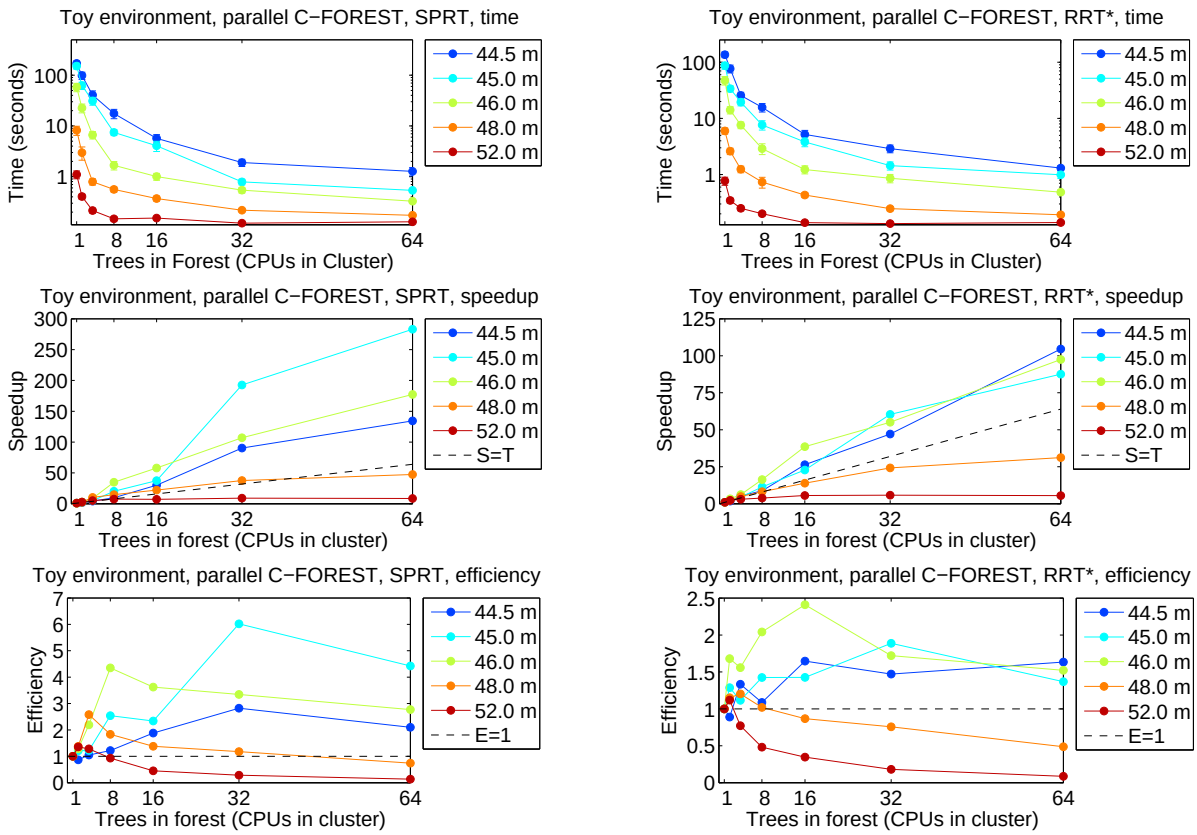


Figure 3.9: Parallel C-FOREST using SPRT or RRT* trees (Left and Right, respectively) on a toy multi-robot problem. Top: time required to find a solution of a particular length (mean and standard error over 32 trials)—note the log scale. Color denotes solution length (m). Center and Bottom: the resulting speedup and efficiency, respectively. Larger forests find better solutions more quickly. Better speedup and efficiency are observed on more difficult problems. Many data-points have efficiency > 1.

Figure 3.9 displays results for parallel C-FOREST. Points and bars represent mean and standard error over 32 trials, respectively. The left and right sub-figures of correspond to C-FOREST with SPRT and RRT* trees, respectively. The top sub-figures show how long it takes C-FOREST to obtain solutions of varying lengths in either environment, where length is the total length of the solution through the combined configuration space of all robots. Solutions with less length are more desirable and are expected to take longer to calculate. The center and bottom sub-figures display the mean speedup and parallelization efficiency that was observed, respectively.

Figure 3.10 displays results for sequential C-FOREST. Points and bars represent mean and standard error over 50 trials, respectively. The organization of Figure 3.10 is similar to that of

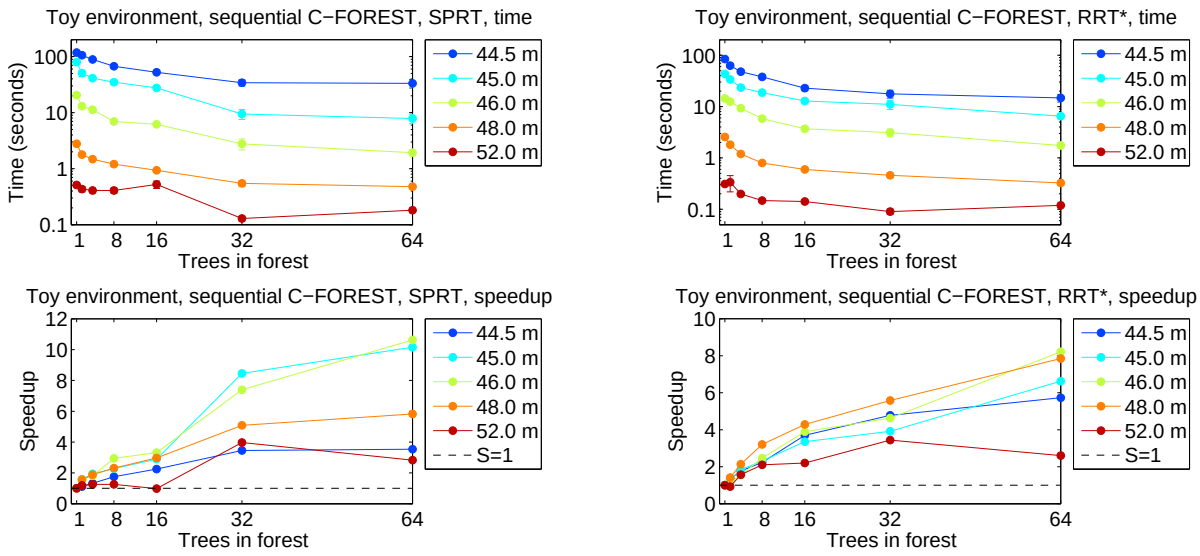


Figure 3.10: Sequential C-FOREST using SPRT or RRT* trees (Left and Right, respectively) on a toy multi-robot problem. Top: time required to find a solution of a particular length (mean and standard error over 50 trials)—note the log scale. Color denotes solution length (m). Bottom: the resulting speedup. Larger forests find better solutions more quickly. Better speedup is observed on more difficult problems. Many data-points have speedup > 1.

Figure 3.9, except that efficiency is omitted since only one CPU is used.

3.4.3 Centralized multi-robot planning: office environment

C-FOREST is evaluated on a multi-robot planning problem in the office environment provided by our lab (Figure 3.8). Four holonomic robots each contribute two dimensions to the configuration space, for a total of eight. Experiments are performed using sequential and parallel C-FORESTs composed of SPRT and RRT* trees. This experiment tests C-FOREST in a realistic environment that requires significant robot-obstacle collision checking. It is designed to show that the advantages of C-FOREST prevail also in realistic problems.

Figure 3.11 displays results for parallel C-FOREST. Points and bars represent mean and standard error over 32 trials, respectively. As in Figure 3.9 left and right sub-figures correspond to C-FORESTs of SPRT trees and RRT* trees, respectively. The top, center, and bottom sub-figures depict the time required to obtain solutions of varying quality, the mean observed speedup, and the mean observed parallelization efficiency, respectively.

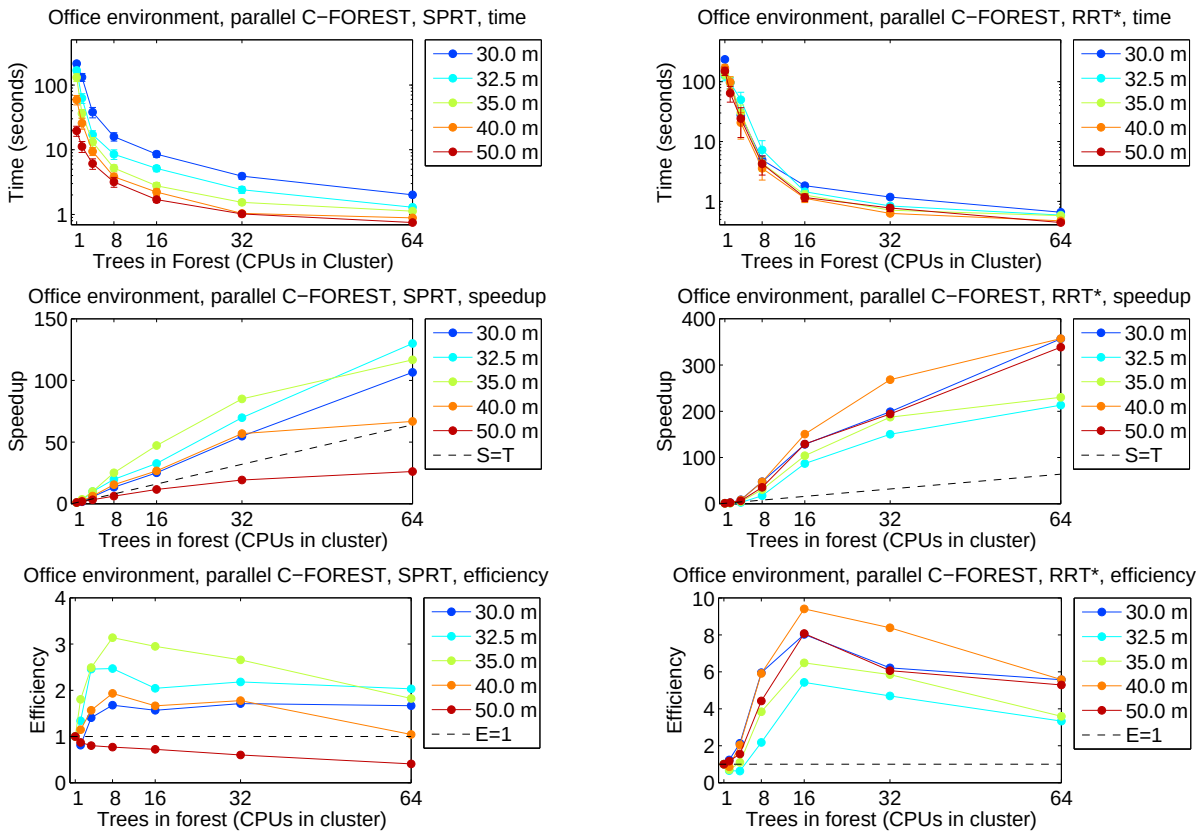


Figure 3.11: Parallel C-FOREST using SPRT or RRT* trees (Left and Right, respectively) on an office multi-robot problem. Top: time required to find a particular solution length (mean and standard error over 32 trials)—note the log scale. Color denotes solution length (m). Center and Bottom: the resulting speedup and efficiency, respectively. Larger forests find better solutions more quickly. Better speedup and efficiency are observed on more difficult problems. Many data-points have efficiency > 1.

Figure 3.12 displays results for sequential C-FOREST. Points and bars represent mean and standard error over 50 trials, respectively. The organization of Figure 3.12 is similar to that of Figure 3.10.

The theoretical analysis in Section 3.2 shows that speedup is a result of three things: (1) quicker node insertion via smaller individual tree size—due to decreased expected time to achieve a particular solution quality. (2) Increased probability a sampled point is useful—by focusing the sampling envelope. (3) Increasing probability a (useful) sampled point is connected to the tree—by increasing the visibility envelope of the tree via engrafting current best paths from one tree to another.

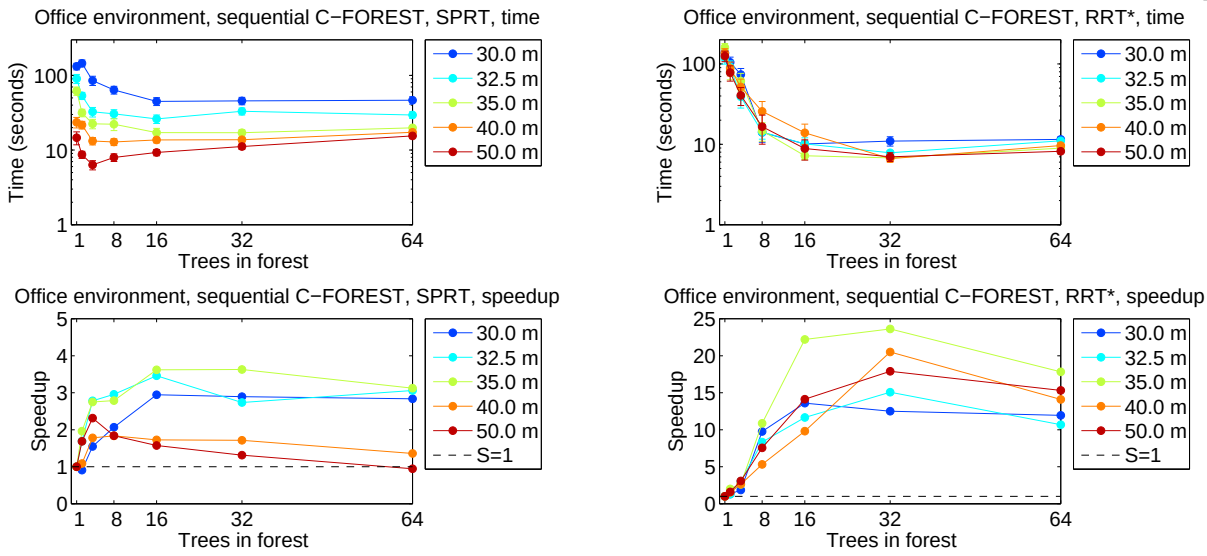


Figure 3.12: Sequential C-FOREST using SPRT or RRT* trees (Left and Right, respectively) on an office multi-robot problem. Top: time required to find a solution of a particular length (mean and standard error over 50 trials)—note the log scale. Color denotes solution length (m). Bottom: the resulting speedup. Larger forests find better solutions more quickly. Better speedup is observed on more difficult problems. Many data-points have speedup > 1.

All of these benefits are inherent in the results depicted in Figures 3.9-3.12. However, in order to assess the relative effects of each, the office environment experiment is rerun for solution quality 30 meters with modified versions of the algorithm that either do not decrease the sampling envelope size or do not share and engraft the current best solution (length L is still shared). Figures 3.13 and 3.14 depict the adjusted planning times and efficiencies for the parallel and sequential versions of C-FOREST, respectively.

3.5 Discussion of C-FOREST results

Overall, planning with C-FOREST works exceptionally well. The parallel C-FOREST algorithm gives significantly better results vs. a single tree—more trees correlate to better solutions, regardless of the type of tree being used. Super linear speedup is observed in all experiments (sometimes above 350), many efficiencies greater than 2, and some greater than 9. To the best of my knowledge, the greatest efficiency previously observed in parallel path planning is 1.2. Results are similarly positive for the sequential C-FOREST algorithm. This algorithm is only useful when

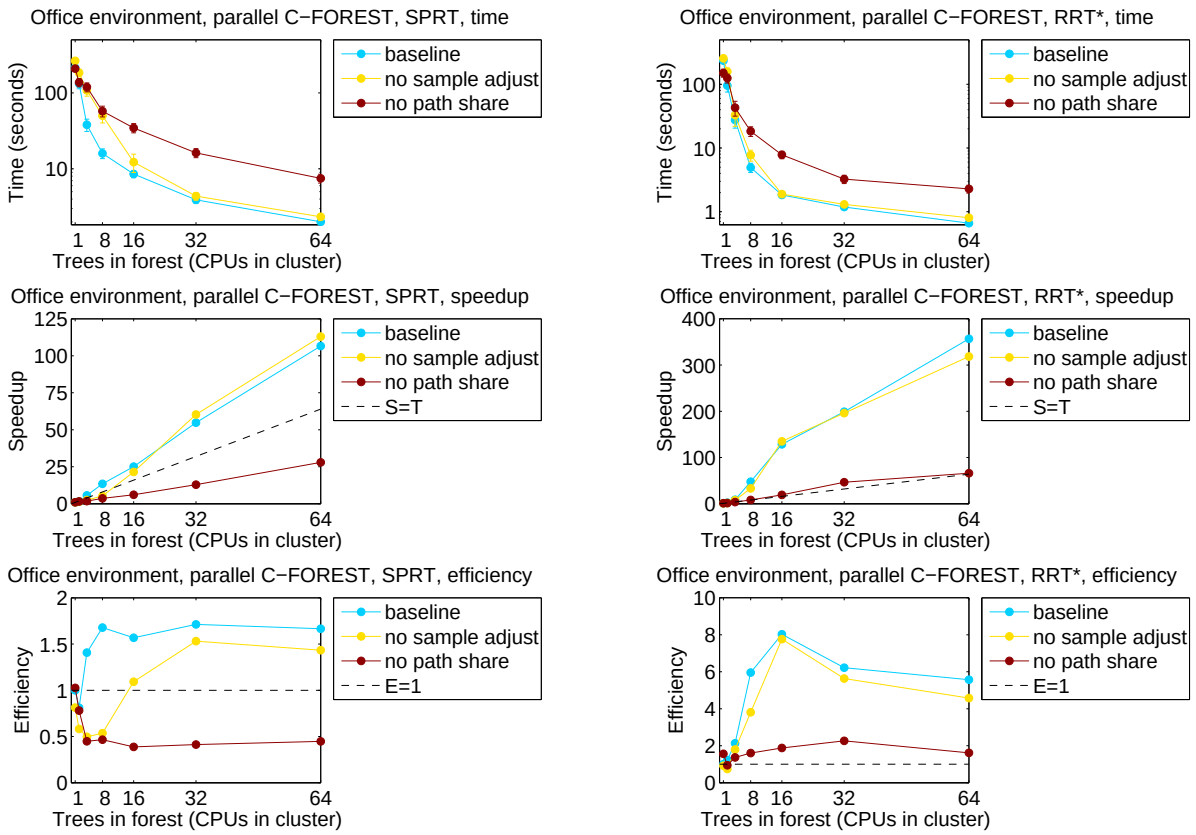


Figure 3.13: Variations of parallel C-FOREST using SPRT or RRT* trees (Left and Right, respectively), on an office multi-robot problem for solution length = 30m. Blue is normal parallel C-FOREST, yellow is a variant that does not decrease the sampling envelope, red is a version that does not engraft best solutions (solution lengths are still shared). Top: time required to find a solution (mean and standard error over 32 trials)—note the log scale. Center and Bottom: the resulting speedup and efficiency. Engrafting appears to be more important than decreasing the sampling envelope, especially for larger forests.

speedup is super linear vs. T; however, speedup greater than 2 is observed in all experiments, and sometimes greater than 20.

In general, the speedup observed using the single processor version is greater than what we might expect based on the efficiencies observed with a cluster. I believe this can be attributed to the fact that the sequential version of C-FOREST passes messages in memory while the cluster must use a network. It is also possible that the sequential algorithm's practice of moving on to the next tree as soon as a better solution is found provides additional advantage.

Speedup > 1 and efficiencies > 1 are observed for most experiments, solution lengths, and forest sizes. However, there are some data points for which efficiency and speedup are less than

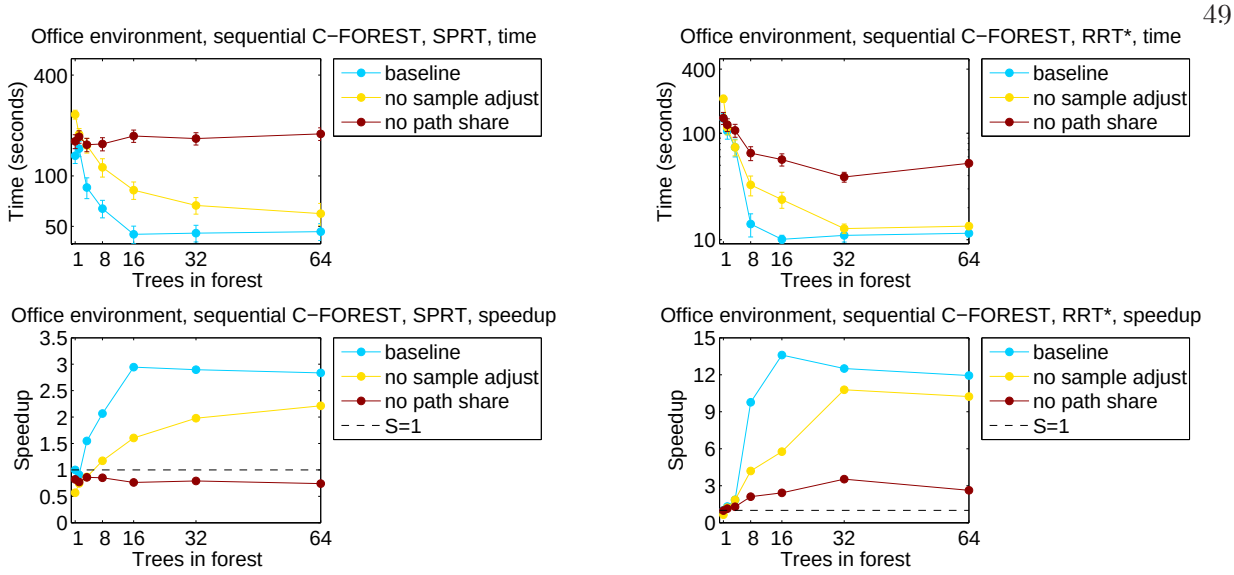


Figure 3.14: Variations of sequential C-FOREST using SPRT or RRT* trees (Left and Right, respectively), on an office multi-robot problem for solution length = 30m. Blue is normal parallel C-FOREST, yellow is a variant that does not decrease the sampling envelope, red is a version that does not engraft best solutions (solution lengths are still shared). Top: time required to find a solution (mean and standard error over 50 trials)—note the log scale. Bottom: the resulting speedup. Engrafting appears to be more important than decreasing the sampling envelope, especially for larger forests.

1 for both the parallel and sequential version of the algorithm. In general, data points for which results are not super linear correspond to problems that are relatively easy to solve. This result makes sense given Corollary 3.3 in Section 3.2 which indicates that a few Any-Time solutions need to be found in order to fully capitalize on super linearity. For easy problems, the first solution may be good enough, and so super linear speedup is not observed. Note however, that speedup > 1 is still observed for nearly all data points in all experiments. Therefore, with respect to solution time, it is still almost always in one's best interest to use parallel C-FOREST instead of a single tree.

For parallel C-FOREST (run on a cluster of T CPUs) the highest efficiencies were achieved using less than 64 computers in the particular experiments that I ran. Thus, with respect to power consumed per solution, there may be an inherent limit to the cost-savings C-FOREST can provide. Although the observed power savings of up to 89% (for efficiency 9.4) is quite decent. Experiments also show that using sequential C-FOREST can lead to power savings of over 95% (for speedup 23.6). While this highlights some of the better results, it is important to note that substantial

power savings were observed on most experiments (e.g., whenever efficiency > 1 for the parallel version of the algorithm, or speedup > 1 on the sequential version).

For most applications the most important metric is time required per solution. Assuming this evaluation criteria is used, the results show that using more trees is almost always better. In parallel C-FOREST, it was observed that mean solution times always decrease as more computers are added to the cluster—despite a falling, yet still greater than 1 efficiency. For sequential C-FOREST in the office environment, the benefits of adding more trees eventually plateau then gradually decline. In the simple environment the speedup is still increasing for sequential C-FOREST at 64 trees.

Algorithmically, it appears that sharing and engrafting nodes has a larger effect than reducing the size of the sampling envelope based on the length of the best solution. This can be seen in Figures 3.13 and 3.14, where forests that do not share nodes exhibit efficiencies slightly above or below 1 depending on if RRT* or SPRT trees are being used. That said, reducing the sampling envelope does have a noticeable effect in all experiments, with larger effects observed for forests containing few trees (e.g., those with $T \leq 16$).

A practical issue not yet addressed is how to select T in order to achieve the maximum efficiency (i.e., power savings) when using the parallel algorithm or to maximize speedup when using the sequential algorithm. However, it is clear that the optimal value of T for a particular problem is almost certainly greater than 1, assuming only one tree is constructed per CPU. Given the favorable efficiencies observed for higher values of T, it seems better to error on the side of using too many CPUs than too few. The high efficiency of the forest means that one can use many CPUs and still have a better power consumption per solution than what is observed using a single CPU. However, if power consumption is the most important metric, then an even better option appears to be using sequential C-FOREST on a single CPU.

This rationalization may be little consolation to those who desire a method of explicitly calculating the best T for a particular problem. To them I offer the following discussion: The analysis in Section 3.2 suggests C-FOREST works well because it allows multiple trees to share information about the sampling envelope and environmental visibility set. Therefore, it seems

reasonable that environments with similar properties should exhibit similar behavior vs. T. Here 'properties' may include things like the dimensionality of the configuration space, the average obstacle size and distribution, the ratio of the configuration space that is free of obstacles, and the type of robot being used. Experimental results suggest that performance seems to be stable vs. T that vary by an order of magnitude or more, and that it is better to error on the side of using too many trees than not enough. Therefore, calibrating T on a similar problem will likely give a decent approximation to its optimal value. Even if the optimal value of T cannot be estimated exactly, there appears to be ample room for error. For example, using an algorithm with 30% to many or too few trees has a relatively small effect on solution quality vs. time.

3.6 C-FOREST conclusions

I present a new approach to single-query path planning called *Coupled Forests Of Random Engrafting Search Trees* (or C-FOREST). In the C-FOREST algorithm multiple probabilistically independent random trees are coupled such that new random points are sampled, and existing nodes pruned, based on the best solution found by any tree in the forest. Current best solutions are also shared among and engrafted onto other trees in the forest. This allows good solutions to be improved by all trees in the forest and provides all trees increased visibility of the configuration space.

A significant contribution of this work is a theoretical proof showing that parallel C-FOREST (where each CPU grows a search tree), can achieve super linear speedup vs. the number of CPUs. In other words, C-FOREST has parallelization efficiency > 1. This is the first proof that super linear speedup in parallelized path planning can be attributed to algorithmic design, instead of more common hardware phenomena that are less scalable and harder to reproduce (e.g., better cache alignment). Super linear speedup is very rare. When it exists, it is possible to design faster non-distributed algorithms by dividing processing time between each sub-problem on a single CPU. Therefore, I also propose a non-distributed sequential version of C-FOREST that operates according to this concept.

Three sets of experiments are performed, in three different environments, using three different state-of-the art underlying random trees, three different distance metrics, and two different robotic systems. To the best of my knowledge, the parallel C-FOREST we test with RRT* is the first parallel implementation of RRT*.

Experiments show that parallel C-FOREST nearly always out-performs a stand-alone tree, often with super linear speedup. Indeed, to the best of our knowledge the efficiency obtained by C-FOREST is significantly larger than any previous result in parallel robotic path planning. Efficiencies over 2 are common and efficiencies up to 9.4 are observed (with speedups well over 350), while the best previous result I am aware of has efficiency 1.2. The speedup of sequential C-FOREST is also significant—often over 2 and up to 23.6.

My analysis suggests these positive results should be reproducible for C-FORESTs utilizing any type of random search tree that is expected to converge to optimality, and for any configuration space that obeys the triangle inequality. C-FOREST works especially well for difficult problems, where the probability of finding a better path is small.

Chapter 4

Any-Com C-FOREST: path planning with an ad-hoc distributed computer created over a networked robotic team

Complete solutions to multi-robot problems can be computationally complex. Although less expensive methods can enable practical performance in many real-world situations, these are incomplete and can fail in the most challenging circumstances (see Chapter 2). Often, each robot in a team is equipped with its own computer and the ability to communicate. Given these resources, it makes sense to divide computational effort among all robots a solution will benefit. That is, a networked team of robots can be re-cast as a distributed computer to solve the problems encountered by its composite robots. This is particularly useful for complex communal tasks such as centralized multi-robot path-planning. Specifically, this form of ad-hoc distributed computer can be used to run the C-FOREST algorithm described in Chapter 3, where the problem solved is the exact multi-robot path planning problem faced by the robotic team.

In practice, wireless bandwidth is environment dependent and often beyond the control of the user or a system. Yet, algorithms for coordinating networked robot systems usually rely on a minimum quality of service and fail otherwise. In contrast, I am interested in distributed algorithms that are able to utilize unreliable communication, and coin the term "Any-Com" to describe them. The idea is to find a suboptimal solution quickly, and then refine toward optimality as communication permits. This is analogous to the "Any-Time" paradigm, in which algorithms adapt to the available computation *time* [15]. Note that the C-FOREST algorithm is well suited to this idea given that it provides a distributed framework in which each CPU can make independent progress, but

successful communication between CPUs are beneficial—often resulting in super linear speedup.

In this chapter I present the necessary modifications required to run C-FOREST on an ad-hoc distributed computer formed from a robotic team. I call the revised algorithm *Any-Com C-FOREST*. Although Any-Com C-FOREST uses distributed computation it is algorithmically centralized due to the fact that all robots are viewed as single pieces of a composite robot. The ad-hoc distributed computer—acting as a single entity—calculates a solution. In previous work, centralized solutions have either been calculated on a single robot and then disseminated, or solved by each robot individually (see chapter 2). Note, I originally presented this idea in [91], where the algorithm was called Any-Com Intermediate Solution Sharing (Any-Com ISS).

In general, I believe that Any-Com algorithms should exploit perfect communication and have gracefully performance declines otherwise. However, just as Any-Time algorithms cannot calculate a solution in 0 time, Any-Com C-FOREST may not find a solution when communication totally fails. Worst-case scenarios aside, Any-Com C-FOREST is robust to a high degree of communication disruption.

4.1 Any-Com C-FOREST methodology

To utilize the distributed computational power of a team of mobile robots we need algorithms that function in environments where communication is unreliable, but we also desire that reliable communication can be exploited when it exists. The basic C-FOREST algorithm has strong inherent Any-Com properties. It is robust to moderate packet loss because dropped messages do not affect an agent's ability to eventually find a solution. Each agent maintains its own randomly created tree, and successful communication focuses search in beneficial ways and helps the team find better solutions more quickly. Robots share their individual intermediate solutions *during* path-planning so that all agents can prune globally sub-optimal branches from their local trees. This enables each robot to focus effort on finding only better solutions than those currently known to *any* robot. It also gives all robots a chance to directly refine the best intermediate solution. Therefore, even out-of-date messages have the potential to be beneficial, as long as the solution

they contain is better than the receiving agent's current best.

On the other hand, using C-FOREST for multi-robot path planning in a realistic setting requires additional considerations. In this chapter I focus on what is needed assuming that a team has already been established (this is extended in Chapter 5 to address team formation). If robots do not know other team member's starting and goal locations *a priori* then a start up communication exchange is required to share this data. Also, some mechanism is required to enable the team to reach a consensus on (which version of) the final solution to use, since multiple robots may find different improvements immediately before planning time is exhausted. Once a solution has been found and agreed upon, robots must coordinate movement along the final solution—while all team members should theoretically be able to follow the time schedule dictated by the solution, it is dangerous to assume that delays will never happen.

Practical Any-Com C-FOREST is achieved by adding synchronization mechanisms to the algorithm to enable consistency within the team with respect to initial data, final solution, and movement. Each message $\mathbf{m}$ contains the following data, based on the sending robot's current knowledge:

- $\mathbf{m}.\mathbf{D}$ A set containing a data field d_r about each robot r that the sender knows about, including the start $d_r.s$ and goal $d_r.g$ for each $d_r \in \mathbf{D}$.
- $\mathbf{m}.\mathbf{P}_{bst}$ Best solution (currently known to the sender).
- $\mathbf{m}.L_{bst}$ Best solution's length.
- $\mathbf{m}.r_{bst}$ ID of the robot that generated the best solution.
- $\mathbf{m}.\mathbf{V}$ List of robots that believe $\mathbf{m}.\mathbf{P}_{bst}$ is the best solution.
- $\mathbf{m}.\mathbf{F}$ List of robots that have submitted a final solution.
- $\mathbf{m}.M$ Movement flag that is true if the sender has started moving.
- $\mathbf{m}.B$ The amount of time that this robot is behind schedule (with respect to movement).

AnyComCFOREST()
1: $\mathbf{D} = \{d_r\}$
2: $L_{bst} = \infty$
3: $\mathbf{P}_{bst} = \emptyset$
4: $r_{bst} = \infty$
5: $M =$ **false**
6: $B = 0$
7: $\mathbf{F} = \emptyset$
8: **ListenerThread**()
9: **SenderThread**()
10: **while** $\exists\ r \in \mathbf{R}$ s.t. **NeedData**(r) **do**
11: **sleep**($1/\omega$)
12: **RandomTree**(**t**)
13: $\mathbf{F} = \mathbf{F} \cup \{r\}$
14: **AgreeAndMove**()

ListenerThread()
1: **while RobotsNotAtGoal**() **do**
2: **m** = **GetMessage**()
3: **if** $\exists\ r \in \mathbf{R}$ s.t. **NeedData**(r) **then**
4: **for all** r s.t. $d_r \in \mathbf{m}.\mathbf{D}$ **and** **NeedData**(r) **do**
5: $s_r = d_r.s$
6: $g_r = d_r.g$
7: $\mathbf{D} = \mathbf{D} \cup m.\mathbf{D}$
8: **else if** $\mathbf{m}.M$ **then**
9: $\mathbf{P}_{bst} = \mathbf{m}.\mathbf{P}_{bst}$
10: $L_{bst} = \mathbf{m}.L_{bst}$
11: $r_{bst} = \mathbf{m}.r$
12: $M =$ **true**
13: $\mathbf{V} = \mathbf{m}.\mathbf{V} \cup \{r\}$
14: $\mathbf{F} = \mathbf{m}.\mathbf{F} \cup \{r\}$
15: **else if TimeLeft**() **and** ($\mathbf{m}.L_{bst} < L_{bst}$ **or** ($\mathbf{m}.L_{bst} = L_{bst}$ **and** $\mathbf{m}.r \leq r_{bst}$)) **then**
16: **if** $\mathbf{m}.L_{bst} \neq L_{bst}$ **or** $\mathbf{m}.r \neq r_{bst}$ **then**
17: $\mathbf{V} = \mathbf{m}.\mathbf{V} \cup \{r\}$
18: $\mathbf{P}_{bst} = \mathbf{m}.\mathbf{P}_{bst}$
19: $L = \mathbf{m}.L_{bst}$
20: $r_{bst} = \mathbf{m}.r$
21: **t.Prune**(L_{bst})
22: **t.SetSampleBounds**(L_{bst})
23: **if not TimeLeft**() **then**
24: $\mathbf{F} = \mathbf{m}.\mathbf{F} \cup \mathbf{F} \cup \{r\}$
25: **if** $\mathbf{m}.L_{bst} = L_{bst}$ **and** $\mathbf{m}.r_{bst} = r_{bst}$ **then**
26: $\mathbf{V} = \mathbf{m}.\mathbf{V} \cup \mathbf{V}$

RandomTree(**t**)
1: **while TimeLeft**() **and not** M **do**
2: $v =$ **RandomPoint**(L_{bst})
3: **t.Insert**(v)
4: **if** $\mathbf{t}.L_{bst} < L_{bst}$ **and not** M **then**
5: $\mathbf{P}_{bst} = \mathbf{t}.\mathbf{P}_{bst}$
6: $L_{bst} = \mathbf{t}.L_{bst}$
7: $r_{bst} = \mathbf{t}.r$
8: $\mathbf{V} = \{r\}$
9: **t.Prune**(L_{bst})
10: **t.SetSampleBounds**(L_{bst})

SenderThread()
1: **while RobotsNotAtGoal**() **do**
2: $\mathbf{m}.\mathbf{D} = \mathbf{D}$
3: $\mathbf{m}.L_{bst} = L_{bst}$
4: $\mathbf{m}.\mathbf{P}_{bst} = \mathbf{P}_{bst}$
5: $\mathbf{m}.r_{bst} = r_{bst}$
6: $\mathbf{m}.\mathbf{V} = \mathbf{V}$
7: $\mathbf{m}.\mathbf{F} = \mathbf{F}$
8: $\mathbf{m}.M = M$
9: $\mathbf{m}.B = B$
10: **Broadcast**(**m**)
11: **sleep**($1/\omega$)

AgreeAndMove()
1: **while RobotsNotAtGoal**() **do**
2: **if RobotsAgree**() **then**
3: $B_r =$ time this robot is behind schedule
4: **if** $B_r < B$ **then**
5: stop moving along path
6: **else if** $B_r > B$ **then**
7: $B = B_r$
8: continue moving along path
9: **else**
10: continue moving along path

RobotsAgree()
1: **if** M **or** ($\forall\ r \in \mathbf{R}$, $r \in \mathbf{F}$) **or** ($r_{bst} = \mathbf{t}.r$ **and** $\forall\ r \in \mathbf{R}$, $r \in \mathbf{V}$) **then**
2: $M =$ **true**
3: **return true**
4: **return false**

Figure 4.1: Algorithm for Any-Com C-FOREST (Left-Top), and new subroutines. Note that **ListenerThread**() and **SenderThread**() are started on their own threads. **RobotsNotAtGoal**() returns *true* if not all robots are at the goal and *false* if all robots are at the goal. **m** = **GetMessage**() receives an incoming message and stores the data in **m**. **NeedData**(r) returns true if start/goal data has not been received from robot r. All other subroutines are identical to those in Figure 3.2 in Chapter 3, with the exception that **TimeLeft**() also returns false if M is true.

The practical implementation of Any-Com C-FOREST is displayed in Figure 4.1. **D** is initialized to contain this agent's data about its start and goal on line 1. The best solution is initialized to the empty set, its length to infinity, and the generating agent to infinity (lines 2-4). The movement flag is initialized to false (line 5), the amount this robot is behind schedule is initialized to 0 (line 6), and the set of agents that have submitted their final solution is initialized to the empty set (line 7). Separate threads are used to receive incoming messages (line 8) and send messages (line 9). Planning is handled according to the modified random tree algorithm but does not start until all start/goal data has been received (lines 10-12). After the allotted planning time has been exhausted, this agent adds itself to the list of agents that have submitted a final solution (line 13). Control is then passed to the function **AgreeAndMove**() on line 14.

Each robot keeps a copy of what it believes to be the best solution found by any robot, and each robot is responsible for adding itself to the lists **D**, **V**, and **F** (**AnyComCFOREST**() lines 1 and 13, **ListenerThread**() lines 13-14, 17, 24, and **RandomTree**(**t**) line 8). In order to keep the network up-to-date, messages are dropped if they contain paths that are worse than the best path known to the receiving agent (**ListenerThread**() line 15). Ties are broken by robot id r.

The sender thread broadcasts this robots world view at rate ω (**SenderThread**() lines 1-11). The listener thread is responsible for receiving messages from other robots (**SenderThread**() line 2), and then adjusting this robots world view accordingly. During the start-up data exchange phase the listener thread adds start and goal information about other robots to the combined start and goal states (lines 3-7). If a message contains data that indicates the sending robot is moving, then the message data automatically replaces the best solution to guarantee consistency within the team (lines 8-14). If a new best solution is received during the planning phase, then the global data are updated accordingly (lines 15-22). If there is not time left for planning, then the list of agents that have submitted their final solution is updated (lines 23-24). If the agent that sent the message agrees with this agent about the best solution, then the list of all agents that support the solution is updated accordingly (lines 25-26).

RobotsAgree() checks if an agreement has already occupied (line 1, first part). Otherwise,

any robot can correctly deduce when an agreement has occurred if it knows all robots have submitted a final solution (line 1, second part). This is because better solutions are no longer being generated and the best solution known to the sending robot is always sent in every message—the actual best solution must have been passed along with the knowledge that the robot who generated it has submitted a final solution. In the unlikely event of a tie, the solution found by the robot with the lower ID is used.

There is also an additional method of reaching an agreement using $\mathbf{V}$. By carefully tracking which solutions other robots support during the planning phase, it is possible to approximately forecast the final vote. A robot can conclude that its own solution should be used if it believes all robots currently support its most recent solution (line 1, third part). Although this may allow a suboptimal solution to be chosen, it is unlikely. Further, if an agent erroneously believes *all* robots currently support its solution, then it must have had the best solution in the past, so the cost of erroneously picking a suboptimal solution is mitigated. A scenario where different robots move along different *incompatible* solutions is impossible because two or more robots cannot simultaneously believe all robots support their most recent solution. This is due to the fact that *only* the robot that generated a solution can initiate movement along it. If two different robots generate competing solutions, neither will initiate movement until one robot advertises support for the other's solution—and they cannot both support the other's solution because one solution is guaranteed to be better than the other (or, in the case of ties, come from the robot with lower ID).

Once a robot knows an agreement has been reached, it sets M to true (**RobotsAgree**() line 2), and begins moving along its path per the best solution (**AgreeAndMove**() lines 2-10). Moving robots track keep track of the furtherest that any robot is running behind schedule, and then adjusts movement along their own paths accordingly (lines 3-10). Note that in practice this means that each robot adjusts the time it plans to be at a particular location by B.

There is an implicit assumption in the algorithm as it is depicted above that enough messages are successfully transmitted that agents start moving at similar times. In extremely harsh communication environments the algorithm may need to be additionally modified to include a syn-

chronized countdown timer to guarantee that all agents begin moving within an allowable time tolerance of each-other.

Theoretically, allowing more agents to work on a random-tree problem will increase the chances a good solution is found quickly, regardless of whether or not Any-Com C-FOREST is actually causing cooperation during the planning phase. To determine how much (if any) advantage Any-Com C-FOREST provides, it is compared to having *each* agent individually find a unique solution to the complete problem. Individual solutions are then broadcast so that the team can use the best one found by any angent. The latter method is referred to as *Voting*, and similar ideas have been explored in the past [29]. Finally, to give context to the relative performance of Any-Com C-FOREST vs. Voting, both are compared to a client-server framework. In the client-server system, called *Baseline*, the server is charged with calculating a complete solution using a single random tree, and then sharing it with the other robots. Any-Com C-FOREST, Voting, and Baseline use the same underlying random tree algorithm (Figure 3.2-Left). Performance is also compared to Any-Time RRT [43].

Both Any-Com C-FOREST and Voting use the same underlying message-passing protocol to disseminate information within the group. The idea is simple: each robot broadcasts information to every other robot at a predefined rate ω using the User Datagram Protocol (UDP). UDP drops unsuccessful messages, which keeps the information flowing through the network up-to-date. Baseline modifies this by having the server set M to true as soon as the planning time is over. Therefore, each robot begins moving as soon as the solution is received from the server. In order to keep Baseline as naive as possible, the client robots do not rebroadcast the solution to each other, but the server continues to rebroadcast at ω.

Each search-tree is generated randomly and each solution is drawn from a distribution over all possible solutions. Assuming $R = |\mathbf{R}|$ robots, the union of all trees is a $O(R)$ times larger tree maintained collectively by the entire team. Theoretically, both Any-Com C-FOREST and Voting should increase the team's collective chances of finding a desirable solution, vs. Baseline, because R random samples are drawn from this distribution instead of 1. I hypothesize Any-Com C-FOREST

will produce better solutions than the other two methods due to the distributed computational advantages it provides.

4.2 Any-Com C-FOREST experiments

Three experiments are performed with 5 robots in an office environment to evaluate C-FOREST vs. Voting and Baseline. The first experiment is conducted in simulation to evaluate theoretical performance over a wide range of parameters and also compares vs Any-Time RRT. The second experiment uses real robots to validate that the algorithms function in practice, but comparison vs Any-Time RRT is omitted due to the latter's poor performance on the first experiment. The third experiment is conducted on real robots in a deliberately challenging communication environment where imperfect Faraday cages are used to disrupt communication. The CU Prairiedog robotic platform is used for the real robot experiments (see Figure 4.2). It is built on top of the iRobot create base and uses the ROS operating system by Willow Garage. Robots are equipped with the Stargazer Indoor Localization System. The Computational Units are System 76 Netbooks with built-in wireless networking capabilities.

In all experiments C-FOREST uses the SPRT algorithm (presented in Appendix B) as the underlying random tree. I chose to use SPRT instead of RRT* because I desire solution quality $L_{bst} = \|\mathbf{P}_{bst}\|$ to be measured as either the sum of the distance along each robot's individual path through the workspace (to reflect the cumulative distance traveled), or as the maximum time it takes any robot to achieve the goal (to reflect the time that must pass until robots are free to begin a new task). Note that the latter metric is used for the experiments in this chapter. Proper use of either metric is not possible with RRT*, due to the fact that graph neighbors are selected with the help of a kd-tree that relies on the Euclidean distance through the team's combined configuration space.

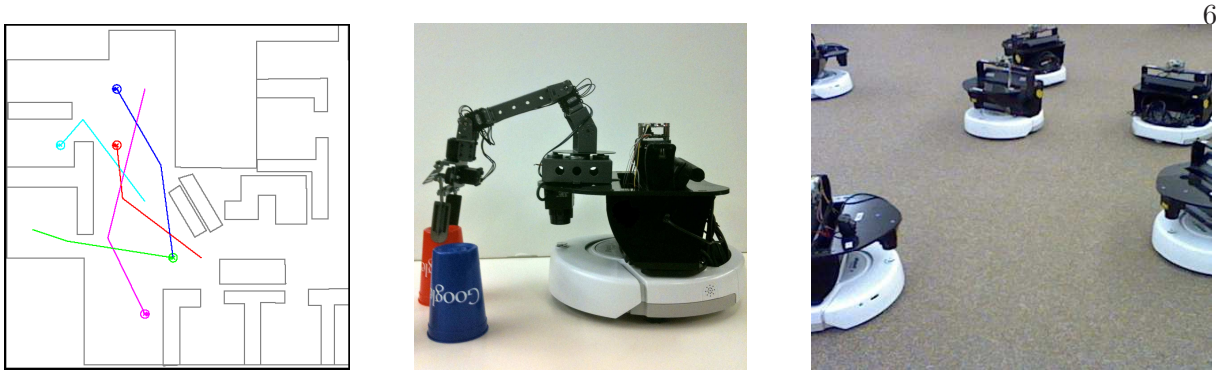


Figure 4.2: Left: 5 robots in an office environment and the resulting paths. Center and Right: Prairiedog platform. Each robot is equipped with an indoor localization system, a netbook and IEEE 802.11g wireless communication.

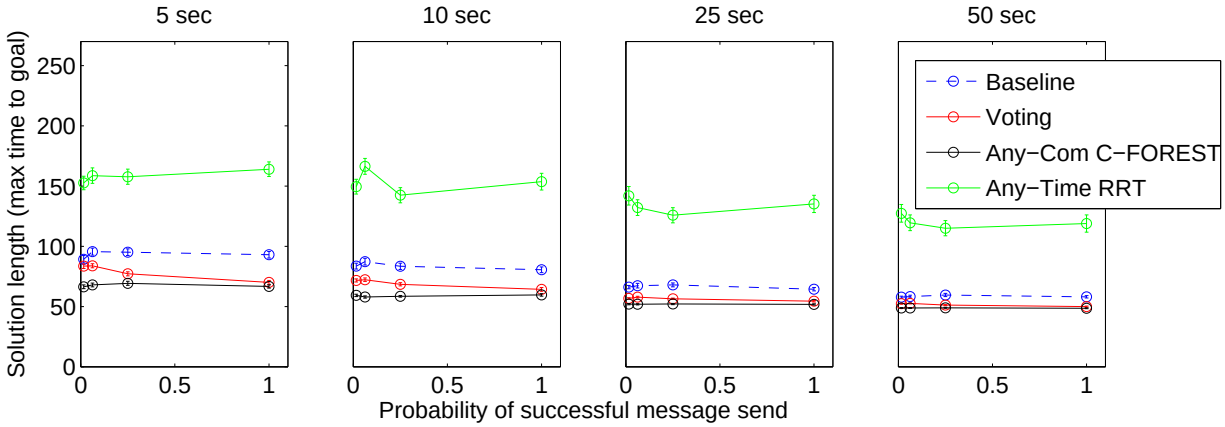


Figure 4.3: Solution lengths from the simulated office experiment. Sub-plots show different planning times μ. Mean and standard error over 100 runs are shown as points and error-bars respectively.

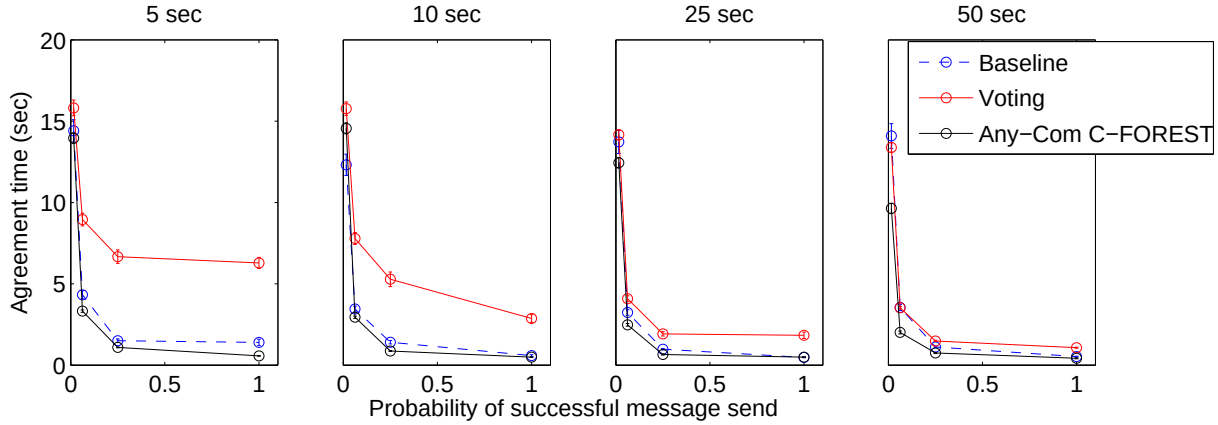


Figure 4.4: Agreement time from the simulated office experiment. Sub-plots show different planning times μ. Mean and standard error over 100 runs are shown as points and error-bars respectively.

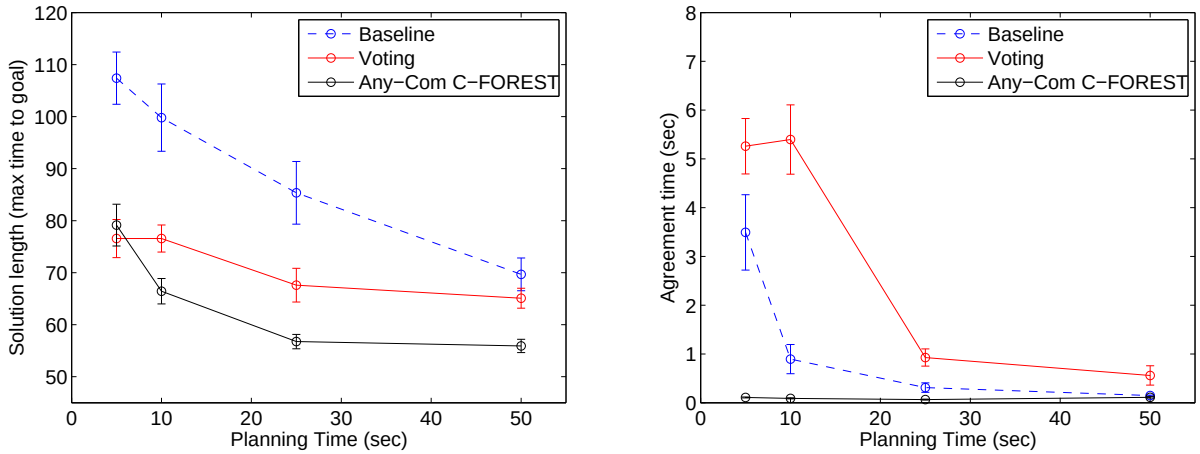


Figure 4.5: Solution lengths (Left) and average agreement time (Right) from the real robot office experiment. Mean and standard error over 20 runs represented as points and error-bars, respectively

4.2.1 Simulated office experiment

This experiment evaluates the relative performance of Any-Com C-FOREST, Voting, Baseline, and Any-Time RRT (Figure 4.2). Note that Any-Time RRT is run on a single robot. The performance of all four algorithms is evaluated vs. message success probability τ vs. planning time μ. In this experiment $\tau = \{1, 1/4, 1/16, 1/64\}$ probability of success and $\mu = \{5, 10, 25, 50\}$ seconds. 100 runs are performed per each combination of parameters to facilitate statistical analysis of results. Mean and standard errors of the resulting solution lengths are displayed in Figure 4.3 and agreement times in Figure 4.4 (agreement time is the time after μ and before movement). Both Any-Time RRT and Baseline are run on a single agent and have identical agreement phases. Therefore, the agreement times for Any-Time RRT are omitted.

4.2.2 Real robot office experiment

This experiment is conducted on 5 actual robots, but is otherwise similar to the simulated office experiment. Robot maximum speed is set to 0.2 meters per second. During planning $\omega = 4$, and during the agreement phase $\omega = 32$. The change is due to the preliminary results in Experiment 1, where it is clear that the agreement phase can become lengthy in terms of messages sent. Also,

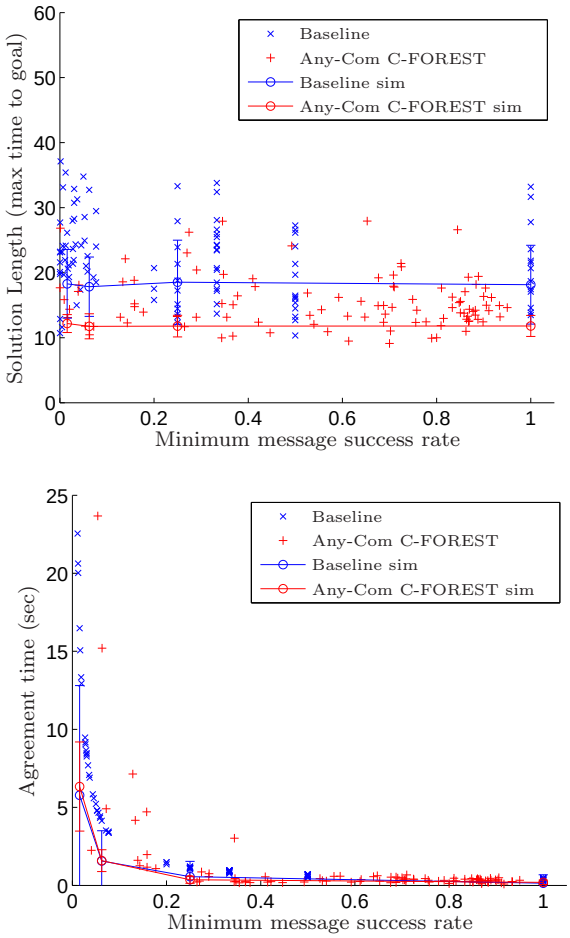


Figure 4.6: Faraday cage experiment results. Crosses and stars correspond to the effective, measured packet loss in the environment. Error-bars show means and standard deviations over 100 simulations assuming Poisson distributed package loss.

path-planning is computationally intensive while the agreement phase is not, and robots are able to spare additional resources to increase ω. The same μ are used as in Experiment 1. Each data-point represents 20 runs. We plot solution quality and agreement time vs. planning time in Figure 4.5 Left and Right, respectively. Signal quality was relatively good in this experiment, the observed packet loss rate was less than 50%. We forgo comparison vs. Any-Time RRT due to the positive performance of the other methods in the simulated experiment.

4.2.3 Faraday cage experiment

In this experiment five robots plan paths through an office environment while the wireless channel is systematically disturbed by shielding the RF system using an imperfect Faraday cage created by a tin-can. This is done in order to significantly disrupt communication—but not completely prohibit it. Statistics are collected on the actual solution quality and consensus time as a function of the effective packet throughput. Note that a different planning problem is solved than in the previous two experiments.

Results from the simulated experiment predict that Any-Com C-FOREST should outperform the other methods, even given packet-loss rates as high as 95%. However, the theoretical model used for simulation assumes that communication quality is Poisson distributed, and cannot account for all real-world communication disturbances. This experiment is designed to test Any-Com C-FOREST under harsh communication constraints in a real-world environment, and either validate or refute the earlier simulated predictions.

Experimental results are depicted in Figure 4.6, along with predictions from the theoretical simulations. The real data are depicted as crosses/stars. The theoretical data are shown as circles and error bars (representing the mean and standard deviation over 100 simulations, respectively, per a particular drop-rate). Note, the vertical appearance of the data-points at 1, 1/2, and 1/3 is an artifact of the low number of messages sent by the baseline approach and caused by experiments where solutions were successfully distributed on the first, second, and third, attempts, respectively.

4.3 Discussion of Any-Com C-FOREST results

With regard to solution quality, both Any-Com C-FOREST and Voting out-perform Baseline, and Any-Com C-FOREST outperforms Voting. All three methods outperform Any-Time RRT. Using a two-sample Kolmogorov-Smirnov test on results from the first two experiments, algorithms are compared based on solution lengths. Difference between algorithms are found to be statistically significant ($p < .05$) for all but one method-parameter combinations in the simulated experiment

(i.e. for one method vs. another with μ and τ held constant), and all but one parameter combination in the real robot experiment (Voting vs. Any-Com C-FOREST at $\mu = 5$ sec). In fact, $p < 0.001$ for most data-points, and when the results from the first two experiments are considered together, p becomes vanishingly small.

Performing a similar analysis for the Faraday cage experiment results makes less sense due to the high variability of τ. However, in the Faraday cage experiment, Any-Com C-FOREST out-performs the the other methods in terms of solution quality (i.e., path-time-to-goal). The three methods are similar with respect to mean agreement time, except that Any-Com has a lower standard deviation—which I attribute to early consensus building during the planning phase. Basic trends in the experimental results are similar to those predicted by the theoretical simulations. However, agreement times are greater than those predicted by the theoretical model. I believe this is due to the fact that wireless communication is not Poisson distributed in the real world. Results show that Any-Com C-FOREST can drastically speed up computation, even if packet loss is as high as 97%. In other words, the team is able to act as an effective distributed computer with only 3% of the messages getting through. These results validate my hypothesis that Any-Com C-FOREST will outperform the other two methods.

Examining the solution quality vs. planning time for the various methods in Figures 4.3 and 4.5-Left illustrate just how well Any-Com C-FOREST performs. Voting finds similar quality solutions using less than half the planning time as Baseline, on average, while Any-Com C-FOREST finds similar quality solutions in $\leq 1/n$ of the time. These results agree well with those from the basic C-FOREST algorithm in Chapter 3. They provide strong evidence that the robotic team is functioning as an effective ad-hoc distributed computer. Given the team is using n times as much computational power, the expected ratio of required planning time is $1/n$. Therefore, the value of $< 1/n$ that was observed in the real robot office experiment shows that the ad-hoc distributed computer is able to achieve parallelization efficiency > 1.

It often takes longer than 5 seconds (or even 10) for an agent to find a solution. That is, $\mu = 5$ is not enough time to guarantee that all robots have found a solution. In such a case, after 5

seconds has passed, Any-Com C-FOREST uses the best solution found by any robot so far, while Baseline must wait until the server finds its first solution, and Voting must wait until all robots have found a solution. This has two interesting affects. First, the agreement times of Baseline and Voting are greater than Any-Com C-FOREST because all robots must wait until the server or all robots have found a solution, respectively, before an agreement can be reached. Second, by waiting extra time until n solutions exits, Voting has an increased chance of finding a "good" solution vs. Any-Com C-FOREST. While this may initially seem desirable, Any-Com C-FOREST is able to start movement at the expected time, while the other algorithms suffer unexpected delays. We believe this is why the results for Voting and Any-Com C-FOREST are similar for $\mu = 5$ sec in the real robot office experiment (i.e. $p > .05$), and also why the agreement times for Voting are inflated for $\mu = 5$ and $\mu = 10$ in the simulated office experiment.

Another interesting trend is that Any-Com C-FOREST solution quality does not get much worse as communication becomes unreliable. Theoretically, as $\tau \to 0$ the results of Any-Com C-FOREST will approach those of Voting.[1] There is a hint of this in the simulated office experiment, where τ is controlled, especially for longer planning times. However, it appears communication must drastically deteriorate before Any-Com C-FOREST begins to suffer. In fact, packet loss rates as high as 98% have little affect on solution quality.

The most noticeable effect of poor communication is an increase in the time it takes the robots to agree on a single solution. Assuming that communication failure is strictly all-to-all and Poisson-distributed, increasing the messaging rate ω during the agreement phase can mitigate the effects of communication deterioration (as in the real world office experiment). In any case, the bandwidth will eventually become saturated, and further diminishing τ will eventually prevent an agreement from taking place within a useful time. Therefore, Any-Com C-FOREST should not be used when $\tau \approx 0$. That said, it is impossible for *any* complete algorithm to function when $\tau \approx 0$. As a practical measure, the $\tau \approx 0$ case could be handled using a time-out. After which, robots start moving based on the best solutions known to them individually. Assuming on-board sensors exist,

[1] They are also trivially the same at $\tau = 0$, since neither method will be able to reach a consensus.

conflicts would then be resolved using the cocktail-party model. Although this 'worst-case-scenario' forces the algorithm to become incomplete until communication is resumed, it is arguably better than letting the team remain motionless forever.

The simulated experiments predict Baseline should have similar agreement times to Any-Com C-FOREST, while the real experiments show Any-Com C-FOREST as the clear winner. The fact that these benefits do not extend to the Voting method (even for $\mu > 10$) suggests some other mechanism is responsible for the relatively quick agreement time of Any-Com C-FOREST. We credit this improvement to the auxiliary vote-forecasting agreement method available to Any-Com C-FOREST.

It is important to note that I have been using a communication protocol in which each robot sends messages directly to every other robot. While this works well for small numbers of robots, it may not transfer well to larger teams. In practice, any message protocol can be used that has the ability to propagate information to the entire team. However, other protocols may cause performance to change. In particular, I suspect that multi-hop protocols may decrease performance vs. a particular environmental communication state, since a particular message will be assaulted multiple times.

4.4 Any-Com C-FOREST conclusions

I coin the term "Any-Com" to describe algorithms that iteratively refine a solution toward optimality as communication permits. Any-Com algorithms are well suited for use on ad-hoc distributed computers, where communication between computational nodes may be unreliable. The motivation behind using Any-Com for multi-robot path planning is that a networked team of robots can be thought of as an ad-hoc distributed computer, and should adapt to use as much collaborative problem solving as communication quality permits. In general, the Any-Com idea is useful for solving computationally intensive problems, especially those with solutions of value to multiple agents. Centralized multi-robot navigation has both of these qualities.

I present the modifications necessary to turn C-FOREST into a practical Any-Com multi-

robot path-planning algorithm, and call the resulting algorithm *Any-Com C-FOREST*. The basic C-FOREST algorithm itself has strong Any-Com properties. Dropped messages do not prohibit a solution from eventually being found, while successful messages improve solution quality (both in overall path quality, and the time it takes to reach an agreement). Any-Com C-FOREST provides additional synchronization mechanisms that ensure the entire team uses the same solution and begins moving at the same time. It also explicitly uses a message passing protocol to share data between computational nodes (robots). I envision Any-Com C-FOREST as one tool among many in the multi-robot planning arsenal—useful in the specific case when a complete algorithm must be used (i.e., when a group of robots finds itself confronted with a difficult problem that cannot be solved by less expensive incomplete planning methods).

Three experiments are performed using a team of $n = 5$ robots, and results are compared to a basic server-client model as well as a voting method (in the server-client framework one agent plans and then distributes the solution to the other robots, while in voting each agent is allowed to plan separately and then the team uses the best solution found by any single agent). As with the basic C-FOREST algorithm, Any-Com C-FOREST can achieve super linear speedup vs. solving the same problem on a single computational node (robot). Any-Com C-FOREST is also at least twice efficient as the voting method assuming at least 2% message success rate.

As bandwidth approaches 0 the solution quality of Any-Com C-FOREST theoretically declines gracefully to that of the voting method, while both remain better than the server-client model. In fact, communication loss as high as 98% has little affect on solution quality. Unfortunately, the time it takes to reach consensus approaches infinity as communication approaches 0. This is not unexpected, as all complete multi-robot planning algorithms are inherently vulnerable to *total* communication failure. In an actual production implementation, total (or near-total) communication loss can easily be identified and used as a condition to fall-back to sensor-based cocktail party methods. Ignoring this worst-case-scenario, Any-Com C-FOREST is found to be robust to a high degree of communication interference.

While this chapter is a focused case-study on Any-Com applied to multi-robot navigation,

I believe that the Any-Com idea is not limited to the multi-robot path planning domain. I hope that the Any-Com concept will spread to other problems that can benefit from the computational power of ad-hoc distributed computing, and envision a world in which mobile robots dynamically take advantage all available computational resources to solve complex problems.

Chapter 5

Dynamic-Team Any-Com C-FOREST: centralized multi robot path planning with dynamic teams

This Chapter extends Any-Com C-FOREST by giving it the ability to use dynamic teams. I call the resulting algorithm *Dynamic-Team Any-Com C-FOREST*. The basic idea is that all robots start in their own team, and teams are combined only if doing so is necessary to guarantee safety and/or completeness. The idea is inspired by the twin observations that centralized multi-robot problem complexity is exponential in the number of robots in a team, and that many robots may not interfere with each-other's ability to navigate—despite operating near each other.

Any-Com C-FOREST and the dynamic team idea also complement each other for a number of other reasons. First, Any-Com C-FOREST assumes that robots working on the same problem are within communication range, and dynamic teams can be used to ensure this condition is met. Second, dynamic teams provide a graceful way to handle the discovery of previously unknown robots; once discovered, Any-Com C-FOREST provides a framework to check if they should be added to a team, and to harness their computational power if so. Third, and most importantly, they are both designed to attack the centralized multi-robot problem—albeit from different angles. By combining the two ideas, I hope to create a more resilient algorithm that inherits the positive aspects of both.

Previous work on dynamic teams has focused on the necessity of dynamic teams due to limited communication. In [28] team formation is seen as a positive event and allowed to occur as soon as two robots can communicate. This is because limited communication necessarily hinders team

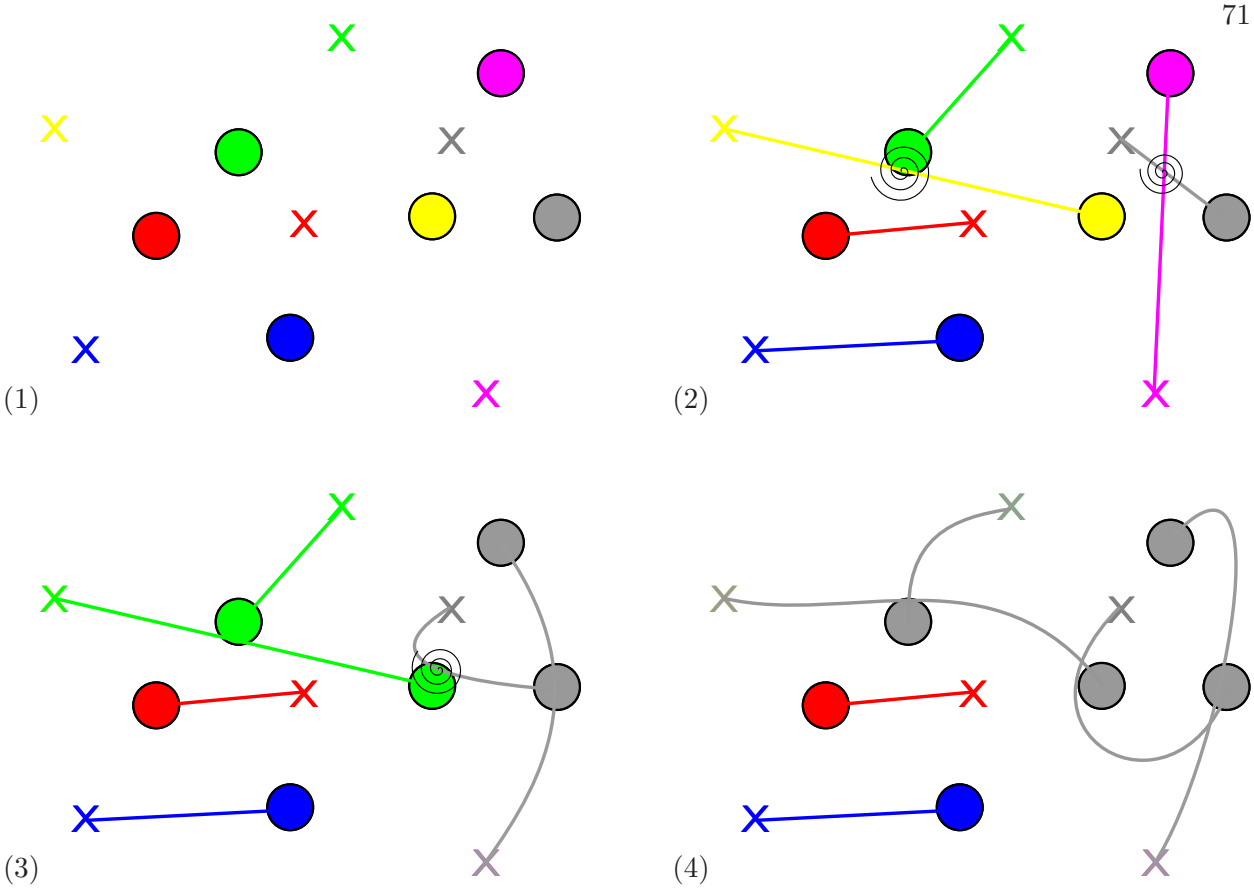


Figure 5.1: Dynamic-Team Any-Com C-FOREST (without selection of a conflict area). (1) Each robot starts in its own team. (2) Path conflicts cause pink and gray to form a team, and yellow and green to form a team. (3) New paths found by the green team conflict with new paths of the gray team, causing both teams to merge into a single team. (4) Total team formation includes two 1-robot teams and a 4-robot team.

formation and prevents information exchange. In contrast, I believe aggressive team formation overlooks the dynamic team framework's natural ability to break an $R \geq 2$ robot problem into two separate problems of size n and m, where $n + m = R$. Search complexity is dependent on the Lebesgue measure of the configuration space. If the Lebesgue measure of a configuration space containing a single robot is $\|\mathbf{C}_1\|_l$, then the Lebesgue measure of the multi-robot configuration space of R identical robot opperating in the same workspace is $\|\mathbf{C}_R\|_l = \|\mathbf{C}_1\|_l{}^R$. It is much easier for two teams to solve $O\left(\|\mathbf{C}_1\|_l{}^m\right)$ and $O\left(\|\mathbf{C}_1\|_l{}^n\right)$ complex problems in parallel, than for one large team to solve an $O\left(\|\mathbf{C}_1\|_l{}^{n+m}\right)$ problem. Therefore, I believe it makes sense to keep teams as small as possible, and only combine teams if doing so is necessary to avoid collisions and maintain

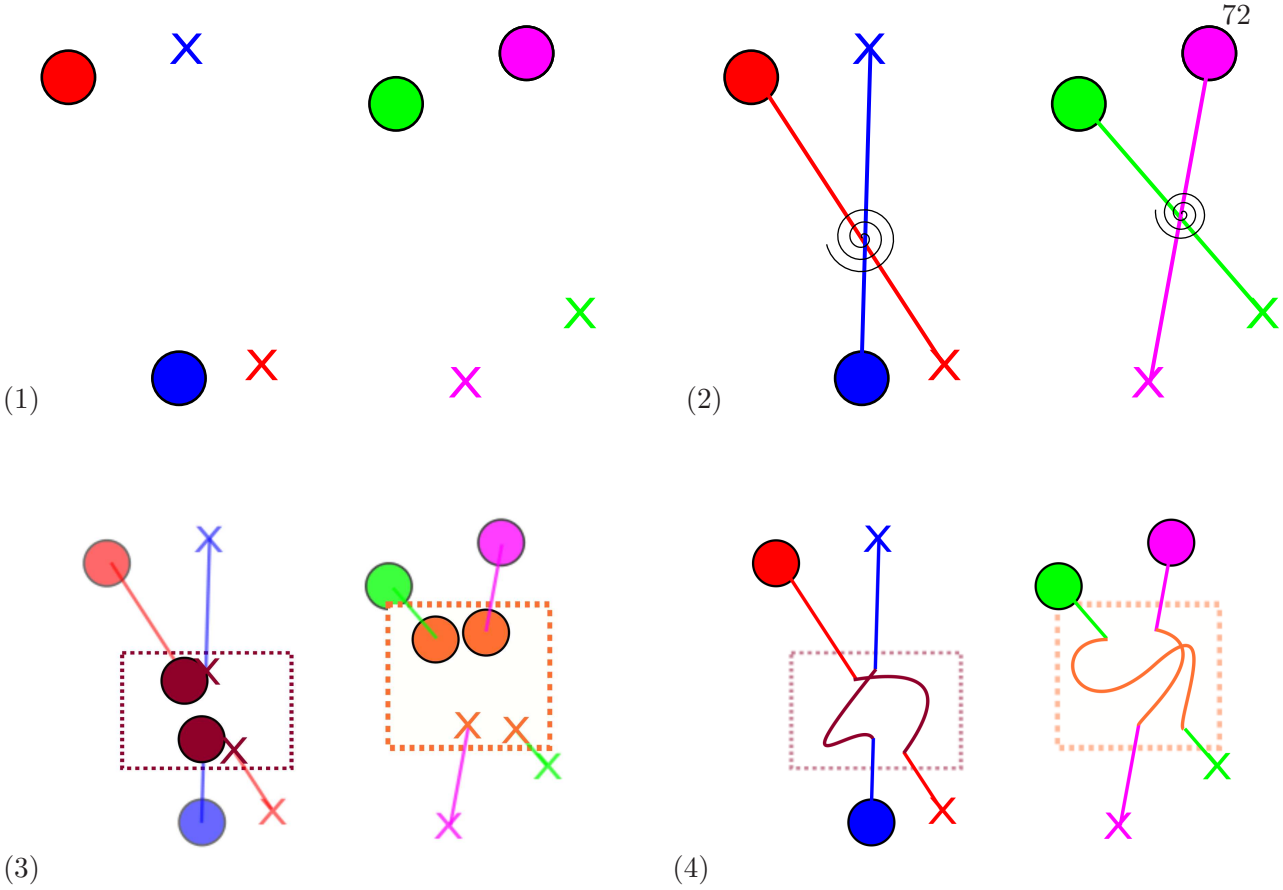


Figure 5.2: Dynamic-Team Any-Com C-FOREST (with selection of a conflict area). (1) Each robot starts in its own team. (2) Conflicts cause red and blue to form a team and pink and green to form a team. (3) Each team selects a small region around the conflict to re-plan in. (4) Each robot follows the team path within the conflict region, and its original path outside the conflict region.

completeness guarantees (See Figure 5.1).

Given that a single team's search complexity is $O(\|\mathbf{C}_1\|_l{}^{\Delta})$, where $\Delta = |\boldsymbol{\Delta}|$ is the number of robots in a particular dynamic team $\boldsymbol{\Delta}$, it is computationally advantageous to keep $\|\mathbf{C}_1\|_l$ small. Assuming that Δ is fixed, this means that team search should be confined to the smallest portion of the configuration space necessary to guarantee whatever algorithmic properties we are interested in (e.g., optimality, completeness, etc.)—see Figure 5.2. This presents an interesting design decision. Limiting search to a subset of the configuration space necessarily prevents some solutions from being found. This means that care must be taken to maintain guarantee on completeness, and that optimality is likely to be lost. On the other hand, while planning in the entire configuration space

may theoretically provide complete and optimal solutions, it may cause the search problem to be so difficult that a solution cannot be found within a practical amount of time.

My personal bias is to favor problem complexity considerations over optimality. The rational being that finding a quick suboptimal solution is more useful than an optimal (or less suboptimal) solution that takes longer to calculate. The assumption here is that the extra movement along a sub-optimal trajectory is worth the decrease in planning time caused by shrinking the size of the planning sub-space. There are two reasons I believe this is a valid assumption in situations where Dynamic-Team Any-Com C-FOREST is expected to be used. First, shrinking the diameter of the planning space causes an exponential decrease in problem complexity. Therefore, unless the environment is diabolically designed, the decrease in planning time is likely to be exponentially larger than the increase in path length. Second, long-term navigational solutions are likely to be invalidated by unforeseen future robot-robot conflicts and so replanning is likely. Thus, it does not make sense to spend a disproportionate amount of time converging toward an optimal solution when the majority of it may never be used.

Therefore, the version of Dynamic-Team Any-Com C-FOREST presented in this chapter seeks to minimize both the number of robots in a team, as well as the size of the configuration space that teams must plan through, while hopefully maintaining completeness guarantees. This is accomplished by combining teams only if their paths conflict (in both time and space), and then having the team to solve the combined problem of moving from one side of the conflict region to the other. For instance, if two robot's paths conflict, then the combined problem solved by the two robot team is limited to the area directly around the original conflict. Each robot can use its original path to move to and from the conflict region, but within the conflict region it must move according to the multi-robot solution. The selection of the conflict region is done in a way attempts to maintain completeness guarantees.

Indeed,"Dynamic-Team" plays the same role along the computation spectrum that "Any-Com" plays along the communication spectrum. Dynamic teams require a small amount of computational power to be useful, but become unnecessary given an infinite amount of computational

power. In practice, the selection of both team size and conflict region size that can be made dependent on the computational resources available to the resulting team.

5.1 Dynamic-Team Any-Com C-FOREST methodology

The major difference between Dynamic-Team Any-Com C-FOREST and Any-Com C-FOREST (presented in Chapter 4), is that the Dynamic Team version adds additional functionality to combine and dissolve teams, including the selection of which sub-set of the configuration space a team should plan through. As with Any-Com C-FOREST robots broadcast messages $\mathbf{m}$ that reflect the sender's current belief of the world. Messages are similar to those introduced in Section 4.1, except for the following modifications and additions:

- $\mathbf{m}.\boldsymbol{\Delta}$ is a list of all robots in the sender's team.
- d_r, the individual elements of $\mathbf{m}.\mathbf{D}$ are modified as follows: $d_r.s$ and $d_r.g$ are maintained to reflect the robots start and goal with respect is current sub-problem, and the additional fields $d_r.\mathbf{P}_{nav}$ and $d_r.\varepsilon$ are added. $d_r.\mathbf{P}_{nav}$ contains the entire path between the current location $d_r.s$ and the goal $d_r.g$ that robot r is currently using to navigate—including its time parameterization. $d_r.\varepsilon$ contains the current planning epoch of robot r. Note that $\mathbf{m}.\mathbf{D}$ still contains data about *all* robots that the sending agent knows about, not just those in the sender's team.

All other fields of $\mathbf{m}$ remain unchanged, except that they are defined with respect to the sender's team $\boldsymbol{\Delta}$ instead of the set of all robots $\mathbf{R}$.

Dynamic Team Any-Com C-FOREST is outlined in Figures 5.3 and additional subroutines are showed in 5.3. Initialization is performed on lines 1-5, where each robot inserts its own data into $\mathbf{D}$ and places itself into its own team $\boldsymbol{\Delta}$. The host robot also sets the path it advertises indicating its planned navigation to $\mathbf{P}_{nav}$ to be its start location. The planning iteration that the host robot advertises to other robots $\mathbf{D}.d_{\mathbf{t}.r}.\varepsilon$ is initialized to 0. The planning iteration ε is increased every time a new sub-problem is solved, and is used to ensure consistency within dynamic teams.

DynamicTeamAnyComCFOREST()

1: $\mathbf{D} = \{d_{\mathbf{t}.r}\}$
2: $\mathbf{\Delta} = \{\mathbf{t}.r\}$
3: $\mathbf{P}_{nav} = s_{\mathbf{t}.r}$
4: $\mathbf{D}.d_{\mathbf{t}.r}.\varepsilon = 0$
5: **Reset**()
6: **ListenerThread**()
7: **SenderThread**()
8: **loop**
9: **while** $\exists\ r \in \mathbf{\Delta}$ s.t. **NeedData**$(r, \varepsilon, \mathbf{D})$ **do**
10: **sleep**$(1/\omega)$
11: $[\mathbf{C}_{\mathbf{\Delta}}, v_{\not\mathbf{\Delta}}] =$ **PickCSpace**$(\mathbf{C}, \mathbf{\Delta}, \mathbf{D})$
12: **RandomTree**$(\mathbf{C}_{\mathbf{\Delta}}, \mathbf{t})$
13: **if NeedToReset then**
14: **continue**
15: $\mathbf{F} = \mathbf{F} \cup \{r\}$
16: $\mathbf{P}_{nav} =$ **PathCombine**$(\mathbf{P}_{nav}, \mathbf{C}_{\mathbf{\Delta}}, \mathbf{P}_{bst})$
17: **AgreeAndMove**()
18: **Reset**()

RandomTree$(\mathbf{C}_{\mathbf{\Delta}}, \mathbf{t})$

1: **while TimeLeft**() **and not** M **and not NeedToReset do**
2: $v =$ **RandomPoint**(L_{bst})
3: $\mathbf{t}$.**Insert**(v)
4: **if** $\mathbf{t}.L_{bst} < L_{bst}$ **and not** M **then**
5: $\mathbf{P}_{bst} = \mathbf{t}.\mathbf{P}_{bst}$
6: $L_{bst} = \mathbf{t}.L_{bst}$
7: $r_{bst} = \mathbf{t}.r$
8: $\mathbf{V} = \{r\}$
9: $\mathbf{t}$.**Prune**(L_{bst})
10: $\mathbf{t}$.**SetSampleBounds**(L_{bst})

SenderThread()

1: **loop**
2: $d_{\mathbf{t}.r}.\mathbf{P}_{nav} = \mathbf{P}_{nav}$
3: $d_{\mathbf{t}.r}.\varepsilon = \varepsilon$
4: $\mathbf{m}.\mathbf{D} = \mathbf{D}$
5: $\mathbf{m}.L_{bst} = L_{bst}$
6: $\mathbf{m}.\mathbf{P}_{bst} = \mathbf{P}_{bst}$
7: $\mathbf{m}.r_{bst} = r_{bst}$
8: $\mathbf{m}.\mathbf{V} = \mathbf{V}$
9: $\mathbf{m}.\mathbf{F} = \mathbf{F}$
10: $\mathbf{m}.M = M$
11: $\mathbf{m}.B = B$
12: **Broadcast**$(\mathbf{m})$
13: **sleep**$(1/\omega)$

ListenerThread()

1: **loop**
2: $\mathbf{m} =$ **GetMessage**()
3: **for all** r s.t. $(d_r \notin \mathbf{m}.\mathbf{D}) \vee (d_r \in \mathbf{m}.\mathbf{D} \wedge \mathbf{m}.\mathbf{D}.d_r.\varepsilon \geq \mathbf{D}.d_r.\varepsilon)$ **do**
4: $\mathbf{D}.d_r = \mathbf{m}.\mathbf{D}.d_r$
5: **if** $\mathbf{m}.r \notin \mathbf{\Delta}$ **and** $(\exists\ r_1, r_2$ s.t. $r_1 \in \mathbf{m}.\mathbf{\Delta} \wedge r_2 \in \mathbf{\Delta} \wedge$ **Conflict**$(\mathbf{m}.\mathbf{D}.d_{r_1}, \mathbf{D}.d_{r_2}))$ **then**
6: $\mathbf{\Delta} = \mathbf{\Delta} \cup \mathbf{m}.\mathbf{\Delta}$
7: $\varepsilon =$ **max**$(\varepsilon, \mathbf{m}.\mathbf{D}.d_{r_1}.\varepsilon) + 1$
8: **NeedToReset = true**
9: **else if** $(\mathbf{m}.r \in \mathbf{\Delta}$ **and** $\mathbf{m}.\mathbf{D}.d_r.\varepsilon > \varepsilon)$ **or** $(\mathbf{m}.r \notin \mathbf{\Delta}$ **and** $\exists\ \mathbf{t}.r \in \mathbf{m}.\mathbf{\Delta})$ **then**
10: $\mathbf{\Delta} = \mathbf{\Delta} \cup \mathbf{m}.\mathbf{\Delta}$
11: $\varepsilon = \mathbf{m}.\mathbf{D}.d_r.\varepsilon$
12: **NeedToReset = true**
13: **else if** $\mathbf{m}.r \in \mathbf{\Delta}$ **and** $\mathbf{m}.\mathbf{D}.d_r.\varepsilon = \varepsilon$ **then**
14: **if** $\mathbf{m}.M$ **then**
15: $\mathbf{P}_{bst} = \mathbf{m}.\mathbf{P}_{bst}$
16: $L_{bst} = \mathbf{m}.L_{bst}$
17: $r_{bst} = \mathbf{m}.r$
18: $M =$ **true**
19: $\mathbf{V} = \mathbf{m}.\mathbf{V} \cup \{r\}$
20: $\mathbf{F} = \mathbf{m}.\mathbf{F} \cup \{r\}$
21: **else if TimeLeft**() **and** $(\mathbf{m}.L_{bst} < L_{bst}$ **or** $(\mathbf{m}.L_{bst} = L_{bst}$ **and** $\mathbf{m}.r \leq r_{bst}))$ **then**
22: **if** $\mathbf{m}.L_{bst} \neq L_{bst}$ **or** $\mathbf{m}.r \neq r_{bst}$ **then**
23: $\mathbf{V} = \mathbf{m}.\mathbf{V} \cup \{r\}$
24: $\mathbf{P}_{bst} = \mathbf{m}.\mathbf{P}_{bst}$
25: $L = \mathbf{m}.L_{bst}$
26: $r_{bst} = \mathbf{m}.r$
27: $\mathbf{t}$.**Prune**(L_{bst})
28: $\mathbf{t}$.**SetSampleBounds**(L_{bst})
29: **if not TimeLeft**() **then**
30: $\mathbf{F} = \mathbf{m}.\mathbf{F} \cup \mathbf{F} \cup \{r\}$
31: **if** $\mathbf{m}.L_{bst} = L_{bst}$ **and** $\mathbf{m}.r_{bst} = r_{bst}$ **then**
32: $\mathbf{V} = \mathbf{m}.\mathbf{V} \cup \mathbf{V}$

Reset()

1: $L_{bst} = \infty$
2: $\mathbf{P}_{bst} = \emptyset$
3: $r_{bst} = \infty$
4: $M =$ **false**
5: $B = 0$
6: $\mathbf{F} = \emptyset$
7: $\varepsilon = \mathbf{D}.d_{\mathbf{t}.r}.\varepsilon$
8: $v_{\not\mathbf{\Delta}} = \emptyset$
9: **NeedToReset = false**

Figure 5.3: Algorithm for Dynamic-Team Any-Com C-FOREST (Left-Top), and new/modified subroutines. Other subroutines appear in Figure 5.4 or are identical to those in Figure 4.1 in Chapter 4.

AgreeAndMove()
1: **while not NeedToReset do**
2: **if RobotsAgree() then**
3: B_r = time this robot is behind schedule
4: **if** $B_r < B$ **then**
5: stop moving along path
6: **else if** $B_r > B$ **then**
7: $B = B_r$
8: continue moving along path
9: **else**
10: continue moving along path
11: **if RobotAt**($v_{\blacktriangle}$) **then**
12: $\mathbf{\Delta} = \{\mathbf{t}.r\}$
13: stop moving along path

RobotsAgree()
1: **if** M **or** ($\forall\ r \in \mathbf{\Delta}$, $r \in \mathbf{F}$) **or**
($r_{bst} = \mathbf{t}.r$ **and** $\forall\ r \in \mathbf{\Delta}$, $r \in \mathbf{V}$) **then**
2: M = **true**
3: **return true**
4: **return false**

NeedData($r, \varepsilon, \mathbf{D}$)
1: **if** $d_r \notin \mathbf{D}$ **then**
2: **return true**
3: **else if** $\mathbf{D}.d_r.\varepsilon < \varepsilon$ **then**
4: **return true**

$[\mathbf{C}_{\mathbf{\Delta}}, v_{\blacktriangle}]$ = **PickCSpace**($\mathbf{C}, \mathbf{\Delta}, \mathbf{D}$)
1: $\mathbf{D}_\Delta = \bigcup_{r \in \Delta} d_r$
2: $\mathbf{K} = \bigcup_{r \in \Delta}$ **FindFirstConflict**($d_r, \mathbf{D}_\Delta \setminus d_r$)
3: **if** $\exists\ r \in \mathbf{\Delta}$ s.t. $K_r \notin \mathbf{K}$ **then**
4: v_χ = **Centroid**($\mathbf{K}$)
5: **for all** r s.t. $r \in \Delta \wedge K_r \notin \mathbf{K}$ **do**
6: $\mathbf{K} = \mathbf{K} \cup$ **ClosestPoint**($d_r.\mathbf{P}_{nav}, v_\chi$)
7: $\mathbf{C}_{\mathbf{\Delta}}$ = **SubspaceContaining**($\mathbf{C}, \mathbf{K}$)
8: **while** $\mathbf{C}_{\mathbf{\Delta}} \neq \mathbf{C}$ **and not Safe**($\mathbf{C}_{\mathbf{\Delta}}, \Delta$) **do**
9: $\mathbf{C}_{\mathbf{\Delta}}$ = **Expand**($\mathbf{C}_{\mathbf{\Delta}}, \mathbf{C}$)
10: **for all** $r \in \mathbf{\Delta}$ **do**
11: s_r = first point along $\mathbf{D}.d_r.\mathbf{P}_{nav}$ in $\mathbf{C}_{\mathbf{\Delta}}$
12: g_r = last point along $\mathbf{D}.d_r.\mathbf{P}_{nav}$ in $\mathbf{C}_{\mathbf{\Delta}}$
13: $v_{\blacktriangle} = g_{\mathbf{t}.r}$

Figure 5.4: New/modified subroutines for Dynamic-Team Any-Com C-FOREST. Other subroutines appear in Figure 5.3 or are identical to those in Figure 4.1 in Chapter 4.

Additional initialization happens within the **Reset**() function, which is also called each time the robot begins to solve a new subproblem. The best path (for the current problem) $\mathbf{P}_{bst}$, its length L_{bst}, and ID of its generating robot are initialized (**Reset**() lines 1-3), as well as the movement flag M, the amount the robot is behind schedule B, the planning iteration ε, the location at which the robot can dissolve its team $v_{\blacktriangle}$, and **NeedToReset** which is a master flag that is used to indicate that a new subproblem must be solved (lines 4-9).

Back in the main Dynamic-Team Any-Com C-FOREST algorithm, the listener and sender threads are started on lines 6 and 7, and the main loop begins on line 8, and will continue to run forever (Due to the fact that the total number of robots is unknown, and new robots may require the host robot to move away from its goal after it has been reached. The main loop gets restarted every time a new sub-problem must be solved (e.g., when a new robot joins the team). As in Any-Com C-FOREST, the robot waits until it receives the most up-to-data data for each robot in its team (lines 9-10).

After the robot knows all desired navigational paths of its team ($\mathbf{D}.d_r.\mathbf{P}_{nav}$ for each $r \in \mathbf{\Delta}$), it uses this information to calculate which subset of the environment should be used for planning $\mathbf{C}_{\mathbf{\Delta}}$ on line 11. Note that all robots must use the same deterministic algorithm to find $\mathbf{C}_{\mathbf{\Delta}}$ in order to guarantee C-FOREST compatibility. Actual planning takes place on line 12. Assuming a solution is found before the sub-problem changes then the robot adds itself to the list of teammates that have submitted a final solution line 15. The robot combines the new solution $\mathbf{P}_{bst}$ with its old navigational path $\mathbf{P}_{nav}$—using the former within $\mathbf{C}_{\mathbf{\Delta}}$ and the latter outside of $\mathbf{C}_{\mathbf{\Delta}}$, and updating the time parametrization accordingly. Finally, the robot enters the agree and move phase (lines 16-17).

In practice I have found it advantageous to set the time associated with the first point on $\mathbf{P}_{nav}$ to be 0, and then updating $\mathbf{P}_{bst}$ accordingly. While this causes points on $\mathbf{P}_{bst}$ that are located before $\mathbf{P}_{nav}$ to be negative, each robot immediately updates B to reflect the time that it expects it will arrive at the beginning of $\mathbf{P}_{nav}$, and so other robots adjust accordingly.

The subroutine **RandomTree**($\mathbf{C}_{\mathbf{\Delta}}, \mathbf{t}$) is similar to what was used in Any-Com C-FOREST, except that it will return early if **NeedToReset** is set to true. **AgreeAndMove**() is modified in a similar fashion. The latter also checks if the robot reaches $v_{\not\mathbf{\Delta}}$, and if so then dissolves the team (lines 11-12). **RobotsAgree**() is also similar to its Any-Com C-FOREST version, except that only robots within the dynamic team are considered. **SenderTread**() is modified to populate $\mathbf{m}$ with the additional fields required by Dynamic-Team Any-Com, and **NeedData**($r, \varepsilon, \mathbf{D}$) is modified to additionally check for planning iteration consistency.

A large portion of the new functionality required to use dynamic teams takes place within **ListenerTread**(). Lines 3-4 keep $\mathbf{D}$ up to date (in addition to start and goal data, $\mathbf{D}$ contains the path that each robot is using for navigation and their current planning iteration). Lines 5-12 check for different cases that will cause team size to be increased. Line 5 checks if any robots in the sender's team conflict with any robots in the host robot's team. If so then the host robot set its planning iteration to be larger than the maximum iteration involved in the combination, adds all robots from the sender's team to its own team, and then sets **NeedToReset** to true (lines 6-8).

Line 9 checks if there is either a within-team planning iteration incrementation or if an out-of-team robot has added the host-robot to its team. In either case, the host-robot combines teams with the other robot, adopts the other robot's planning iteration, and then sets **NeedToReset** to true (lines 10-12). If a message comes from a within-team robot that is using the same planning iteration (line 13), then behavior is identical to the previous version of the algorithm presented in Section 4.1 (lines 14-32).

Selection of $\mathbf{C}_{\mathbf{\Delta}}$ the subspace of $\mathbf{C}$ team $\mathbf{\Delta}$ agrees to use for planning is accomplished via the $[\mathbf{C}_{\mathbf{\Delta}}, v_{\not\mathbf{\Delta}}] = \mathbf{PickCSpace}(\mathbf{C}, \mathbf{\Delta}, \mathbf{D})$ function. For simplicity, all team data is accumulated in the list $\mathbf{D}_{\Delta}$ on line 1. Next, K_r the first conflict vs. time of each robot $r \in \mathbf{\Delta}$ is found and accumulated in the list $\mathbf{K}$ on line 2. If some robots do not have conflicts vs. any other robot in the team then some method of including them in the search space must be devised (this can happen due to the fact that robots in the same team may have already resolved their conflicts during a previous planning iteration). Although this could be handled in many different ways (e.g., removing non-conflicting robots from the team), I have chosen to include all robots in the conflicting teams. The rationalization for this is that since all robots in $\mathbf{\Delta}$ were previously coordinating movement with at least one currently conflicting robot, it is likely that new solutions may result in another conflict with the same robot. Therefore, if robot r does not have any conflicts, then its conflict point is defined to be the closest point on $\mathbf{D}_{\mathbf{\Delta}}.d_r.\mathbf{P}_{nav}$ to the centroid of all other conflict points v_χ, lines 3-6. The closest point is chosen in hopes of minimizing the size of $\mathbf{C}_{\mathbf{\Delta}}$.

$\mathbf{C}_{\mathbf{\Delta}}$ is initialized to the smallest subspace of $\mathbf{C}$ that contains all points in $\mathbf{K}$, line 7. Next, $\mathbf{C}_{\mathbf{\Delta}}$ is increased until either we can be sure that completeness guarantees are maintained via the $\mathbf{Safe}(\mathbf{C}_{\mathbf{\Delta}}, \Delta)$ function, or $\mathbf{C}_{\mathbf{\Delta}}$ contains the entire space $\mathbf{C}$, lines 8-9. I will postpone discussion about $\mathbf{Safe}(\mathbf{C}_{\mathbf{\Delta}}, \Delta)$ until the next section. Finally, the sub-area start and goal coordinates are saved, lines 11-12, and the point at which the host robot can dissolve its team is also recorded, line 13.

5.1.1 Maintaining completeness

Maintaining completeness while using a subset of the original configuration space for planning is a tricky problem. I believe that in the worst case the general problem of selecting the smallest $\mathbf{C}_{\mathbf{\Delta}}$ that maintains completeness guarantees is at least as hard as planning through the entire $\mathbf{C}$, due to the fact that, in the worst case, it is impossible to find a solution even when $\mathbf{C}_{\mathbf{\Delta}} = \mathbf{C}$—and verifying this requires solving the planning problem using the entire $\mathbf{C}$.

The logic for using $\mathbf{C}_{\mathbf{\Delta}}$ is that problem complexity is reduced when $\mathbf{C}_{\mathbf{\Delta}} \subsetneq \mathbf{C}$. Therefore, any useful algorithm for selecting $\mathbf{C}_{\mathbf{\Delta}}$ should run fast compared to solving the path planning problem through $\mathbf{C}_{\mathbf{\Delta}}$. A strategy that I have found to work well in practice, and the one that I use in $\mathbf{Safe}(\mathbf{C}_{\mathbf{\Delta}}, \Delta)$, is to use a set of fast and greedy heuristic-based checks that will never falsely report that completeness is maintained when it is not, but *may* erroneously report that completeness is not maintained when it actually is. This is why the size of $\mathbf{C}_{\mathbf{\Delta}}$ is increased until we can be sure that completeness is maintained.

I now prove a few theorems showing cases where completeness is guaranteed to be maintained. I assume a 2D workspace with circular holonomic robots that have the ability to stop in place. Similar proofs exist for robots that do not meet these conditions, and can be achieved by replacing the robotic footprint with the maximal space required to slow to a halt, turn around, or otherwise maintain movement required for safety (i.e., circling in place to maintain altitude in an aircraft). Similar proofs also exist for 3D workspaces. Let the maximum diameter of the robot footprint be denoted ϕ. I assume that all robot use the same portion of the workspace to define their portion of the configuration space. In other words, $\forall r_1, r_2 \in \mathbf{\Delta}$ s.t. $r_1 \neq r_2$ it is the case that $\mathbf{C}_{r_1} = \mathbf{C}_{r_2}$. Recall that $\mathbf{C}_{\mathbf{\Delta}} = \bigcup_{r \in \mathbf{\Delta}} \mathbf{C}_r$. Let $\mathbf{W}_r$ be the portion of the workspace that corresponds to $\mathbf{C}_r$. Given the previous assumptions, $\mathbf{W}_r$ is the same for all $r \in \mathbf{\Delta}$, so let $\mathbf{W}_{\mathbf{\Delta}} = \mathbf{W}_{r \in \mathbf{\Delta}}$ to highlight the fact that we are considering the same subset of the workspace for each member of the entire team. I also assume that all start and goal locations are valid (i.e. there are no robot-robot conflicts before planning begins or at goal location, and start/goal locations are safely within the planning

area), and that each robot has a unique ID number.

Lemma 5.1: *Assuming Δ robots are organized along the top row of a Δ by Δ grid-matrix, where each grid has side-length ϕ, it is possible to reorder the robots in any permutation without collisions.*

Proof. The proof is by construction, and there are three cases $\Delta = 1$, $\Delta = 2$, and $\Delta \geq 3$. Case 1 is trivially true. Case 2 has two sub-cases: In sub-case 2.1 the robots are already in the correct position, which is trivially true. In sub-case 2.2 the robots need to be swapped, which is possible by moving the right-most robot down, then the left most robot over into the correct position, and the finally the first robot left and then up into the correct position. For Case 3 an algorithm to reorder the robots is as follows: let robots be identified by their left to right order in the desired permutation. Place the robot with the highest ID in the correct location by swapping it with the robot in the desired position (this is possible since one robot can move right along the second row of the matrix, while the other can move left along the third row of the matrix). Repeat with the second highest ID, third highest ID, etc. □

Corollary 5.1: *Assuming Δ robots are organized along the top row of a 3 by Δ grid-matrix, where each grid has side-length ϕ, it is possible to reorder the robots in any permutation without collisions.*

Proof. This is immediately evident given that the $\Delta = 1$ and $\Delta = 2$ cases are guaranteed by Lemma 5.1, and the case $\Delta \geq 3$ can be solved by the same algorithm used in Case 3 of Lemma 5.1. □

Lemma 5.2: *Assuming Δ robots are located in a 2 by Δ grid-matrix, where each grid has side-length ϕ, it is possible to place all robots in the top row without collisions.*

Proof. The proof is by construction. Let the grids in the top row be numbered $1, 2, ..., \Delta$. An algorithm to place all robots along the top row is as follows: let the robot closest to grid-1 be called robot-1, ties are broken by selecting the left most robot. Of the remaining robots, let the robot closest to grid 2 be called robot-2. Repeat this process for the rest of the grids and robots. Next, simultaneously move all robots toward their corresponding grids at the same speed until all robots

either cannot move without entering a collision state or have reached their desired location. Note that at least one robot must now be closer to its corresponding grid due to the fact that there is too much space ($\phi\Delta$ across) for traffic jams to prevent all Δ robots from moving at all. The entire process is repeated until all robots are organized along the top row. Note that this is analogous to placing marbles in a container and turning it up-side-down so all the marbles fall toward the ground. □

Corollary 5.2: *Assuming Δ robots are located in a 3 by Δ grid-matrix, where each grid has side-length ϕ, it is possible to place all robots in the top row without collisions.*

Proof. This is immediately evident given Lemma 5.2 and the fact that allowing more space to maneuver will not hurt the ability of robots to reach the goal. □

Corollary 5.3: *Assuming Δ robots are located in a Δ by Δ grid-matrix, where each grid has side-length ϕ, it is possible to place all robots in the top row without collisions.*

Proof. There are two cases trivial cases, $\Delta = 1$ and $\Delta \geq 2$, the first it trivially true and the second is immediately evident given Lemma 5.2 and the fact that allowing more space to maneuver will not hurt the ability of robots to reach the goal. □

Corollary 5.4: *Assuming Δ robots are located in the top row of a 2 by Δ grid-matrix, where each grid has side-length ϕ, it is possible to move the robots to any safe configuration within that space without collisions.*

Proof. This is immediately evident given that the inverse problem can be solved by Lemma 5.2, and then the robot paths reversed. □

Corollary 5.5: *Assuming Δ robots are located in the top row of a 3 by Δ grid-matrix, where each grid has side-length ϕ, it is possible to move the robots to any safe configuration within that space without collisions.*

Proof. This is immediately evident given that the inverse problem can be solved by Corollary 5.2, and then the robot paths reversed. □

Corollary 5.6: *Assuming Δ robots are located in the top row of a Δ by Δ grid-matrix, where each grid has side-length ϕ, it is possible to move the robots to any safe configuration within that space without collisions.*

Proof. This is immediately evident given that the inverse problem can be solved by Corollary 5.3, and then the robot paths reversed. □

Theorem 5.1: *If no obstacles are present then completeness guarantees are maintained if $\mathbf{W_\Delta}$ is defined by a rectangle with side-lengths 3Δ by $\phi\Delta$.*

Proof. Such a rectangle contains a 3 by Δ grid where each grid has side-length ϕ. Regardless of initial configuration, it is possible to move all robots so they are located along the upper row of the grid , reorder them appropriately (corollary 5.1), and then move all robots from that position to any other position within the square (corollary 5.5). □

Theorem 5.2: *If no obstacles are present then completeness guarantees are maintained if $\mathbf{W_\Delta}$ is defined by a square with side-length $\phi\Delta$.*

Proof. Such a square contains a Δ by Δ grid where each grid has side-length ϕ. Regardless of initial configuration, it is possible to move all robots so they are located along the upper row of the grid (corollary 5.3), reorder them appropriately (Lemma 5.1), and then move all robots from that position to any other position within the square (corollary 5.6). □

Theorem 5.3: *If obstacles are present then completeness guarantees are maintained if $\mathbf{W_\Delta}$ contains a free-space patch defined by a rectangle with side-lengths 3Δ by $\phi\Delta$ that is accessible to all robots.*

Proof. All robots can get to a free-space rectangle where they can rearrange themselves in any configurations. This allows the robots to order themselves appropriately so that robots that must leave the rectangle do so in the appropriate order. □

Theorem 5.4: *If obstacles are present then completeness guarantees are maintained if* $\mathbf{W}_{\mathbf{\Delta}}$ *contains a free-space patch defined by a square with side-length* $\phi\Delta$ *that is accessible to all robots.*

Proof. All robots can get to a free-space square where they can rearrange themselves in any configurations. This allows the robots to order themselves so that they can leave the square in the appropriate order. □

For algorithmic purposes the square case is only more useful than the the rectangle case when $\Delta \leq 2$. Additional theorems can be proved that guarantee completeness is maintained given even less stringent requirements on free-space. I will now give a rough sketch of two ideas. The first idea requires all robots to have access to a 2Δ by $\phi\Delta$ free-space rectangle plus another 2Δ by 2Δ free-space square (robots go to the rectangle, then take turns using the 2Δ square to swap locations, then move to their final locations). Another requires all robots to have access to $\Delta + 1$ free-space squares of size ϕ by ϕ that robots can occupy without blocking other robots from getting to or from any of these squares (this is essentially the same as the previous proof sketch, except that the rectangle used to store robots during the swapping phase has been chopped into Δ pieces that are dispersed throughout the environment and the final square is used to help swap robots). While these ideas are clearly more general than the proofs given above, they are also more computationally complex to automate—and I do not use them in practice.

5.1.2 Modifications

There are a number of reasons to delay team formation as long as possible. For instance, the world is uncertain and communication range is limited. Just because robots appear to conflict at some future time does not necessarily mean that they will—other (sooner) conflicts may change a robots plans. For instance, assume two teams $\mathbf{\Delta}_A$ and $\mathbf{\Delta}_B$ are in conflict. There may exist another robot/team $\mathbf{\Delta}_C \not\subset (\mathbf{\Delta}_A \cup \mathbf{\Delta}_B)$ initially beyond communication range, but that eventually joins $\mathbf{\Delta}_A$ or $\mathbf{\Delta}_B$ and forces them to re-plan. Thus, the original "conflicting" solution of $\mathbf{\Delta}_A$ or $\mathbf{\Delta}_B$ may either become invalid or cease to be in conflict before either team actually reaches the original

conflict point.

Since complete multi-robot navigation is exponentially complex in the number of robots, keeping teams small is a priority and it is advisable to delay team formation until collisions appear unavoidable (e.g., if $\mathbf{\Delta}_A$ and $\mathbf{\Delta}_B$ are in conflict *and* the two conflicting robots are physically near each other, then the chances of collision are much greater). Another consideration is that, although Any-Com C-FOREST is theoretically robust to poor communication, it can benefit from improved communication (and has problems when communication totally fails). Assuming communication is generally better at closer range, forming teams such that teammates are near each other also reduces the chance of total communication failure. Therefore, $\mathbf{Conflict}(d_{r_1}, d_{r_2})$ delays team formation by returning false until a robot r_1 is closer than a predefined threshold to r_2.

5.2 Dynamic-Team Any-Com C-FOREST experiments

All experiments in this chapter are performed using use the Prairiedog robotic platform (see Figure 4.2). Robots run the ROS operating system, localize using the Hagisonic Stargazer, and are equipped with a map of the environment. Robots exchange data using IEEE 802.11g wireless in ad-hoc mode. The target speed is set to 0.2 meters per second. As in Chapter 4 C-FOREST uses the SPRT algorithm (presented in Appendix B) as the underlying random tree. Solution length is measured as the sum of all individual path lengths.

Five experiments are performed. The first three compare Dynamic-Team Any Com C-FOREST with and without using sub-area selection for determining $\mathbf{C_{Delta}}$, respectively. The rest of the experiments are designed to compare the performance of Dynamic-Team Any-Com C-FOREST vs. Any-Com C-FOREST.

5.2.1 Large Andrews Hall experiment

This experiment evaluates the benefits of planning in $\mathbf{C_\Delta} \subsetneq \mathbf{C}$ vs. planning in the entire space $\mathbf{C_\Delta} = \mathbf{C}$. Four robots are deployed in Andrews Hall, a large building on the University of Colorado at Boulder campus with a challenging floor-plan that requires team formation (see

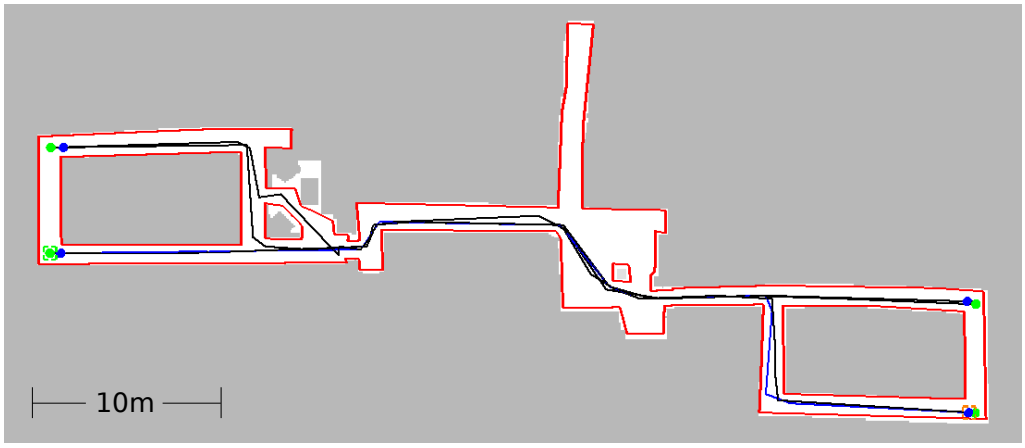


Figure 5.5: Andrew's Hall (gray and white), with the polygon obstacle map (red), and Conflicting solutions found when robot team sizes = 1 (black paths). Each robot plans from its current location (blue) to a goal location (green).

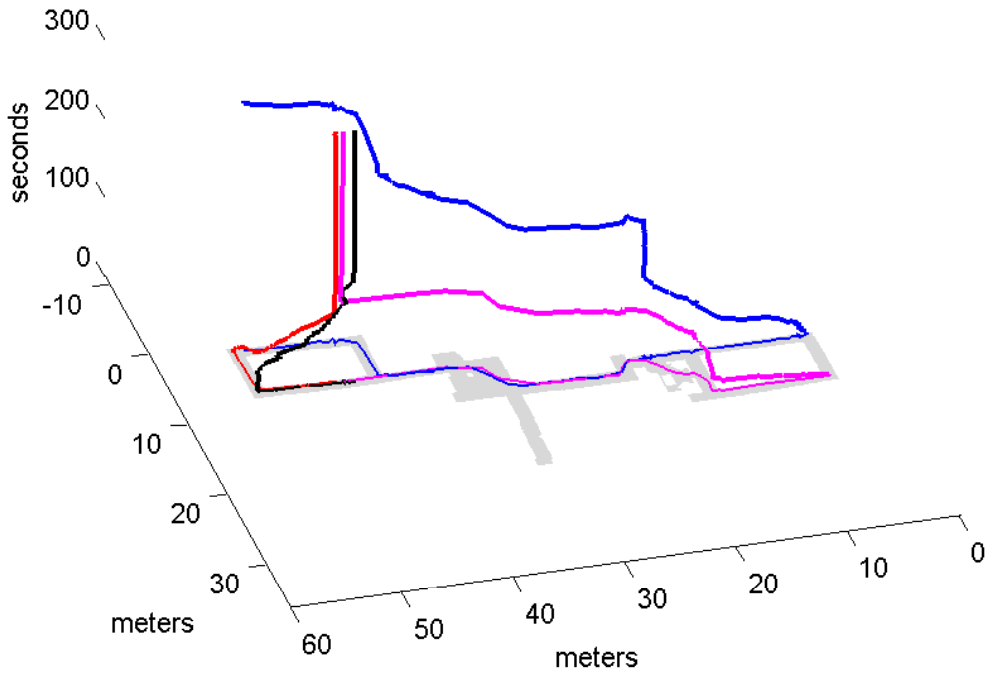


Figure 5.6: Actual robot paths vs. time (thick colored lines), and their projections on the free-space of Andrews Hall (thin colored lines and gray, respectively). Time 0 corresponds to the start of movement. Blue successfully reaches the goal, but the experiment is aborted when the red + black + pink team is unable to find a solution after 10 minutes. Note, only the first 5 minutes of the experiment are plotted.

Figure 5.5). Two robots start in each of the east and west wings of the building, respectively. Robots are assigned the task of trading places with a robot in the opposite wing. As discussed in

the previous section, robots initially plan a path to the goal for themselves, then form larger teams and re-plan if they encounter other robots/teams who's paths conflict with their own. Planning is forced to take place in the entire configuration space $\mathbf{C}_{\mathbf{\Delta}} = \mathbf{C}$. However, teams are not combined unless robots have conflicting paths and are closer to each-other than 3 meters. 5 runs of the experiment are performed.

Failure or partial failure is observed in all 5 runs of the experiment. That is, never did all four robots successfully reach their goals in a single run. I postpone a full discussion of this result until the next section; however, the failure is due to the computational complexity of a large workspace combined with large dynamic team size. Planning through the entire residence hall is too complex a problem for teams of more than two robots to handle in any reasonable amount of time, and the design of the building makes it unlikely that all robots will reach their goals without forming teams of size three or four. Over all 5 runs, the average observed communication quality between any two members of the same team is 60.69% with a standard deviation of 22.64%—so communication quality is not to blame for the team's repeated failure to find a solution.

Figure 5.6 illustrates robot location vs. time from a typical run of the experiment when $\mathbf{C}_{\mathbf{\Delta}} = \mathbf{C}$. The red robot catches up to the black robot and they form a dynamic team in the east wing of the building (the left wing of the plot). Before red and black discover a 2-robot solution, the pink robot joins their team. The resulting 3 robot team is unable to find an initial solution after planning for 10 minutes, and the experiment is halted. In this run, blue is the only robot able to successfully reach its goal.

5.2.2 Small Andrews Hall experiment

This experiment is designed to highlight the important role of the size of $\mathbf{C}_{\mathbf{\Delta}} \subsetneq \mathbf{C}$ by forcing a worst case team size Δ to plan in a subset of the environment used in the previous experiment. 4 robots start in locations as they might enter the large common room in the center of Andrew's Hall. Teams are combined if conflicting robots are within 10 meters of each-other. This causes all robots to quickly form one large team. $\mathbf{C}_{\mathbf{\Delta}}$ is defined such that $\mathbf{W}_{\mathbf{\Delta}}$ is the entire common room.

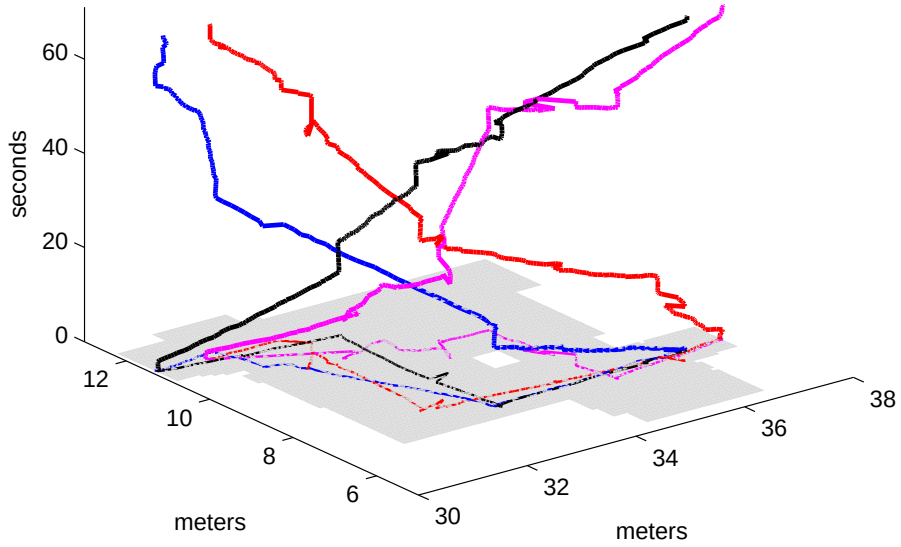


Figure 5.7: Workspace is limited to the large common room. Actual robot paths vs. time (thick colored lines), and their projections on the free-space (thin colored lines and gray, respectively). Time 0 corresponds to the start of movement. The resulting robot paths are plotted vs. time (thick colored paths), along with their projections on the free-space (colored paths and gray, respectively).

Table 5.1: Small Andrews Hall experiment statistics, 10 runs with 4 robots

	mean	standard deviation
In-team communication quality	67.47%	26.68%
Time to first solution (seconds)	14.24	9.55
First solution length (meters)	46.59	4.30
Final solution length (meters)	38.02	3.27
Actual distance traveled (meters)	44.88	7.80

10 runs are performed, and all robots successfully reach their goals in every run. Figure 5.7 depicts robot locations vs. time for a typical run. The mean and standard deviation of observed statistics over all 10 runs are displayed in Table 5.1. The mean measured distance robots actually traveled is greater than the mean final solution length due to small pose jumps between localization tags and temporary localization error incurred between global localization updates during rotation.

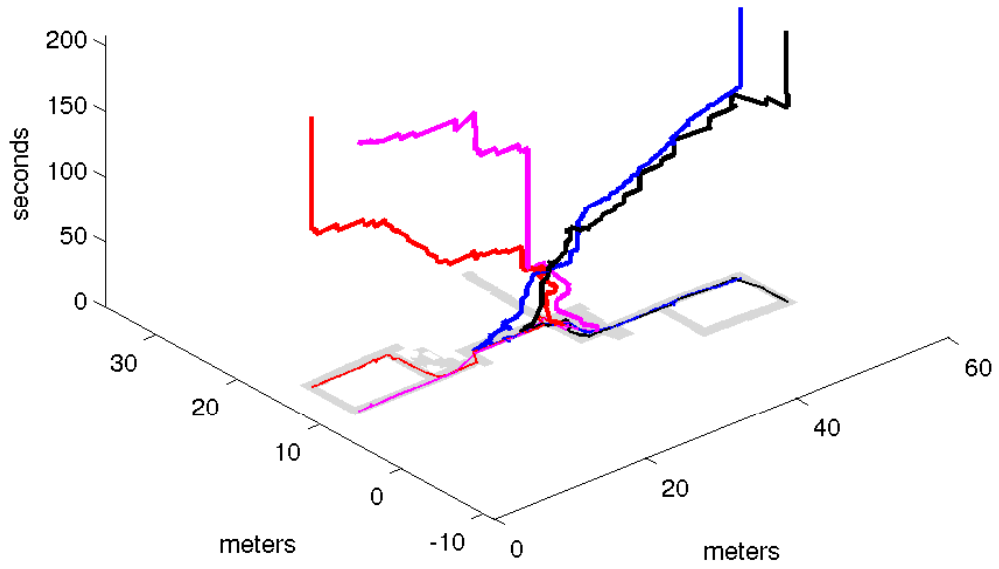


Figure 5.8: Actual robot paths vs. time (thick colored lines), and their projections on the free-space of Andrews Hall (thin colored lines and gray, respectively). Time 0 corresponds to the first team formation.

Table 5.2: Large Andrews Hall experiment (with conflict region selection) statistics, 10 runs with 4 robots

	mean	standard deviation
In-team communication quality	63.18%	27.48%
Actual distance traveled (meters)	161.8	12.6

5.2.3 Large Andrews Hall experiment with conflict region selection

This experiment is designed to verify that Dynamic-Team Any-Com C-FOREST using conflict region selection can overcome the difficulties encountered by the same algorithm without conflict region selection (e.g., as observed in the large Andrews Hall experiment). In other words, to test the hypothesis that using $\mathbf{C}_{\mathbf{\Delta}} \subsetneq \mathbf{C}$ is beneficial. Four robots are placed as they would enter the common room/hallway in the center of the building and told to go to the corners of the opposite wing. This results in a bottle-neck condition that favors the creation of a 4-robot team. Each team is allowed to plan for twice the time it takes to find an initial solution.

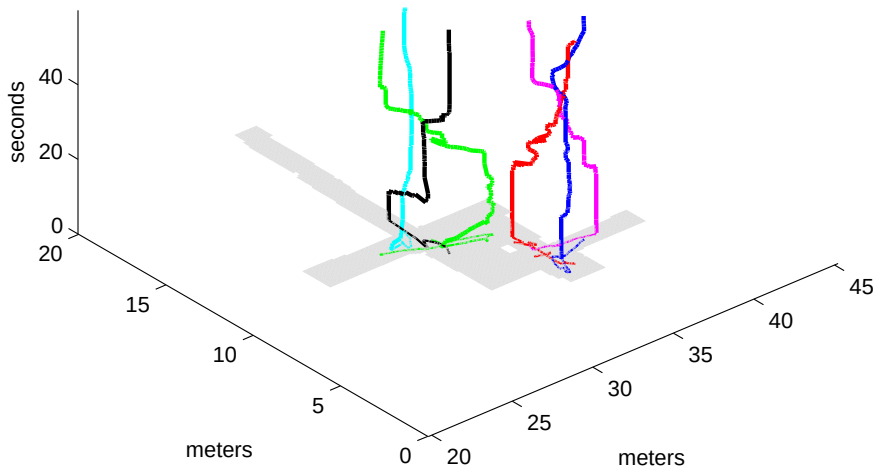


Figure 5.9: A typical solution from the six robot experiment with dynamic teams and conflict region selection. Actual robot paths vs. time (thick colored lines), and their projections on the free-space of Andrews Hall (thin colored lines and gray, respectively). Two teams of three robots each form on either side of the common room. Note that the floor-plan is included to aid in visualization and does not correspond to the conflict areas.

Table 5.3: Six robot experiment (with dynamic-teams) statistics, 10 runs with 6 robots

	mean	standard deviation
In-team communication quality	63.81%	13.72%
Actual distance traveled (meters)	37.30	8.60

10 runs are performed, and all robots successfully reach their goals in every run. Figure 5.8 depicts robot locations vs. time for a typical run. The mean and standard deviation of observed statistics over all 10 runs are displayed in Table 5.2.

5.2.4 Six robot experiment with dynamic-teams

This experiment is designed to showcase the full power of Dynamic-Team Any-Com C-FOREST. Two groups of three robots are set up at the two bottle-neck positions in either end of the common room in Andrew's Hall (Figure 5.9). The robots are told to exchange places within

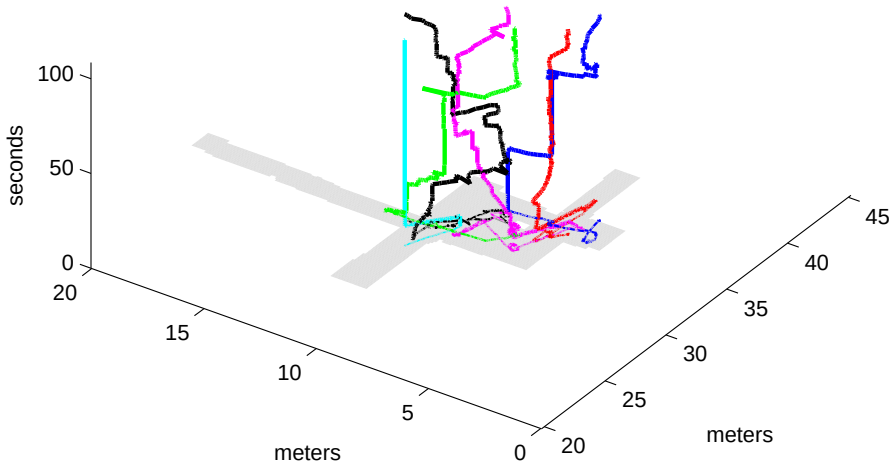


Figure 5.10: A typical solution from the six robot experiment without dynamic teams but still using conflict region selection. Actual robot paths vs. time (thick colored lines), and their projections on the free-space of Andrews Hall (thin colored lines and gray, respectively). Paths tend to be longer than in the previous experiment. Note that the floor-plan is included to aid in visualization and does not correspond to the conflict area.

Table 5.4: Six robot experiment (without dynamic-teams) statistics, 10 runs with 6 robots

	mean	standard deviation
In-team communication quality	66.59%	9.21%
Actual distance traveled (meters), if an experiment was successful	64.62	18.35

each group. This placement is used to encourage the formation of two three robot teams. Each team is allowed to plan for twice the time it takes to find an initial solution.

10 runs are performed, and all robots successfully reach their goals in every run. Figure 5.9 depicts robot locations vs. time for a typical run. The mean and standard deviation of observed statistics over all 10 runs are displayed in Table 5.3.

5.2.5 Six robot experiment without dynamic-teams

This experiment is designed to test the importance of using dynamic teams, assuming that conflict region selection is being used. As in the previous experiment, two groups of three robots are set up at the two bottle-neck positions in either end of the common room in Andrew's Hall (Figure 5.9). The robots are told to exchange places within each group. However, unlike the previous experiment all robots are programmed to immediately form a single 6-robot team. Note that the team still uses conflict region selection to choose $\mathbf{C}_{\mathbf{\Delta}} \subsetneq \mathbf{C}$. The team is allowed to plan for twice the time it takes to find an initial solution.

10 runs are performed. In 5 of 10 runs the test is halted due to a time-out, where time-out occurs if an initial solution has not been found within five minutes of planning time. Two other experiments failed after a solution was found: on one run the battery ran out in a net-book, and on another there was a non-recoverable localization error. Figure 5.10 depicts robot locations vs. time for a typical run. The mean and standard deviation of observed statistics over all 10 runs are displayed in Table 5.4, where communication quality is over all 10 runs, and distance traveled is over the three experiments where all robots reached the goal.

5.3 Discussion of Dynamic-Team Any-Com C-FOREST results

When $\mathbf{C}_{\mathbf{\Delta}} = \mathbf{C}$ in the large Andrews Hall experiment the configuration space of teams with $\Delta \geq 3$ is so large that the problem becomes intractable. This happens despite the distributed computing power of Any-Com C-FOREST, and the fact that teams are not combined unless robots are close to each other. The observation that intra-team packet loss was less than 40%, on average, suggests that the team's difficulty in finding a solution was due to problem complexity and not poor communication. This result highlights the fact that dynamic team formation can lead to grid-lock and failure because it forces teams to deal with the worst-case complexity of the problem.

Both the small Andrews hall experiment and the large andrew's hall experiment with conflict region selection show that $\mathbf{C}_{\mathbf{\Delta}} \subsetneq \mathbf{C}$ is beneficial even in conjunction with a worst-case team size. I

believe these results highlight the benefits of using $\mathbf{C}_{\mathbf{\Delta}} \subseteq \mathbf{C}$.

The six robot experiments reiterate the importance of using dynamic teams to reduce team size (e.g., in addition to the size of the configuration space associated with each robot). Even though the same problem instance is used in both experiments, results in the dynamic team experiment are much better than those in the non-dynamic-team experiment. Using dynamic teams allows the six robot problem to be broken into two easily solved three robot problems. Not using dynamic teams both increases problem complexity to the point that half of the runs end in a time-out, and causes increased path length when runs do not time out.

5.4 Dynamic-Team Any-Com C-FOREST conclusions

In this chapter I present a distributed centralized multi-robot path-planning algorithm called Dynamic Team Any-Com C-FOREST. Each robot starts in its own team, and teams are joined if their individual solutions conflict. Combining teams based on path conflicts allows non-conflicting teams to solve problems of reduced complexity in parallel. Within a particular team, the ad-hoc distributed computation of Any-Com C-FOREST utilizes the computing potential of all team-members to find better solutions more quickly. In order to reduce problem complexity, the resulting multi-robot path planning problems are constrained to a subset of the original configuration space. Robots coordinate progress along combined solution to avoid conflicts, but can move to and from the combined solution using their previously calculated paths.

Dynamic-Team Any-Com C-FOREST is experimentally evaluated in a large, complex, indoor environment. Experiments evaluate the benefits of solving the multi-robot planning problem through a reduced portion of the configuration space. When instructed to swap places from one end of the building to the other, robots succeed in forming teams after they discover each other. However, when the building's entire configuration space is used for planing the resulting problem complexity prohibits teams of three or more robots from finding a solution within any useful amount of time. In contrast, re-planning through the conflict region (a small subset of the environment around the original conflict) reduces problem complexity substantially, allowing a solution to be

found and all robots to reach their goals. Further experiments show that reducing the size of the configuration space per robot is often not enough to guarantee tractability, and highlight the importance of decreasing team size when possible.

Although computational complexity of centralized algorithms is a known theoretical issue, the experiments in this chapter show experimentally that it can cause dynamic team formation to trigger grid-lock and failure—due to increased team size and/or environment size. This illustrates the importance of algorithms that are able to choose appropriately complex sub-problems for dynamic teams to solve. I believe that dynamic teams operating in large environments should attempt to solve the smallest sub-problems necessary to avoid collision, while also maintaining completeness. This strategy is advantageous because it decreases the size of the base to which the exponent is raised.

Dynamic Team Any-Com C-FOREST attempts to extend the size of problems for which a complete solution can be calculated. By delaying team formation as long as possible and planning in a reduced subset of the configuration space, dynamic teams attempt to minimize the complexity of the problem required to be solved by any particular team. When a team must be formed, each team uses Any-Com C-FOREST to divide the computational effort of calculating the members' common solution.

Chapter 6

Conclusions

The thesis that I argue in this dissertation is "*Sharing Any-Time search progress over an ad-hoc distributed computer that is created from a dynamic team of robots enables probabilistically complete, centralized, multi-robot path-planning across a broad class of instances with varied complexity, communication quality, and computational resources.*" To support this thesis I propose and experimentally evaluate three new algorithmic advances. In Chapter 3 I demonstrate that centralized algorithms can be parallelized in an extremely efficient manner and propose a new distributed framework for single-shot high dimensional planners called C-FOREST. In Chapter 4 I investigate the idea that a networked team of robots can be used as an ad-hoc distributed computer in order to use C-FOREST for solving the multi-robot path planning problem that exists between team members. I call the resulting algorithm Any-Com C-FOREST because the algorithm automatically adopts to utilize whatever communication is available. In Chapter 5 I extend the idea again, this time allowing dynamic teams to form and dissolve as robots move through the environment. If a robot conflicts with another team, then it is added to the team, and the team uses Any-Com C-FOREST to find a conflict free solution through the problematic region. the latter algorithm is called Dynamic Team Any-Com C-FOREST.

The success of Dynamic-Team Any-Com C-FOREST provides the best evidence for the validity of my overarching thesis. It demonstrates that it is possible to automatically adjust problems to minimize computational loads by keeping both teams sizes and their configuration spaces small. It also shows that this can be done while simultaneously adapting to different levels of communication

(due to the fact that it is built on Any-Com C-FOREST). In addition to supporting my thesis, the work presented in this dissertation has numerous contributions, both in the field of path planning and beyond. A quick summary includes:

(1) Proposal of a new method for distributed single-query path planning called C-FOREST.

- Theoretical proof that C-FOREST can have super linear speedup vs. number of CPUs.
- Experimental validation that C-FOREST can have super linear speedup in practice.
- Proposal of a modified Sequential C-FOREST for use on a single CPU.
- Experimental validation that Sequential C-FOREST is beneficial on a single CPU.
- Experimental validation that C-FOREST and Sequential C-FOREST can be used with a variety of distance metrics, configuration spaces, robots, and random search-trees.

(2) Proposal of the Any-Com C-FOREST algorithm for use on an ad-hoc distributed computer.

- Coinage of the term "Any-Com" to describe distributed and/or multi-agent algorithms that have graceful performance declines vs. decreasing communication quality.
- Demonstration that a team of robots can be used as an ad-hoc distributed computer.
- Experimental validation that Any-Com C-FOREST performs well in practice, even when communication is relatively poor.

(3) Proposal of Dynamic-Team Any-Com C-FOREST algorithm, in which robots form teams and re-plan in sub-regions of the configuration space as necessary to avoid collisions.

- Observation that increasing team size can cause practical planning failure due to increased problem complexity.
- Theoretical insight into how the size of the configuration space can be reduced while maintaining probabilistic completeness.

- Experimental validation that Dynamic-Team Any-Com C-FOREST allows more difficult planning instances to be solved than Any-Com C-FOREST, with respect to planning time.
- Experimental validation that both team size and configuration space diameter independently contribute to problem complexity.

The distributed C-FOREST algorithm presented in Chapter 3 has provably super-linear speedup vs. the number of CPUs that are used—a rare and useful property. The C-FOREST idea can be extended to any random high-dimensional search tree that is expected to converge to optimality, and in any search space that obeys the triangle inequality. I experimentally demonstrate its usefulness (including super linear speedup) in conjunction with a variety of underlying search-trees, configuration spaces, robots, distance metrics, and cluster sizes.

I propose a practical way of using ad-hoc distributed computers to solve multi-agent problems in In Chapter 4. Where ad-hoc refers to *both* (1) the fact that ad-hoc wireless communication is used to connect computational nodes, and (2) the on-the-fly process by which the distributed computer emerges as independent computational nodes discover each other. While I envision ad-hoc distributed computers being most useful to multi-agent problems that have a dependency on geographic location, I hope the idea is also useful in other domains.

In Chapter 4 the term "Any-Com" is coined to describe the class of algorithms that utilize whatever available communication exists to calculate a solution, where more communication means that better solutions can be found more quickly. While Any-Com algorithms are naturally suited for use in an ad-hoc distributed computing framework, this concept can also be found in other domains. I propose the Any-Com C-FOREST algorithm for solving the centralized multi-robot path planning problem. Any-Com C-FOREST enables the robots involved in a particular path-planning problem to pool their resources, by forming an ad-hoc distributed computer, in order to solve it.

Finally, I observe that forming teams in order to solve the centralized multi-robot path plan-

ning can make the resulting problem too computationally complex to solve in a realistic amount of time. The reason for this is that problem complexity is exponentially dependent on the dimensionality of the search-space, and dimensionality is proportional to the number of robots in a team. I propose Dynamic-Team Any-Com C-FOREST to (1) keep teams as small as possible by only combining teams when doing so is necessary in order to avoid collisions, and (2) have teams plan through the smallest sub-set of the search space necessary to overcome the collision state while also maintaining algorithmic completeness.

Bibliography

[1] R. Alami, S. Fleury, M. Herrb, F. Ingrand, and F. Robert. Multi-robot cooperation in the martha project. Robotics and Automation Magazine, IEEE, 5(1):36–47, 1998.

[2] R. Alami, F. Robert, F. Ingrand, and S. Suzuki. Multi-robot cooperation through incremental plan merging. In Proc. IEEE International Conference on Robotics and Automation, pages 2573–2578, 1995.

[3] J. Allred, A. B. Hasan, S. Panichsakul, W. Pisano, P. Gray, J. Huang, R. Han, D. Lawrence, and K. Mohseni. Sensorflock: an airborne wireless sensor network of micro-air vehicles. In Proceedings of the 5th international conference on Embedded networked sensor systems, pages 117–129, 2007.

[4] N. M. Amato and L. K. Dale. Probabilistic roadmap methods are embarrassingly parallel. In Proc. IEEE International Conference on Robotics and Automation, pages 688–694, 1999.

[5] P. Amstutz, N. Correll, and A. Martinoli. Distributed boundary coverage with a team of networked miniature robots using a robust market-based algorithm. Annals of Mathematics and Artifcial Intelligence. Special Issue on Coverage, Exploration, and Search, 52(2-4):307–333, 2009.

[6] B. Aronov, M. de Berg, A. F. van der Stappen, P. Svestka, and J. Vleugels. Motion planning for multiple robots. In Proceedings of the fourteenth annual symposium on Computational geometry, pages 374–382, Minneapolis, USA, 1998.

[7] F. Arrichiello, J. Das, H. Heidarsson, A. Pereira, S. Chiaverini, and G. S. Sukhatme. Multi-robot collaboration with range-limited communication: Experiments with two underactuated ASVs. In International Conference on Field and Service Robots, 2009.

[8] K. Asarm and G. Schmidt. Conflict-free motion of multiple mobile robots based on decentralized motion planning and negotiation. In Proc. IEEE International Conference on Robotics and Automation, pages 3526–3533, 1997.

[9] R. A. Askey and R. Roy. Beta function. In F. W. J. Olver, D. M. Lozier, and R. F. Boisvert, editors, NIST Handbook of Mathematical Functions. Cambridge University Press, 2010.

[10] P. Bachmann. Die Analytische Zahlentheorie. Zahlentheorie. B. G. Teubner, Leipzig, 1894.

[11] N. Balakrishnan and A. C. Cohen. Order statistics and inference. Acedemic Press, New York, 1991.

[12] J. Barraquand and J. C. Latombe. Robot motion planning: A distributed representation approach. International Journal of Robotics Research, 10(6):628–649, 1991.

[13] M. Bennewitz, W. Burgard, and S. Thrun. Optimizing schedules for prioritized path planning of multi-robot systems. In Proc. IEEE International Conference on Robotics and Automation, pages 271–276, 2001.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. Communications of the ACM, 18(9):509–517, 1975.

[15] M. Boddy and T. L. Dean. Solving time-dependent planning problems. In Proc. Eleventh International Joint Conference on Artificial Intelligence, pages 979–984, 1989.

[16] M. Bonert. Motion planning for multi-robot assembly systems. M.S. dissertation, University of Toronto, 1999.

[17] M. Bonert, L. H. Shu, and B. Benhabib. Motion planning for multi-robot assembly systems. International Journal of Computer Integrated Manufacturing, 13(4):301–310, 2000.

[18] V. Boor, N. H. Overmars, and A. F. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In Proc. IEEE International Conference on Robotics and Automation, pages 1018–1023, 1999.

[19] V. Braitenberg. Vehicles: Experiments in Synthetic Psychology. MIT Press, Cambridge, MA, 1984.

[20] M. S. Branicky, S. M. LaValle, K. Olson, and L. Yang. Quasi-randomized path-planning. In Proc. IEEE International Conference on Robotics and Automation, pages 1481–1487, 2001.

[21] L. Breiman. Random forests. Machine Learning, 45(1):5–32, 2001.

[22] S. J. Buckley. Fast motion planning for multiple moving robots. In Proc. IEEE International Conference on Robotics and Automation, pages 322–326, 1989.

[23] S. Caselli and M. Reggiani. Randomized motion planning on parallel and distributed architectures. In In Euromicro Workshop on Parallel and Distributed Processing, pages 297–304, 1999.

[24] C. Chang, M. J. Chung, and B. H. Lee. Collision avoidance of two general robot manipulators by minimum delay time. IEEE Transactions on Systems, Man and Cybernetics, 24(3):517–522, 1994.

[25] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. Principles of Robot Motion: therory, algorithms and implimentations. MIT Press, Cambridge, 2004.

[26] C. M. Clark, T. Bretl, and S. Rock. Applying kinodynamic randomized motion planning with a dynamic priority system to multi-robot space systems. In Proc. IEEE Aerospace Conference, volume 7, pages 3621–3631, 2002.

[27] C. M. Clark and S. Rock. Randomized motion planning for groups of nonholonomic robots. In Proc. International Symposium of Artificial Intelligence, Robotics and Automation in Space, 2001.

[28] C. M. Clark, S. M. Rock, and J.-C. Latombe. Dynamic networks for motion planning in multi-robot space systems. In Proc. 7th International Symposium on Artificial Intelligence, Robotics and Automation in Space, pages 3621–3631, 2003.

[29] C. M. Clark, S. M. Rock, and J.-C. Latombe. Motion planning for multiple mobile robots using dynamic networks. In Proc. IEEE International Conference on Robotics and Automation, pages 4222–4227, 2003.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms. MIT Press, Cambtidge, MA, 2001.

[31] N. Correll, N. Arechiga, A. Bolger, M. Bollini, B. Charrow, A. Clayton, F. Dominguez, K. Donahue, S. Dyar, L. Johnson, H. Liu, A. Patrikalakis, T. Robertson, J. Smith, D. Soltero, M. Tanner, L. White, and D. Rus. Building a distributed robot garden. Intelligent Service Robots, Special Issue on Agricultural Robotics, 3(4):219–232, 2010.

[32] H. Davis, A. Bramanti-Gregor, and J. Wang. The advantages of using depth and breadth components in heuristic search. Methodologies for intelligent systems, 3:19–28, 1988.

[33] R. Diankov and J. Kuffner. Openrave: A planning architecture for autonomous robotics. Technical Report CMU-RI-TR-08-34, Robotics Institute, Pittsburgh, PA, July 2008.

[34] E. W. Dijkstra. A note on two problems in connection with graphs. In Numererical Mathematics, volume 1, pages 269–271, 1959.

[35] C. Dixon and E. W. Frew. Maintaining optimal communication chains in robotic sensor networks using mobility control. In International conference on Robot Communication and Coordination, 2007.

[36] J. Elston, E. Frew, D. Lawrence, P. Gray, and B. Argrow. Net-centric communication and control for a heterogeneous unmanned aircraft system. Journal of intelligent and Robotic Systems, 56(1-2):199–232, 2009.

[37] M. Erdmann and T. Lozano-Perez. On multiple moving objects. Algorithmica, (2):477–521, 1987.

[38] H. R. Everett, D. W. Gage, G. A. Gilbreath, R. T. Laird, and R. P. Smurlo. Real-world issues in warehouse navigation. In Proceedings of the SPIE Conference on Mobile Robots IX, volume 2352, pages 629–634, Boston, MA, 1994.

[39] H. R. Everett, R. T. Laird, G. A. Gilbreath, T. A. Heath-Pastore, R. S. Inderieden, K. Grant, and D. M. Jaffee. Multiple resource host architecture for the mobile detection assessment and response system. In Technical Document 3026, Space and Naval Warfare Systems Center, San Diego, CA, 1998.

[40] B. Faverjon. A local based approach for path planning of manipulators with a high number of degrees of freedom. In Proc. IEEE International Conference on Robotics and Automation, volume 4, pages 1152–1159, 1987.

[41] D. Ferguson, N. Kalra, and A. Stentz. Replanning with RRTs. In IEEE International Conference on Robotics and Automation, pages 1243–1248, 2006.

[42] D. Ferguson and A. Stentz. Field D*: An interpolation-based path planner and replanner. 2005.

[43] D. Ferguson and A. Stentz. Anytime RRTs. In Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 5369–5375, 2006.

[44] D. Ferguson and A. Stentz. Using interpolation to improve path planning: The field D* algorithm. Journal of Field Robotics, 23(2):79–101, 2006.

[45] D. Ferguson and A. Stentz. Anytime, dynamic planning in high-dimensional search spaces. In Proc. IEEE International Conference on Robotics and Automation, pages 1310–1315, 2007.

[46] R. Gayle, K. R. Klingler, and P. G. Xavier. Lazy reconfiguration forest (LRF) - an approach for motion planning with multiple tasks in dynamic environments. pages 1316–1323, 2007.

[47] R. L. Graham, D. E. Knuth, and O. Patashnik. Answer to problem 9.60. In Concrete Mathematics: A Foundation for Computer Science, 2nd ed.

[48] Y. Guo and L. D. Parker. A distributed and optimal motion planning approach for multiple mobile robots. In Proc. IEEE International Conference on Robotics and Automation, pages 2612–2619, 2002.

[49] Y. Hada and K. Takasa. A robust and deadlock free navigation of mobile robots based on a task-level feedback control. In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 629–634, 1999.

[50] Y. Hada and K. Takasa. Multiple mobile robot navigation using the indoor global positioning system (igps). In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 1005–1010, Hawaii, United States, 2001.

[51] E. A. Hansen and R. Zhou. Anytime heuristic search. Journal of Artificial Intelligence Research, 28:267–297, 2007.

[52] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In Proc. IEEE Transactions On System Science and Cybernetics, pages 100–107, 1968.

[53] G. Hollinger and S. Singh. Multi-robot coordination with periodic connectivity. In Proc. IEEE International Conference on Robotics and Automation, 2010.

[54] G. Hollinger, S. Yerramalli, S. Singh, U. Mitra, and G. Sukhatme. Distributed coordination and data fusion for underwater search. In IEEE International Conference on Robotics and Automation, pages 349–355, 2011.

[55] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects; PSPACE-hardness of the warehouseman's problem. The International Journal of Robotics Research, 3(4):76–88, 1984.

[56] M. A. Hsieh, L. Chaimowicz, A. Cowley, B. Grocholsky, J. Keller, V. Kumar, C. J. Taylor, Y. Endo, R. Arkin, B. Jung, D. F. Wolf, G. S. Sukhatme, and D. MacKenzie. Adaptive teams of autonomous aerial and ground robots for situational awareness. Journal of Field Robotics, 24(11):991–1014, 2007.

[57] D. Hsu, T. Jiang, J. Reif, and Z. Sun. The bridge test for narrow passages with probabilistic roadmap planners. In Proc. IEEE International Conference on Robotics and Automation, 2003.

[58] D. Hsu, J. C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In Proc. IEEE International Conference on Robotics and Automation, pages 2719–2726, 1997.

[59] R. K. D. Hsu, J. C. Latombe, and S. Rock. Kinodynamic motion planning amidst moving obstacles. In Proc. IEEE International Conference on Robotics and Automation, pages 537–543, 2000.

[60] G. Huber. Gamma function derivation of n-sphere volumes. The American Mathematical Monthly, 89(5):301–302, 1982.

[61] L. Jiang, J.-H. Huang, A. Kamthe, T. Liu, I. Freeman, J. Ledbetter, S. Mishra, R. Han, and A. Cerpa. Sensearch: Gps and witness assisted tracking for delay tolerant sensor networks. In Proceedings of the 8th International Conference on Ad-Hoc, Mobile and Wireless Networks, pages 255–269, 2009.

[62] K. Kant and S. W. Zuker. Trajectory planning in time-varying environments, 1: Tpp = ppp + vpp. In Technical Report TR-84-7R, McGill University, Computer vision and Robotics Laboratory, Canada, 1984.

[63] K. Kant and S. W. Zuker. Toward efficient trajectory planning: the path -velocity decomposition. The international journal of robotics research, 5(3):72–89, 1986.

[64] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In Proceedings of Robotics: Science and Systems, Zaragoza, Spain, June 2010.

[65] S. Kato, S. Nishiyama, and J. Takeno. Coordinating mobile robots by applying traffic rules. In Proc. IEEE International Conference on Intelligent Robots and Systems, pages 1535–1541, 1992.

[66] L. E. Kavraki, J. C. Latombe, R. Motwani, and P. Raghaven. Randomized query processing in robot path planning. Journal of Computer and Science Systems, 57(1):50–60, 1998.

[67] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. The International Journal of Robotics Research, 5(1):90–98, 1986.

[68] S. Koenig and M. Likhachev. Improved fast replanning for robot navigation in unknown terrain. In Proc. IEEE International Conference on Robotics and Automation, pages 968–975.

[69] Y. Koga and J.-C. Latombe. On multi-arm manipulation planning. In Proc. IEEE International Conference on Robotics and Automation, volume 2, pages 945–952, 1994.

[70] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In Proc. IEEE International Conference on Robotics and Automation, 1991.

[71] A. M. Ladd and L. E. Kavraki. Measure theoretic analysis of probabilistic path planning. IEEE Transactions on Robotics and Automation, 20(2):229–242, 2004.

[72] E. Landau. Handbuch der Lehre von der Verteilung der Primzahlen. B. G. Teubner, Leipzig, 1909.

[73] J.-C. Latombe. Motion planning: A journey of robots, molecules, digital actors and other artifacts. In The International Journal of Robotics Research, volume 18, pages 1119–1128, 1999.

[74] S. M. LaValle. Planning Algorithms. Cambridge University Press, Cambridge, 2006.

[75] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In IEEE International Conference on Robotics and Automation, pages 473–479, 1999.

[76] S. M. LaValle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In Algorithmic and Computational Robotics: New Directions, pages 293–308, 2001.

[77] B. H. Lee and C. Lee. A minimum-time trajectory planning method for two robots. IEEE Transactions on Systems, Man and Cybernetics, 17(1):21–32, 1987.

[78] D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. Acta Informatica, 9(1):23–29, 1977.

[79] S. Leroy, J. P. Laumond, and T. Simeon. Multiple path coordination for mobile robots: a geometric algorithm. In Proc. International Conference on Artificial Intelligence, 1999.

[80] S. Li. Concise formulas for the area and volume of a hyperspherical cap. Asian Journal of Mathematics and Statistics, 4(1):66–70, 2011.

[81] T.-Y. Li and Y.-C. Shie. An incremental learning approach to motion planning with roadmap management. In IEEE International Conference on Robotics and Automation, pages 3411–3416, 2002.

[82] M. Likhachev and S. Koenig. Incremental a*. In Proceedings of the Neural Information Processing Systems, 2001.

[83] V. J. Lumelski and A. A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. IEEE Transactions on Automatic Control, AC-31(11):1057–1063, 1986.

[84] V. J. Lumelsky and K. R. Harinarayan. Decentralized motion planning for multiple mobile robots: The cocktail party model. Autonomous Robots, 4:121–135, 1997.

[85] J. R. Marden and A. Wierman. The limitations of utility design for multiagent systems. submitted for journal publication, 2011.

[86] M. C. Mozer and J. Bachrach. Discovering the structure of a reactive environment by exploration. Neural Computation, 4(2):447–457, 1990.

[87] D. Murray and J. Little. Using real-time stereo vision for mobile robot navigation. In Proc. of the IEEE Workshop on Perception for Mobile Agents, Santa Barbara, California, 1998.

[88] J. Ng and T. Braunl. Performance comparison of bug navigation algorithms. Journal of Intelligence and Robotic Research, 50(1):73–84, 2007.

[89] P. A. O'Donnell and T. Lozano-Perez. Deadlock-free and collision-free coordination of two robotic manipulators. In Proc. IEEE International Conference on Robotics and Automation, pages 484–489, Scottsdale, AZ, 1989.

[90] M. Otte and N. Correll. The any-com approach to multi-robot coordination. In IEEE International Conference on Robotics and Automation: Network Science and Systems Issues in Multi-Robot Autonomy, 2010.

[91] M. Otte and N. Correll. Any-com multi-robot path-planning: Maximizing collaboration for variable bandwidth. In Proc. 10th International Symposium on Distributed Autonomous Robotics Systems, 2010.

[92] M. Otte and N. Correll. Any-com multi-robot path-planning with dynamic teams: Multi-robot coordination under communication constraints. In Proc. 12th International Symposium on Experimental Robotics, 2010.

[93] M. Otte and N. Correll. C-forest: Parallel path planning with super linear speedup. In Submission, 2011.

[94] M. Otte and G. Grudic. extracting paths from fields built with linear interpolation. In International Conference on Intelligent Robots and Systems, St. Louis, 2009.

[95] M. Overmars and P. Svestka. A probabilistic learning approach to motion planning. In Algorithmic Foundations of Robotics (WAFR), pages 19–37, 1995.

[96] R. B. Paris. incomplete beta functions. In F. W. J. Olver, D. M. Lozier, and R. F. Boisvert, editors, NIST Handbook of Mathematical Functions. Cambridge University Press, 2010.

[97] D. Parsons and J. Canny. A motion planner for multiple mobile robots. In Proc. IEEE International Conference on Robotics and Automation, volume 1, pages 8–13, 1990.

[98] R. Philippsen. A light formulation of the E* interpolated path replanner. Autonomous Systems Lab, Ecole Polytechnique Federale de Lausanne, 2006.

[99] J. Phillips, L. Kavraki, and N. Bedrossian. Spacecraft rendezvous and docking with real-time, randomized optimization. In AIAA Guidance, Navigation, and Control, 2003.

[100] E. Plaku, K. E. Bekris, B. Y. Chen, A. M. Ladd, and L. E. Kavraki. Sampling-based roadmap of trees for parallel motion planning. IEEE Transactions on Robotics, 21(4):587–608, 2005.

[101] E. Plaku and L. E. Kavraki. Distributed sampling-based roadmap of trees for large-scale motion planning. In IEEE International Conference on Robotics and Automation, pages 3879–3884, 2005.

[102] G. Ramanathan and V. S. Alagar. Algorithmic motion planning in robotics: coordinated motion of several disks amidst polygonal obstacles. In Proc. IEEE International Conference on Robotics and Automation, pages 514–522, 1985.

[103] J. H. Reif. Complexity of the movers problem and generalizations. In Proceedings of the IEEE Symposium on Foundations of Computer Science, pages 421–427, 1979.

[104] S. M. Ross. Expectation of a random variable. In Introduction to probability models (9th ed.).

[105] S. Rutishauser, N. Correll, and A. Martinoli. Collaborative coverage using a swarm of networked miniature robots. Robotics and Autonomous Systems, 57(5):517–525, 2009.

[106] M. Ryan. Multi-robot path-planning with subgraphs.

[107] M. Ryan. Exploiting subgraph structure in multi-robot path planning. Journal of Artificial Intelligence Research, 31:497–542, 2008.

[108] G. Sanchez and J.-C. Latombe. On delaying collision checking in prm planning: Application to multi-robot coordination. The international Journal of Robotics Research, 21(5):5–26, 2002.

[109] G. Sanchez and J.-C. Latombe. Using a prm planner to compare centralized and decoupled planning for multi robot systems. In Proc. IEEE International Conference on Robotics and Automation, volume 2, pages 2112–2119, 2002.

[110] A. Sankaranarayanan and M. Vidyasagar. A new path planning algorithm for moving a point object amidst unknown obstacles in a plane. In Proc. IEEE International Conference on Robotics and Automation, pages 1930–1936, 1990.

[111] A. Sankaranarayanan and M. Vidyasagar. Path planning for moving a point object amidst unknown obstacles in a plane: a new algorithm and a general theory for algorithm development. In Proc. IEEE International Conference on Decision and Control, pages 1111–1119, 1990.

[112] C. Schimmel. UNIX Systems for Modern Architectures: Symmetric Multiprocesssing and Caching for Kernel Programmers. Addison-Wesley Professional, Reading, MA, 1994.

[113] J. T. Schwartz and M. Sharir. On the piano mover's problem iii. coordinating the motion of several independent bodies: the special case of circular bodies amidst polygonal barriers. In Proc. IEEE International Conference on Robotics and Automation, pages 514–522, 1985.

[114] J. A. Sethian. A fast marching level-set method for monotonically advancing fronts. Proc. Nat. Acad. Sci., 93:1591–1595, 1996.

[115] M. Sharir and S. Sifrony. Coordinated motion planning for two independent robots. In Proceedings of the fourth annual symposium on Computational geometry, volume 3, pages 107–130, 1991.

[116] K. G. Shin and Q. Zheng. Minimum-time collision free trajectory planning for dual-robot systems. IEEE Transactions on Robotics and Automation, 8(5):641–644, 1992.

[117] T. Simeon, S. Leroy, and J.-P. Laumond. Path coordination for multiple mobile robots: A resolution-complete algorithm. IEEE Transactions on Robotics and Automation, 18(1):42–49, 2002.

[118] A. Stentz. Optimal and efficient path planning for partially-known environments. In Proc. IEEE International Conference on Robotics and Automation, pages 3310–3317, 1994.

[119] A. Stentz. The focussed D* algorithm for real-time replanning. In Proc. of the International Joint Conference on Artificial Intelligence, 1995.

[120] I. A. Sucan and L. E. Kavraki. Kinodynamic motion planning by interior-exterior cell exploration. In Algorithmic Foundation of Robotics VIII (Proceedings of Workshop on the Algorithmic Foundations of Robotics), STAR, pages 449–464, 2009.

[121] I. A. Sucan and L. E. Kavraki. On the implementation of single-query sampling-based motion planners. In IEEE International Conference on Robotics and Automation, pages 2005–2011, 2010.

[122] I. A. Sucan and L. E. Kavraki. On the implementation of single-query sampling-based motion planners. In IEEE International Conference on Robotics and Automation, 2010.

[123] J. van den Berg, S. J. Guy, M. Lin, and Dinesh Manocha. Reciprocal n-body collision avoidance. In Proc. International Symposium on Robotics Research, 2009.

[124] R. Voyles, S. Povilus, R. Mangharam, and K. Li. Reconode: A reconfigurable node for heterogeneous multi-robot search and rescue. In IEEE International Workshop on Safety, Security and Rescue Robotics, 2010.

[125] R. M. Voyles, J. Bae, A. C. Larson, and M. A. Ayad. Wireless video sensor networks for sparse, resource-constrained, multi-robot teams. Intelligent Service Robotics, 2(4), 2009.

[126] C. W. Warren. Multiple robot path coordination using artificial potential fields. In Proc. of IEEE International Conference on Robotics and Automation, pages 500–505, Cincinnati, OH, 1990.

[127] E. K. Xidias and N. A. Aspragathos. Motion planning for multiple non-holonomic robots: a geometric approach. Robotica, 26:525–536, 2008.

[128] A. Yasuaki and M. Yoshiki. Collision avoidance method for multiple autonomous mobile agents by implicit cooperation. In Proc. IEEE Conference Intelligent Robots and Systems, volume 3, pages 1207–1212, 2001.

[129] D.-Y. Yeung and G. A. Bekey. A decentralized approach to the motion planning problem for multiple mobile robots. In Proc. IEEE International Conference on Robotics and Automation, volume 4, pages 1779–1784, 1987.

[130] M. Zucker, J. J. Kuffner, and M. S. Branicky. Multipartite RRTs for rapid replanning in dynamic environments. In International Conference on Robotics and Automation, pages 1603–1609, 2007.

Appendix A

Path planning background

This chapter contains a selected review of relevant technical concepts, as well as a survey of related work. It should provide enough background information to understand the rest of my dissertation, and place it within the context of what has been done before. Section A.1 provides a brief summary of concepts such as 'planning' and 'completeness.'

A.1 Conceptual overview

Many of the sub-fields of robotic algorithms have boundaries that are blurred and/or overlap. While some roboticists may have slightly different ideas about the nuanced distinction between inherently similar concepts—such as 'planning' vs. 'path-planning'—I expect they will agree that my version of the taxonomy is well within the bounds of standard practice.

A.1.1 Planning, paths, and navigation

Planning is the task of finding a sequence of actions $\mathbf{P}$ that will cause a system to transition from an initial state $p_{\mathbf{S}}$ to a goal state $p_{\mathbf{G}}$. It is possible to have multiple goal states. The action sequence $\mathbf{P} = \{p_{\mathbf{S}}, p_i, p_{i+1}, ..., p_{\mathbf{G}}\}$ is called a *plan*.

In the context of robotics, states often represent a robot's physical properties (position, speed, force, etc.). *Path-planning* is the particular type of planning in which states represent position and/or orientation—usually defined relative to the real-world. In this case, $\mathbf{P}$ can be visualized as a bread-crumb trail through the world (see Figure A.1-Left). Each bread-crumb or *point* p_i represents

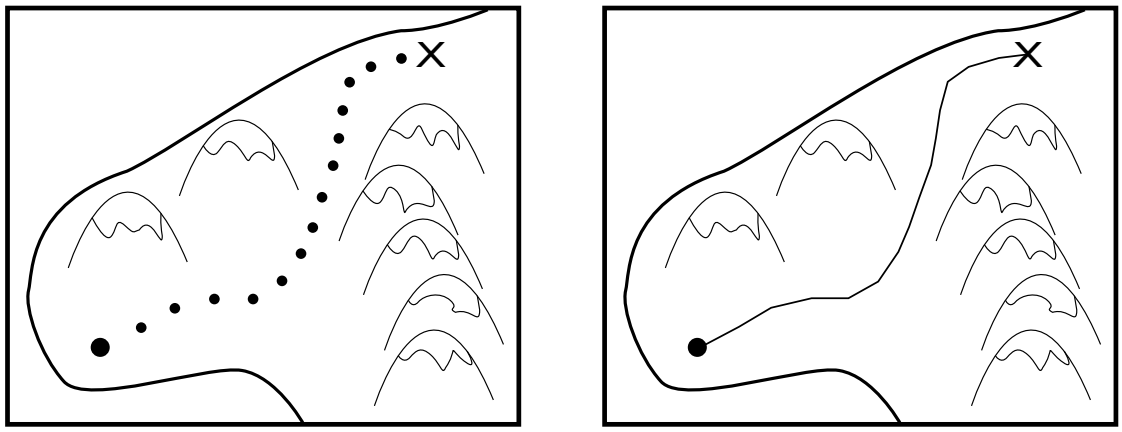


Figure A.1: A path visualized as a trail of bread-crumbs (left) and line segments (right).

a safe location for the robot. It is assumed the robot can maneuver between consecutive points p_i and p_{i+1}. It is standard practice to draw lines between p_i and p_{i+1} for visualization purposes (see Figure A.1-Right). If other physical properties are used instead of position and orientation (e.g., force), then the terms *kinodynamic planning* or *kinodynamic path-planning* are used. This dissertation is concerned with path-planning; however, many of the techniques described may also be useful for kinodynamic planning.

It is often convenient or necessary to associate a time dimension with $\mathbf{P}$, in order to indicate when each p_i should be achieved. This is especially important when a robot must share its environment with other moving bodies. A $\mathbf{P}$ that includes time is called a *trajectory*, and the act of calculating $\mathbf{P}$ is refered to as *trajectory-planning*. Since a trajectory can be thought of as a special type of path, the terms 'path' and 'path-planning' are also frequently used to describe 'trajectory' and 'trajectory-planning'. For the remainder of this dissertation proposal, whether or not a 'path' contains a time dimension (or even if it matters) will be evident from the context in which the term appears. A trajectory will always contain a time dimension.

Plan exicution is the act of following $\mathbf{P}$. It is assumed that $\mathbf{P}$ can be followed to within a small amount of error ϵ, and also that the area located within ϵ of $\mathbf{P}$ is safe. For planning purposes, ϵ can be thought of as a tolerance that is built into the system to guarantee collision avoidance. This is often achieved by increasing robot size (or, alternatively, obstacle size) by ϵ.

Navigation is the combination of deciding how to move and then actually moving. It is a basic primitive of more complex robotic behavior.

One popular form of navigation is based on path-planning. This type of system works by first planning a path **P**, and then executing it. In the event that the agent detects a change in the environment that effects its ability to follow the path, then a new path is planned and executed. This dissertation proposal is concerned with path-planning based navigation; however, other forms of navigation exist (these will be discussed in Section A.2).

A.1.2 Configuration-space vs. work-space

An agent must maintain an internal representation of the world through which to plan a path. In the context of robotic path-planning, there are two important representations one should be familiar with. The *work-space* **W** and the *configuration-space* **C**. The workspace is essentially a map of the environment—there is one dimension in **W** per each dimension in the environment. The configuration-space is an abstract space containing one dimension per each of the system's degrees-of-freedom. Consider a rover moving along the floor of an office-building. **W** is the two-dimensional blue-print of the building, while **C** is the three-dimensional (x, y, θ) space containing all possible combinations of the robot's north-south, east-west, and rotational positions within the building, respectively.

Path-planning occures in **C**, but **C** is defined as a result of the robot's ability to move through **W**. Often it is advantageous to ignore a subset of the dimensions in either **W** or **C**, in order to decrease the computational complexity of the problem. In the office-building scenario described above, **W** does not contain the vertical z dimension—as a result, neither does **C**. Similarly, **C** itself can be reduced by ignoring one or more of the robot's degrees-of-freedom. For instance, rotation can be ignored if the robot can rotate in place. This works because the robot will be able to orient itself at p_{i+1} before moving away from p_i.

Obstacles are regions the robot cannot move through without incurring a collision. Both **W** and **C** contain obstacles; however, obstacles in **C** may either be caused by obstacles in **W** or the

robot's own limitations (e.g., a manipulator arm cannot move through itself).

A.1.3 Completeness

An algorithm is *complete* if it is guaranteed to find a solution in a finite amount of time when one exists, and will report failure in a finite amount of time when a solution does not exist.

A *resolution complete* algorithm will find a solution in a finite amount of time if one exits, but may run forever if a solution does not exist. A typical resolution complete algorithm works by systematically attempting all possible moves through an increasingly accurate environmental representations. In practice, it can be more useful to query whether or not a solution exists for a sufficiently accurate resolution of the environment, since this avoids the possibility of having the algorithm run forever. However, this comes at the price of occasionally overlooking a valid solution.

A *probabilistically complete* algorithm will find a solution when one exists with probability 1 as time approaches infinity. Note, this does not guarantee a solution will be found in finite time [74]. In practice, most algorithms are modified to return failure if they cannot find a solution within a predetermined amount of time.

A.1.4 Any-Time algorithms

Any-time algorithms provide an initial solution as quickly as possible, then better-and-better solutions as time progresses [15]. They allow a trade-off between computation time and solution quality. Ideally, solutions approach optimality as time approaches infinity. Any-Time algorithms are popular in real-time robotic path-planning applications where optimal solutions are difficult to calculate, and actionable solutions must be found in time to guarantee collision avoidance.

A.2 Single-robot navigation

This section briefly describes a few of the major incarnations of single-robot path-planning and navigation techniques. Single-robot methods are relevant to this dissertation proposal because multi-robot navigation often assumes single-robot navigation exists as a basic primitive. Also,

some multi-robot path-planning algorithms are built directly on single-robot versions of the same algorithms.

A.2.1 Reactive algorithms and potential field methods

Often a robot may be directed to seek or avoid various environmental stimuli. This is known as a *reactive algorithm*. While reactive algorithms can be designed for navigation to a destination (e.g., move at the goal), they can also be used to invoke behavior that is not tied to a particular location (e.g., move at the noise). Indeed, it is even possible to implement this type of behavior in an analog system by wiring environmental sensors directly to motors and servos. For this reason, some of the earliest forms of robotic navigation are reactive in nature [19].

Many reactive algorithms model the robot and other environmental features as a system of charged particles or other potential-based physical system. These techniques are known as *potential-field methods* [12, 67, 70, 87]. Navigation to a particular location can be achieved by having the robot attracted to the goal and repulsed by obstacles. Control commands are calculated as a function of the resulting force the (simulated) potential-field exerts on the robot.

A drawback of reactive algorithms is that local optima may trap the robot and prohibit it from ever reaching the goal. For example, the robot may get stuck oscillating between two points or fall into equilibrium on the edge of an obstacle. While the vulnerability to local optima can be reduced by the introduction of random noise, it cannot be eliminated (since the required amount of random noise must be tuned for each new environment the robot encounters). Reactive algorithms are widely used for adjusting a global path based on local disturbances, but tend not to be used for global location-driven navigation.

A.2.2 Rule based navigation (bug algorithms)

Early goal seeking algorithms used provably correct logic in conjunction with simple environmental sensors to navigate. The two-dimensional *Bug Algorithms* are a good example of this type of system [83, 88, 110, 111]. In Bug-1 a robot moves directly at the goal until it reaches an

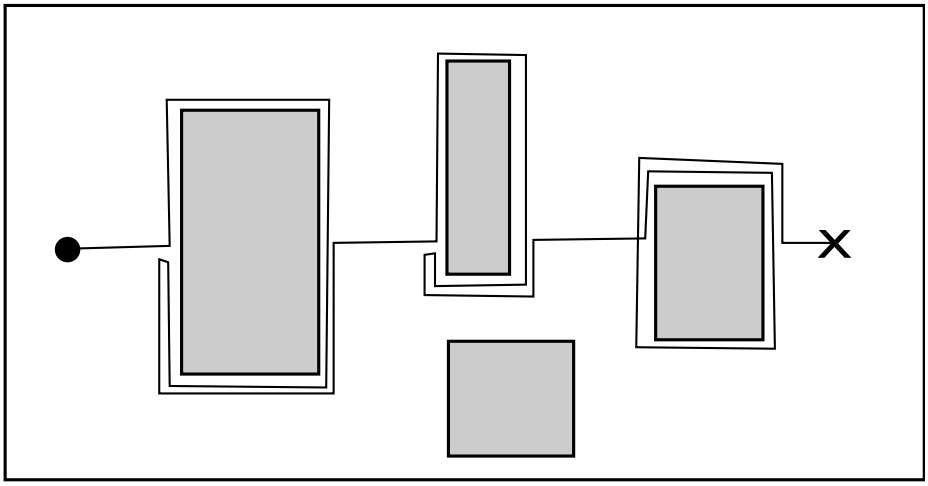


Figure A.2: The route taken by a robot using the Bug1 algorithm.

obstacle. Next, the robot moves around the entire perimeter of the obstacle until it ends up back at the point of initial contact. Finally, the robot *again* moves around the perimeter until it reaches the point that was closest to the goal—where it departs the obstacle to move directly at the goal. This process is repeated every time a new obstacle is encountered (see Figure A.2). Bug-1 is constrained to operate in two-dimensional work-spaces. In the worst case, it spends most of its time traveling around the perimeters of obstacles. Although the addition of more rules can enable more complex and/or efficient behavior, it is hard to reduce the worst-case runtime in all environments [88]. Although popular for two-dimensional navigation, this type of rule-based system can quickly become difficult to analyses in higher-dimensions.

A.2.3 Graph based methods

Often a robot is provided with a map of the world *a priori* and/or is equipped with sensors that enable it to create/update a map as it moves (e.g., laser scanner or camera). In these cases, path-planning is the preferred method of planning, since many algorithms exist that are complete with respect to map data. Graph based path-planning techniques operate in two phases: (1) A graph (directed or undirected) is built in the configuration space, while accounting for map data. (2) A path through the graph is calculated using a graph-search algorithm (see Figure A.3). There

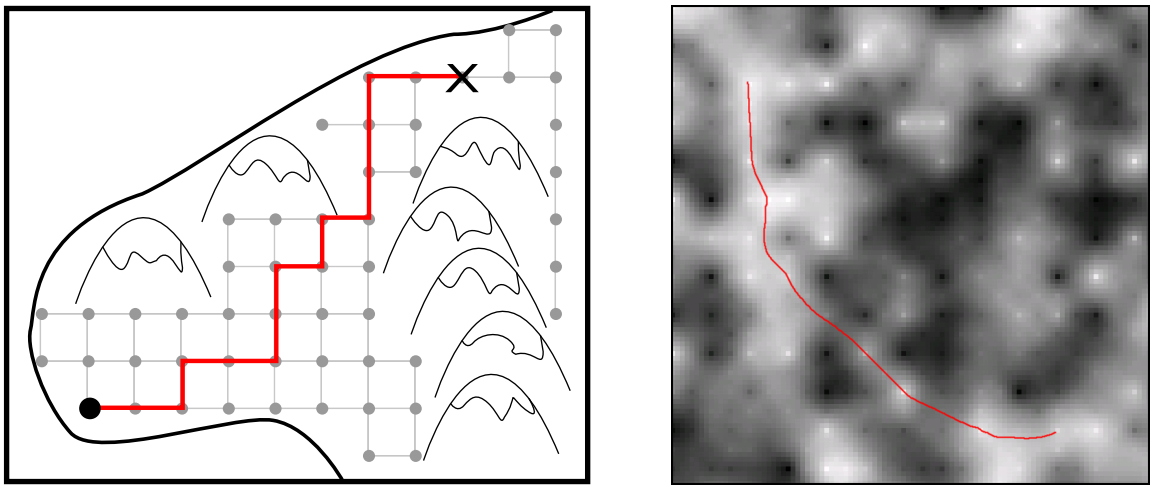


Figure A.3: A path created using a 4-neighborhood grid (Left)—the path is optimal with respect to the graph, but sub-optimal with respect to the real world. A path created using the grid-based algorithm I presented in [94] (Right)—interpolation between edges allows movement in any direction.

are many ways to construct the graph, including (but probably not limited to):

- Along a predefined pattern, such as a grid or repeating hexagons.
- Patterns within patterns (etc.), such as a quad-tree.
- Placing nodes along the edges of obstacles, then connecting nodes that are visible to each other–called a visibility graph [73].
- Learning the graph structure of an environment (e.g., by exploration as in [86]).
- Randomly.

Once built, standard search-techniques can be used to search the graph. While this technically includes basic algorithms such as breadth-first-search, depth-first-search, and Dijkstra's algorithm [30, 34], far more efficient best-first search techniques are usually available—given that graph nodes represent physically related points in the real-world.

Assuming an admissible heuristic exists to estimate the distance to an unexplored node, it is possible to avoid work that cannot possibly lead to a good solution. An early version of this idea

is the A* Algorithm [52]. Given an admissible heuristic, A* is guaranteed to find an optimal path with respect to the graph.

A* is one of the most widely used path-planning algorithms in existence, and it has inspired many descendants. Many of these are designed to solve the special types of path-planning problems encountered in robotics. For example, Lifelong A* is more efficient when multiple goal-searches are performed from different starting locations [82]. The search-tree is rooted at the goal, and most of an old search-tree can be re-used when the starting location changes. In D* and D* lite, most of the old search-tree is reused after the environment changes (e.g. new obstacles are found, old obstacles disappear, or the robot changes location) [68, 118, 119]. Since the map is usually only modified within sensor range of the robot, this often decreases the computation time by orders of magnitude. Any-Time versions of A* also exist [32, 51].

The techniques used to build the graph often create artifacts that cause optimal graph paths to be suboptimal, with respect to the real-world. In particular, imposing a 4-grid on the environment causes movement to be broken into vertical and horizontal segments. Although an 8-grid reduces this problem by also allowing movement along 45 degree diagonals, paths may still be sub-optimal with respet to the real world. Field D* [42, 44] and fast-marching level-set methods [98, 114] seek to overcome this problem by allowing arbitrary movement by interpolating between graph-edges. In [94] I present an extension to the original Field D* algorithm that further reduces artifacts (Figure A.3-Right).

A.3 High-dimensional path-planners

The methods previously described are complete, and even optimal under certain conditions. However, most are ill-suited to plan in more than three dimensions. This section is devoted to path-planning techniques that can be used for problems with many degrees-of-freedom. This includes both single-robot systems, such as manipulator arms, as well as multi-robot systems.

The standard choice for planning in many dimensions still involves a graph-based path. Unlike the low-dimensional methods described in Section A.2.3, most of the computational effort in high-

dimensional graph-based path-planners must be spent on creating the graph (vs. searching through it to find a path). Depending on their application, high-dimensional graph-based path-planners tend to come in one of two flavors: (1) multi-shot planners and (2) single-shot planners. In general, both of these ideas are probabilistically complete.

A.3.1 Multi-query path-planners

If a robot must perform many searches through the same environment, then it is advantageous to create a detailed graph through the configuration space. Once a detailed graph exists, paths can be calculated with relative ease. This idea is known as a *multi-query path-planning*, since the same graph is used to produce multiple paths. Multi-query path-planners tend to be used with industrial manipulator arms and other robots that operate in relatively static environments.

The most widely used multi-query planner is known as *probabilistic road maps* or the *PRM planner* [28, 69, 95, 108, 109]. The initial graph is built by iteratively picking random points in $\mathbf{C}$ and connecting them to the existing graph. This continues until $\mathbf{C}$ contains a sufficiently dense population of graph nodes. Paths are calculated by connecting the start and goal locations to the graph, and then using a standard technique such as A*.

The main challenge lies in ensuring that the final graph adequately fills the configuration space. This can be tricky, especially if there are portions of the environment only accessible via narrow passages. Much research has been devoted to various random sampling techniques [18, 20, 57].

A.3.2 Single-query planners

It does not make sense to save a detailed graph through $\mathbf{C}$ if the robot is expected to encounter a different version of the environment every time it plans. Instead, it is better to focus on finding the smallest graph that enables a decent path to be found for a particular problem. This is known as *single-query planning*. In these methods, graph-creation and graph-search are combined.

Single-query planners usually take the form of random-tree algorithms. As with PRM, new

nodes are chosen by randomly sampling **C**. However, whenever a new point can be connected to the existing graph, it is immediately inserted into the search-tree. Points that cannot be connected to the graph are ignored. Since all nodes in the graph also exist in the search-tree, it is often easier to think of these algorithms as growing a tree through the configuration space from start to goal (or *vise versa*).

As the tree is created, care must be taken not to over-sample any particular portion of the configuration-space (sampling uniformly at random tends to create trees with a disproportionate number of nodes near the root). Often, this concern is addressed by actively sampling from regions that are under-represented in the current tree [58, 59, 99]. While this can be effective, it requires additional overhead to determine the relative representation of any particular area. An algorithm that automatically addresses spatial sampling is called *rapidly exploring random tree* or *RRT* [75, 76]. The RRT algorithm operates by picking a random point p_1 in space, and then finding the tree-node p_t that is closest to p_1 and extending the tree from p_t a small distance toward p_1 (it is also possible to extend p_t all the way to p_1). Re-planning versions also exist [41, 45].

While RRT provides nice probabilistic space-coverage guarantees, its main drawback is that the resulting paths tend to wander around. Although much of this can be eliminated in post-processing, it has been proven that the algorithm will almost surly converge to a sub-optimal solution [64]. A number of attempts have been made to eliminate the wandering algorithmically. *Any-Time RRT* works by iteratively building better and better trees while time remains [43, 45]. *RRT** carefully chooses a set of candidate nodes to extend in-place of p_t, such that the resulting algorithm almost surly converges to the optimal solution [64].

Appendix B

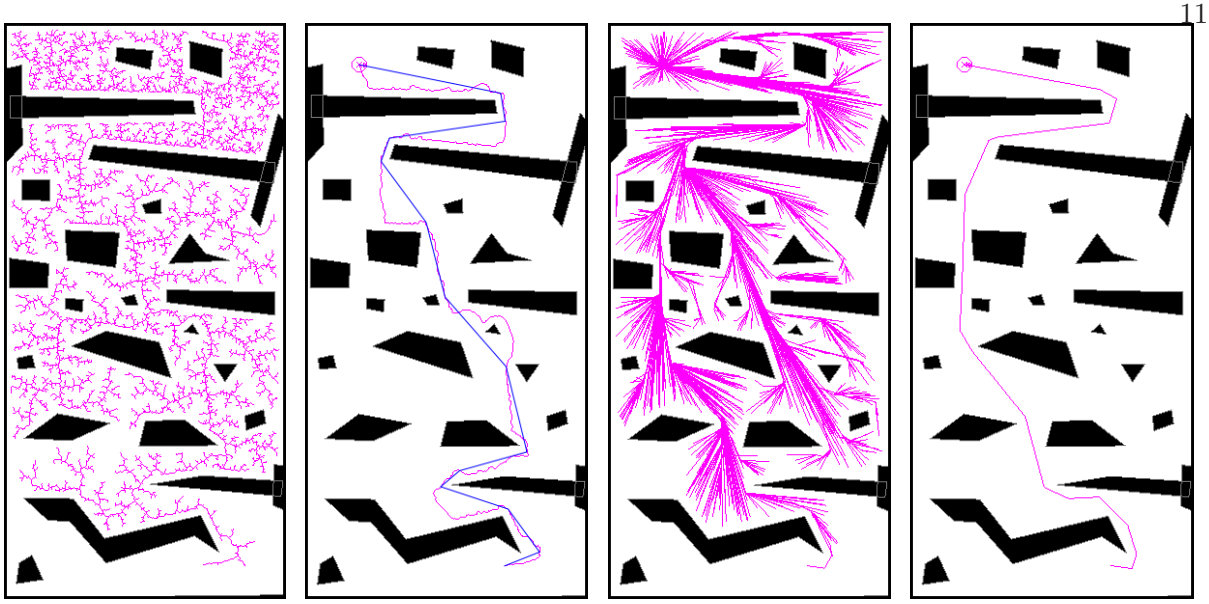
The Any-Time Shortest Path Random Tree path planning algorithm (Any-Time SPRT)

This appendix contains a previously unpublished paper, written by myself under the direction of my advisor Nikolaus Correll, that describes the Any-Time SPRT algorithm that is used as a foundation of much of the work in Chapters 3-5. The version that appears here is self-contained, but retains the original paper's nomenclature and voice—which are *unique* from that found in the rest of my dissertation.

Path planning is an important capability for enabling autonomous robots to function in the real word. Multi-robot planning algorithms find mutually compatible paths for all robots operating in a shared environment [27, 37, 84, 123]. Centralized multi-robot planning algorithms discover this solution using a single agent—usually with guarantees on completeness and/or optimality [109, 113, 127]. However, optimal solutions can be prohibitively expensive, and pragmatic sub-optimal solutions are often tolerated [48, 63].

Any-time algorithms provide an initial solution as quickly as possible, then better-and-better solutions as time progresses [15]. They allow a trade-off between computation time and solution quality. Ideally, solutions approach optimality as time approaches infinity. Any-time algorithms are popular in real-time robotic path-planning applications where optimal solutions are difficult to calculate, and actionable solutions must be found in time to guarantee collision avoidance.

The *Rapidly-exploring random tree* algorithm (or *RRT*) was originally created to search non-



(a) (b) (c) (d)

Figure B.1: Search-trees and resulting shortest-paths (pink) around obstacles (black). (a) and (b) are found using RRT, (c) and (d) using SPRT (our algorithm). Blue is the RRT path after post-processing. RRT wanders the wrong way around some of the obstacles (b), but path shortening only eliminates the detour around the small triangular obstacle and not the large quadrilaterals. SPRT goes the correct way around obstacles and does not require post-processing (d).

convex high-dimensional configuration spaces with differential constraints [75]. RRT is a coverage algorithm that builds a tree such that the expected distance between a random point and the tree is minimized. RRT is widely used to solve single-shot path-planning problems—i.e. cases where the robot or robot-team is expected to traverse the configuration space once.

More recently RRT has been co-opted to solve high-dimensional *shortest-path* planning problems in metric spaces—in particular, the multi-robot shortest-path problem. Despite RRT's nice coverage guarantees, finding shortest-paths is not what it was originally intended for. In fact, because RRT fundamentally performs coverage, and overall tree shape is dependent on a random node insertion order, the resulting trees tend to wander around the configuration space (Figure B.1-a,b). This can lead to sub-optimal results with respect to shortest-paths. Post-precessing can minimize wandering by shortcutting it whenever possible. However, environmental obstacles may still constrain the post-processed path in suboptimal ways (Figure B.1-b).

A more sophisticated method using RRT to find shortest-paths is the Any-Time RRT al-

gorithm [43]. It works by building a secession of individual RRTs, where subsequent RRTs are guaranteed to yield better paths than their predecessors. Improvement vs. time is achieved by using a heuristic to focus search. New nodes are not added unless their cost-to-goal plus the actual cost-to-come through the tree is less than the best known solution length. This is similar to the heuristic used in the A* algorithm [52]. Any-Time RRT finds better solutions to the shortest-path problem than basic RRT. However, despite the focusing heuristic, nodes are still added to minimize distance-to-tree. Thus, the resulting tree still intrinsically performs coverage—albeit constrained to ever more refined search envelopes.

A newer algorithm called RRT* [64] minimize the path wandering by re-wiring old nodes to a newly inserted node if they are within a particular distance of it and will benefit from doing so. The rewire distance is calculated as a function of graph size and carefully managed to guarantee asymptotically similar runtime to RRT, and also convergence to an optimal path. While this is undoubtedly an elegant solution to the path wandering problem, the method relies on finding a new nodes nearest neighbors (i.e., potential rewire-candidates) via a kd-tree [14], and thus is only applicable to configuration spaces in which the "best" neighbor corresponds to the closest based on *Euclidean* distance in the *configuration space*. Our previous research has been on the centralized multi-robot path planning problem, in which distance is defined as either the maximum or the sum over a set of individual distances projected from the configuration space onto the workspace dimensions of each robot. These distance metrics corresponds to maximum time to goal for any robot in the team or the sum of the distance traveled in the real word by all robots. Regardless, these distances are definitely non-Euclidean and therefore cannot be used with RRT*.

Our work is concerned with creating an any-time algorithm for finding shortest-paths in non-convex metric spaces that use any distance function that obeys the triangle inequality. Hence, we call our algorithm *Any-Time Shortest-Path Random Tree* (or *Any-Time SPRT*). Our work is inspired by [43], and developed in parallel to [64], but is fundamentally different in a number of important ways. As in RRT*, we connect new nodes such that *distance-to-root* is minimized instead of distance-to-tree. This fundamentally changes the algorithm vs. RRT from performing coverage

to calculating the shortest-path (see Figure B.1-c,d). Also (similar to RRT*), we maintain a single tree for the duration of the search, and continually improve it to yield better paths—in contrast, Any-Time RRT builds a secession of new trees from scratch. By leveraging previous effort we avoid exploring the same area multiple times. This is important in non-convex environments, where simply locating the goal may be difficult and time-consuming. Third, Any-Time SPRT continually *remodels* the tree—however, the particular way this is done is unique from RRT* and allows our algorithm to be used in conjunction with non-Euclidean definitions of "nearest" neighbors. As with RRT* we re-wire old nodes if they can benefit from being attached to a new node. However, unlike RRT* we must search through all previous nodes in the tree (i.e., instead of a subset returned by the kd-tree). To compensate for the extra time that this requires, we take great care to reduce tree size whenever possible. For instance, *pruning* nodes that cannot possibly help us find better solutions. We believe these differences will allow Any-Time SPRT to find shorter paths more quickly than Any-Time RRT and allow it to be used in non-Euclidean configurations spaces outside the domain of RRT*.

Methodology and theoretical analysis are presented in Sections B.1 and B.2, respectively. In Section B.3 we perform three experiments comparing Any-Time SPRT to Any-Time RRT. We also evaluate the utility of path post-processing in either algorithm. In Section B.4 we discuss results, and conclusions are given in Section B.5.

B.1 SPRT methodology

Sections B.1.1 and B.1.2 outline Any-Time RRT and our Any-Time SPRT, respectively.

B.1.1 Basic Any-Time RRT

Let $\mathbf{C}$ be the robot configuration space. In a multi-robot setting, each individual robot's degrees-of-freedom are combined to obtain the final dimensionality of $\mathbf{C}$ (e.g. 5 robots each operating in R^2 leads to a 10 dimensional $\mathbf{C}$). Let $\mathbf{C}_{free}$ be the subspace of $\mathbf{C}$ that is collision-free. Pseudo-code for Any-Time RRT is given in Figure B.2. *bstln* is the length of the shortest path known

AnyTimeRRT()
1: $bstln = \infty$
2: **while** $time < \mu$ **do**
3: **if RRT**() **then**
4: post best solution
5: delete tree

NewPointWithin(q)
1: pick a point $p_1 \in \mathbf{C}$, where $p_1 = goal$ with probability α
2: $p_2 = \mathbf{Nearest}(p_1)$
3: **if** $h(p_2, p_1) > \delta$ **then**
4: $p_3 = \mathbf{Extend}(p_2, p_1, q)$
5: **if** $h(start, p_3) + h(p_3, goal) \geq bstln$ **and** edge (p_3,p_2) $\in \mathbf{C}_{free}$ **then**
6: **return** $\{p_3, p_2\}$
7: **return** $\{\mathbf{null}, \mathbf{null}\}$

NewRandomPoint()
1: pick a point $p_1 \in \mathbf{C}$, where $p_1 = goal$ with probability α
2: $p_2 = \mathbf{Nearest}(p_1)$
3: **if** $h(p_2, p_1) > \delta$ **then**
4: **if** $h(start, p_1) + h(p_1, goal) \geq bstln$ **and** edge (p_1,p_2) $\in \mathbf{C}_{free}$ **then**
5: **return** $\{p_1, p_2\}$
6: **return** $\{\mathbf{null}, \mathbf{null}\}$

RRT()
1: add $start$ to search-tree
2: **while** $time < \mu$ **do**
3: $\{p_1, p_2\} = \mathbf{NewPointWithin}(q)$
4: **if** $p_1 = \mathbf{null}$ **then**
5: **continue**
6: $S_{dist}(p_1) = S_{dist}(p_2) + h(p_1, p_2)$
7: add p_1 to search-tree as a child of p_2
8: **if** $p_1 = goal$ **then**
9: $bstln = S_{dist}(p_1)$
10: **return true**
11: **return false**

PathShorten($\mathbf{P}$)
1: $f = 1$
2: $b = l$
3: **while** $f < l$ **do**
4: $b = l$
5: **while** $b > f$ **do**
6: **if** edge ($\mathbf{P}_f$,$\mathbf{P}_b$) $\in \mathbf{C}_{free}$ **then**
7: reroute $\mathbf{P}_f$ through $\mathbf{P}_f$
8: $f = b - 1$
9: **break**
10: **else**
11: $b = b - 1$
12: $f = f + 1$

Figure B.2: Any-Time RRT algorithm (Top-Left). **RRT**() finds the next RRT. **NewPointWithin**(q) finds a point to add to the tree that is q away from the current tree. **NewRandomPoint**() finds a random point to add to the tree. **PathShorten**($\mathbf{P}$) greedily shortens path $\mathbf{P}$.

at any particular time. μ is the total time allowed for planning. **AnyTimeRRT**() continuously attempts to find better RRTs, and after each solution is posted the old tree is deleted.

RRT() creates a single RRT. The start is added on line 1 and the tree grows until a new solution is found or no more time remains (lines 10 or 11, respectively). On line 3 a new point p_1 is found that can be connected to a point p_2 already in the tree. On lines 6 and 7 p_1 is added as a neighbor of p_2, while keeping track of $S_{dist}(p)$—the distance from p to the start via the current tree. $h(p_1, p_2)$ is an admissible heuristic estimate of the distance between p_1 and p_2. Specifically, $h(p_1, p_2)$ is the sum of all robots' Euclidean distance between p_1 and p_2 ignoring collisions.

NewPointWithin(q) returns $\{p_3, p_2\}$, where p_3 is new a point located q away from p_2 in the current tree. A random point p_1 is picked on line 1. α is a small predefined probability used to focus search toward the goal. On line 2 we find p_2 the tree-node closest to p_1. On line 3 we check if p_1 is more than a small distance δ away from p_2 to avoid populating the tree with essentially duplicate points (i.e. δ is the search granularity). Subroutine **Extend**(p_2, p_1, q) on line 4 returns the point located q from p_2 toward p_1 (p_1 is returned if $h(p_1, p_2) < q$). On line 5 we check if using p_3 can lead to a better solution based on the start and goal configurations and $bstln$, and verify the edge from p_3 to p_2 is valid.

An additional parameter $\epsilon < 1$ can be used to ensure new RRTs find paths at least $1-\epsilon$ times better than previous solutions. In practice, we find most new solutions are already significantly shorter, and therefore omit ϵ.

B.1.1.1 Holonomic Any-Time RRT

When holonomic robots are used, better results may be achieved by connecting p_2 to p_1 directly, instead of finding p_3 [76]. This is achieved by replacing **NewPointWithin**(q) by **NewRandomPoint**() on line 3 of **AnyTimeRRT**(), and has the additional advantage of eliminating the parameter q.

B.1.1.2 Path shortening

Paths found using RRT tend to wander around—a side effect of RRT performing coverage. Previous work has used a greedy path-shortening algorithm to remove much of this wandering. We believe better results can be obtained by performing the shortening operation after each successful tree has been found (in Any-Time RRT and Holonomic Any-Time RRT). The reason being that this will allow subsequent searches to prune more nodes, thereby focusing effort on finding even better solutions. The specific algorithm we use is displayed in Figure B.2. The path $\mathbf{P}$ has l nodes indexed 1 to l, and $\mathbf{P}_i$ is the i-th node. The algorithm tries to short-cut as many points as possible by sweeping along the path from front-to-back (outer loop, lines 3-12) and back-to-front (inner loop, lines 5-11). Versions of Any-Time RRT with and without this idea are evaluated in Section C.7.

AnyTimeSPRT()

```
1: bstln = ∞
2: while time < μ do
3:     RandomShortcut()
4:     pick a point p1 ∈ C, where
       p1 = goal with probability ρ
5:     if p1 ∉ C_free
       or h(start, p1) + h(p1, goal) ≥ bstln then
6:         continue
7:     p2 = FindBestParent(p1)
8:     if p2 = null then
9:         continue
10:    S_dist(p1) = S_dist(p2) + h(p1, p2)
11:    add p1 to search-tree as a child of p2
12:    if p1 = goal then
13:        bstln = S_dist(p1)
14:        post best solution
15:        PruneTree()
16:    FindChildren(p1)
```

FindBestParent(p_1)

```
1: p2 = null
2: gp2 = bstln
3: for each node pi ∈ Tree do
4:     if p1 is within δ of p2 then
5:         return null
6:     if S_dist(Pi) + h(pi, p1) < gp2 then
7:         if edge (pi,p1) ∈ C_free then
8:             p2 = pi
9:             gp2 = S_dist(Pi) + h(pi, p1)
10: return p2
```

FindChildren(p_1)

```
1: for each node pi ∈ Tree do
2:     if S_dist(pi) + h(pi, p1) < S_dist(p1)
       and edge (pi,p1) ∈ C_free then
3:         S_dist(p1) = S_dist(pi) + h(pi, p1)
4:         reroute pi through p1
5: for descendants of p1 do
6:     update S_dist()
7: if a goal was updated then
8:     bstln = S_dist(goal)
9:     post best solution
10:    PruneTree()
```

RandomShortcut()

```
1: p1 = random node ∈ Tree
2: p2 = null
3: for each node pi ∈ Tree do
4:     if S_dist(pi) + h(pi, p1) < S_dist(p1)
       and edge (pi,p1) ∈ C_free then
5:         p2 = pi
6:         S_dist(p2) = S_dist(Pi) + h(pi, p1)
7: if p2 ≠ null then
8:     reroute p1 through p2
9:     for descendants of p1 do
10:        update S_dist()
11:        if a goal was updated then
12:            bstln = S_dist(goal)
13:            post best solution
14:            PruneTree()
```

PruneTree()

```
1: for each node pi ∈ Tree do
2:     if S_dist(pi) > bstln then
3:         remove all descendents of p1
4:         remove pi
```

Figure B.3: Any-Time SPRT algorithm (Top-Left). **FindBestParent**(p_1) finds the best parent of p_1. **FindChildren**(p_1) reroutes nodes if they would rather have p_1 as their parent. **RandomShortcut**() randomly improves the tree. **PruneTree**() removes nodes that cannot lead to shorter paths.

B.1.2 Any-Time SPRT (shortest path random tree)

Both versions of Anytime RRT are fundamentally coverage algorithms. Although coverage is useful in many circumstances, it can lead to difficulties when finding a shortest path. The main

reason is the wandering nature of RRTs, caused by linking new nodes to have the shortest possible distance-to-tree (Figure B.1-a,b).

Our algorithm is presented in Figure B.3. We believe shorter paths can be achieved by linking nodes to have the least distance-to-root $S_{dist}()$. That is, link p_1 to the tree-node p_2 that gives p_1 the shortest possible path-to-start through the tree. This is done on lines 7-11 of **RandomTree**(). Intelligently connecting nodes cannot completely prevent suboptimal connections. In particular, the optimal parent for a child may be inserted after the child is already in the tree. To account for this, we reroute old nodes that would rather use a new node p_1 as their parent (lines 3-4 of **FindChildren**(p_1)). Afterward, we update the $S_{dist}()$ values of the descendants of p_1 (lines 5-6 of **FindChildren**(p_1)). If a goal node is updated, the new *bstln* is recorded and the corresponding solution is posted (lines 7-9 of **FindChildren**(p_1)).

To avoid a large runtime, a greedy algorithm is used to update $S_{dist}()$ of descendants of p_1 (we postpone a full discussion on runtime until Section B.2.1). While $S_{dist}()$ of all involved nodes is guaranteed to be reduced, an optimal rerouting may not be found. Therefore, we periodically re-link random nodes to their best parents using **RandomShortcut**() to combat suboptimal edges. Although **FindBestParent**() and **FindChildren**($\mathbf{p_1}$) account for the bulk of $S_{dist}()$ reductions, **RandomShortcut**() guarantees resolution optimality as time approaches infinity.

After the first solution has been found, we can use an admissible heuristic to avoid work that cannot possibly lead to better solutions. In particular this is done on lines 4-9 of **RandomTree**(), line 6 of **FindBestParent**(p_1), line 2 of **FindChildren**($\mathbf{p_1}$), and line 4 of **RandomShortcut**(). The heuristic can also be used to prune old-nodes that have become obsolete due to *bstln* (subrutine **PruneTree**()).

B.2 Runtime, theory, and proofs

Let n be the number of nodes in an average tree. Let t be the number of trees created over an entire search. Let m be the total number of nodes we *attempt* to insert into any tree. m/t is the average number of nodes we attempt to insert into a single tree.

B.2.1 Runtime

We must find the nearest neighbor of each node we attempt to insert. A naive implementation of **Nearest**(p_1) examines every node in the tree and requires $O(n)$ time. However, using kd-trees [14] can reduce this to an expected runtime of $O(\log n)$—assuming points are randomly distributed. If points are not randomly distributed, the wost-case runtime is $O(dn^{1-1/d})$, where d is the dimensionality of the configuration space [78]. Thus, the runtime per RRT is expected to be $O((m/t)\log n)$, and the expected runtime of Any-Time RRT is $O(m \log n)$.

While Any-Time RRT uses **Nearest**(p_1) to find the best parent based on distance-to-tree, Any-Time SPRT uses **FindBestParent**(p_1) to find the best parent with respect to distance-to-root. As a result, kd-trees cannot be used (and other tricks currently allude us), so we must perform an $O(n)$ sweep of the entire tree.

The subroutine **FindChildren**(p_1) runs in time $O(n)$. The initial sweep (lines 1-4) uses $O(n)$ time. Updating the descendants of p_1 can be implemented recursively such that each node is touched at most once. This works because each node has only one parent, and all n nodes are descendants of p_1 in the worst-case. If a goal is updated, the tree is pruned in time $O(n)$ (each node is touched at most once). We wait to update $S_{dist}()$ of the descendants of p_1 until after the initial sweep is performed. This is a greedy approach that can lead to suboptimal myopic decisions. For example, if an old node p and its ancestor p_a would both rather use p_1 as their parent, but rerouting causes $S_{dist}(p_a)$ to be reduced more than $S_{dist}(p)$, then p would have actually done better to remain connected to p_a. Therefore, despite being guaranteed by construction to reduce $S_{dist}()$ for all updated nodes, the reduction may not be the largest possible. The reason we settle for a suboptimal greedy algorithm is because updating all nodes optimally would require at least $O(n^2)$ operations—since any updated node has the potential to become a better parent to other nodes. Long-term optimality is ensured by **FindChildren**($\mathbf{p_1}$), and helped by the fact that additional insertions may also decrease distance-to-root.

FindChildren($\mathbf{p_1}$) runs in time $O(n)$. A single sweep of all n nodes is required (lines 3-6)

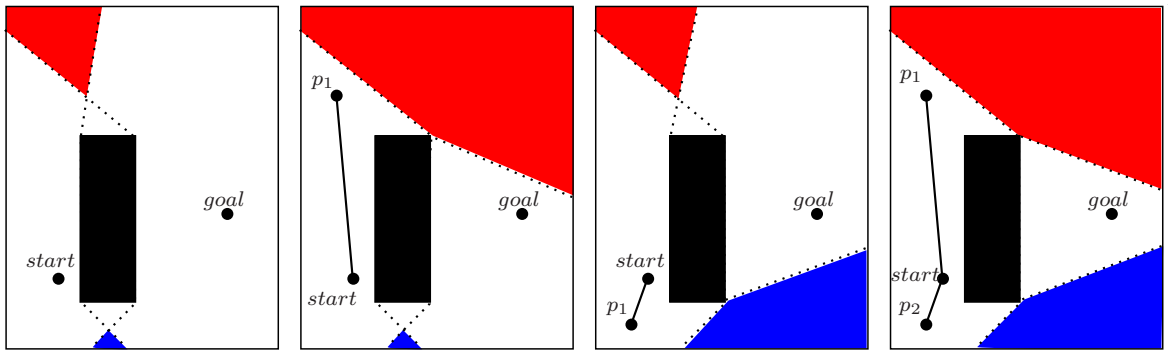


Figure B.4: Assuming the goal is added as p_{i+1} and the tree contains the particular points shown before p_i is added, red R and blue B regions show all possible locations for p_i. If $p_i \in R$ or B then the path will go the long or short way around the obstacle, respectively. In Holonomic RRT the probability a point is inserted into R vs. B is proportional to their respective areas. $i = 1$, 2, 2, and 3 in (a), (b), (c), and (d), respectively. Holonomic RRT is likely to take the long path in this environment.

to find the best parent for p_1. As explained above, updating the descendants of p_1 and pruning the tree can both be implemented in time $O(n)$.

This leads to an overall runtime of $O(nm/t)$ per tree, and a total $O(nm)$ for Any-Time SPRT. Although this is substantially larger than the expected $O(m \log n)$ of Any-Time RRT, we hypothesize Any-Time SPRT will still achieve shorter paths because it actively attempts to find shortest paths instead of performing coverage in ever-more constrained search envelopes.

The path shortening algorithm (Figure B.2) runs in time $O(l^2)$, where l is the length of the pre-shortened path. In practice, $l \ll n$ because n is proportional to the hyper-volume of the search space, while l is proportional to a 1-dimensional path through that space. If Any-Time RRT is modified to perform path-shortening (after each solution is found), then the expected runtime becomes $O(m \log n + tl^2)$

B.2.2 Path wandering phenomenon

The primary purpose of RRT is to perform coverage. The specific properties that encourage coverage also contribute to RRT wandering around the configuration space. This can be problematic if a shortest-path is desired. Even when path-shortening is used, the final path may wander the

long-way around obstacles. Note, in a convex environment there are no holes (i.e. obstacles) to pin the path and RRT with path-shortening may find close to optimal solutions.

Although a thorough investigation of path-wandering is beyond the scope of this paper, the following example provides some intuition (for a more formal discussion see [64]). Consider the single obstacle configuration space in Figure B.4. The start and goal locations are drawn along with dotted-lines on the edges of their visibility envelopes (a visibility envelope is the subset of environment visible from a particular location). Let E_s be the union of visibility envelopes of all nodes in the search-tree. Let E_g be the visibility envelope of the goal. For this discussion, assume holonomic RRT is used. Because points are chosen randomly, the probability a point p is chosen within E is equal to A_E, the area in E, divided by the total configuration area A_C. That is, $P(p \in E) = A_E/A_C$.

Assume the goal is added as the second node after the root. In this case, we know the first point p_1 must be visible by both the start and goal, $p_1 \in E_s \cap E_g$. In Figure B.4-a ($E_s \cap E_g$) is shaded as either red or blue if it is above or below the obstacle, respectively—we denote these regions R or B, respectively. If p_1 exists in R or B, then the path must go the long or short way around the obstacle, respectively. The probability the path goes the short way reduces to $A_B/(A_B + A_R)$, the area of B divided by the combined area of B and R. Thus, when $A_B < A_R$ it is more likely the path will go the long way around the obstacle. This can be extended to higher dimensions by using hyper-volume instead of area.

We can extend this idea to paths with more points, however the general computation is difficult because we must integrate over all possible locations of each point. As a simplified example, consider the parallel cases illustrated in Figures B.4-b and -c, where we fix the location of the first point as shown. As with the previous example, the probability the path goes the short way around the obstacle is given by $A_B/(A_B + A_R)$. Finally, if we assume the goal is added as the fourth point, and fix the locations of the first two points, the resulting case is depicted in Figure B.4-d. It is easy to see holonomic RRT is more likely to go the long way around the obstacle in this particular environment.

Assume Any-Time RRT is being executed and at least one solution has been found. *bstln* contains the length of the best known solution. An interesting consequence of wandering is that *during* the creation of a new RRT, wandering within that tree may prevent the algorithm from ever reaching the goal. Specifically, if there is enough wandering such that all leaf-nodes have $S_{dist} \geq bestln + \theta$ but are yet further from the goal than θ, then it is impossible to find a better solution with the current tree. For this reason, one should consider restarting the RRT after a specific time has elapsed without finding a better solution. Regardless, this scenario provides more evidence that using a coverage algorithm to find a shortest path can be problematic.

B.2.3 Path-length proofs

Here we present a few simple theorems on the relative lengths of paths found using Any-Time SPRT vs. Any-Time RRT.

Theorem B.1: *Assuming a metric space C_{free} and a partially created tree, Any-Time SPRT will add a new point p_i to the tree in a particular way that yields a shorter or equal path-to-root for p_i, when compared to RRT and Any-Time RRT.*

Proof. This is guaranteed by construction because Any-Time SPRT always links a node to the tree in the way that yields the shortest possible path-to-root, given the current tree, using an exhaustive search. If Any-Time SPRT chooses the same parent for p_i as {Any-Time} RRT, then the resulting trees will be identical and p_i will have equal distance-to-root. Otherwise, Any-Time SPRT must have a shorter path-to-root because of the triangle inequality. □

Theorem B.2: *Assuming a metric space, and given a particular ordering of random points to insert into a* **single** *tree, Any-Time SPRT will always find a shorter or equal path to that found by RRT (and by extension Any-Time RRT).*

Proof. This is shown using induction on i. At step i, each node in the Any-Time SPRT tree will have equal or better distance-to-root, compared to the tree that would be generated by Any-Time RRT. If Any-Time SPRT chooses the same parent for p_i as Any-Time RRT, then the resulting

distance-to-root will be equal to or less than that of Any-Time RRT. Otherwise, Any-Time SPRT must have a shorter path-to-root because of the triangle inequality and Theorem B.1. □

Theorem B.3: *Assuming a metric space, and given a particular ordering of random points to insert, Any-Time SPRT will always find a shorter or equal path to that found by Any-Time RRT.*

Proof. Any-Time SPRT keeps all nodes until they cannot possibly yield better paths, while individual Any-Time RRTs are chopped and then regrown. Therefore, any nodes in the most recent RRT of Any-Time RRT that could possibly lead to a better solution must also be available to Any-Time SPRT. Using induction on i, at step i Any-Time SPRT has the same nodes available to it as the current RRT of Any-Time RRT, and Any-Time SPRT will have linked these nodes together in a way that must yield a better or equal distance-to-root (from Theorems B.1 and B.2). However, Any-Time SPRT may also have other nodes (corresponding to Any-Time RRT's previously deleted RRTs), and these may enable even better paths to be found from node i. □

All of the above proofs rely solely on the insertion operation. Additionally considering the find-children and random-improve operations does not affect the correctness of the proofs because each modifies a node/nodes to have even less distance-to-root than they started with—preserving the inductive reasoning steps. It is important to note that all three theorems are defined in terms of a list of random nodes being added to a tree—and we have ignored the time required to insert a particular node. In practice, the faster-per-node Any-Time RRT could conceivably insert more nodes in the same amount of time, and perhaps find better paths as a result. On the other hand, Any-Time RRT is vulnerable to wandering and may require more trees (and thus nodes) to stumble across a desirable solution. We seek to gain insight into these trade-offs experimentally.

B.3 SPRT experiments

To evaluate Any-Time SPRT vs. various versions of Any-Time RRT, we conduct 3 experiments using multiple robots in simulated environments. Comparison methods include standard versions of Any-Time RRT and holonomic Any-Time RRT, as well as modified versions that use

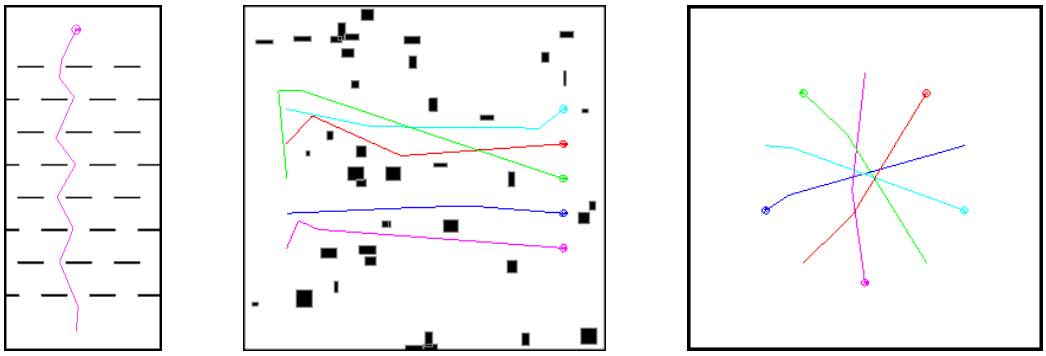


Figure B.5: The workspaces used for Experiments 1, 2, and 3 (Left to Right), and paths.

path-shortening after each solution is found. We evaluate solution quality and the number of nodes used by each algorithm.

Non-holonomic variants of Any-Time RRT require the parameter q to define the maximum distance between new nodes and the current tree. For each environment, we determine q using a two-step parameter sweep averaged over 10 runs per q value. The first step looks at values on an exponential scale, while the second performs a fine grained search near the best value from the first step. We note that, in practice, the optimal value of q for a particular environment is usually not known *a priori*. By determining q in this way, we are *over-fitting* q to each environment (i.e. a particular q is expected to perform well on the environment is was trained on, but should not be expected to transfer to other environments). However, we desire to benchmark our method against the best possible results of the comparison methods.

In all runs of all experiments, robot radius is 0.2 meters, α the probability the goal is used instead of a random point is 0.05, and δ scene resolution is 0.1 meters. As discussed in Section C.4, an RRT should be restarted if it is believed incapable of finding a better solution (i.e. due to path wandering). We record the time τ_1 it takes to find the first solution, and then allow $1.5\tau_1$ before restarting the RRT. For the i-th unsuccessful RRT in a row, we reset $\tau_i = 1.5\tau_{i-1}$ in case τ_{i-1} was too small. After each successful RRT, we reset τ to 1.5 the time it took to create the successful tree.

Experiment 1 consists of a single robot navigating an environment resembling a pachinko

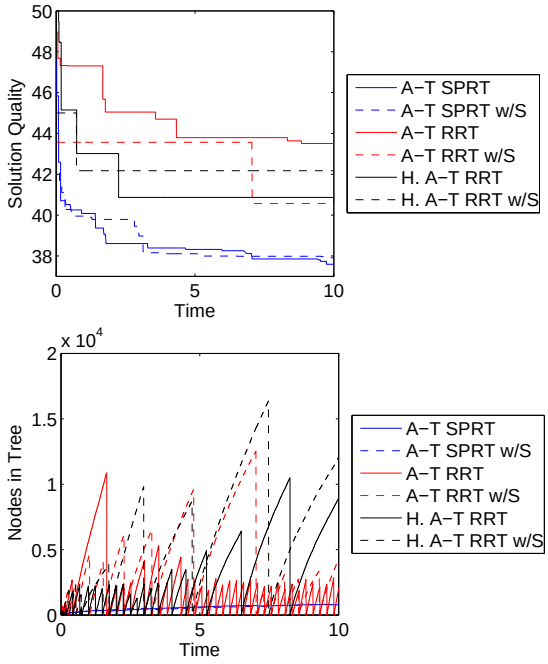


Figure B.6: Path length (Left) and number of nodes in tree (Right) vs. time for a single run of Experiment 1. The saw-tooth appearance of nodes in tree is due to tree deletion in the RRT methods. A-T, H, and w/S denote Any-Time, holonomic, and with path-shortening, respectively.

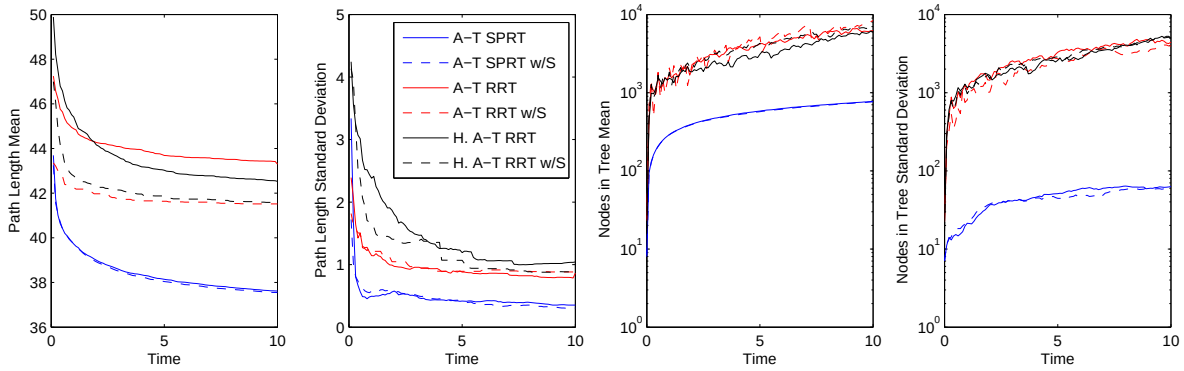


Figure B.7: Experiment 1 path length and tree nodes vs. time over 100 runs. A-T, H, and w/S denote Any-Time, holonomic, and with path-shortening, respectively.

game (Figure B.5-left). $\mu = 10$ seconds. This experiment is designed to illustrate the main differences between Any-Time SPRT and Any-Time RRT in an easy to visualize low dimensional spaces. The results of a single run are displayed in Figure B.6. Mean and standard deviation over 100 runs are displayed in Figure B.7.

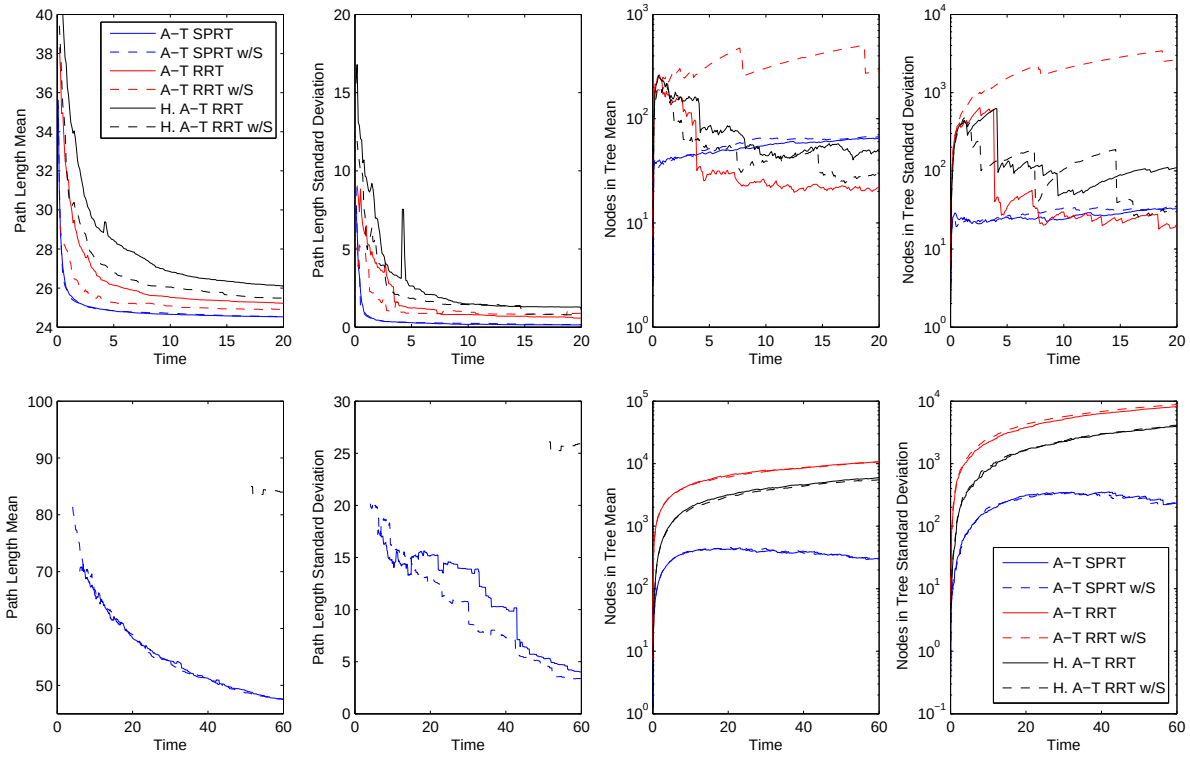


Figure B.8: Experiment 2 path length and tree nodes vs. time over 100 runs. 3 (Top) and 5 (Bottom) robots. A-T, H, and w/S denote Any-Time, holonomic, and with path-shortening, respectively.

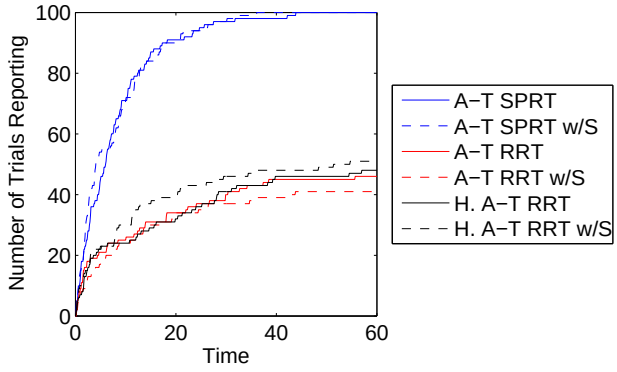


Figure B.9: The number trials that have found their first solution vs. time using 5 robots in Experiment 2, A-T, H, and w/S denote Any-Time, holonomic, and with path-shortening, respectively.

Experiment 2 consists of a random cluttered environment and the addition of more robots. 10 by 10 meter workspace is randomly populated with 40 obstacles—each randomly sized between .1 and .5 meters. Two tests are performed, with 3 and 5 robots and $\mu = 20$ and $\mu = 60$ seconds,

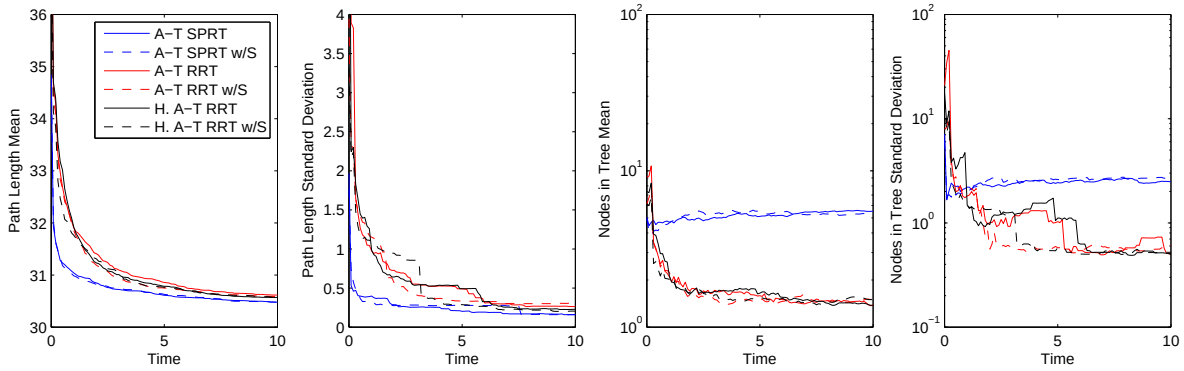


Figure B.10: Experiment 3 path length and tree nodes vs. time over 100 runs. A-T, H, and w/S denote Any-Time, holonomic, and with path-shortening, respectively.

respectively. All robots must move 8 meters. 100 runs are performed per test. The workspace is identical for both tests (Figure B.5-center), but every-other robot is used for the 3-robot test. This experiment is designed to push the limits of our algorithm in high dimensional spaces. Results are displayed in Figure B.8. Length results are omitted at times when less than half the runs of a particular method do not have a solution (remaining results reflect the mean and standard deviation of runs with at least one solution). Figure B.9 shows the number of runs with at least one solution vs. time for the 5-robot series.

Experiment 3 consists of an uncluttered environment with robots starting on one side of a 3 meter circle and ending on the opposite side. We perform 100 tests using 5 robots and $\mu = 10$ seconds. This experiment is designed to asses performance in a nearly-convex configuration space (it is not convex due to robots-robot collisions). The environment is shown in Figure B.5-right. Results are shown in Figure B.10.

B.4 Discussion of SPRT results

Figure B.6-right from Experiment 1 illustrates a main difference between Any-Time SPRT and Any-Time RRT—the former gradually improves a single tree, while the latter restarts a new tree each time the goal is achieved. Frequent tree-chopping is observed as a saw-tooth pattern for the RRT based methods. Subsequent figures do not have the saw-tooth pattern because they are

averaged over 100 runs, but the effects of tree deletion can be observed as large standard deviations in node count.

One interesting result is that Any-Time SPRT requires less n average nodes than Any-Time RRT—an order of magnitude less in the highly-non-convex environments of Experiments 1 and 2. Recall Any-Time SPRT has $O(nm)$ runtime while Any-Time RRT has $O(m \log n)$ expected runtime. Using relatively low n helps Any-Time SPRT equalize the number of iterations it can perform within a given planning time, relative to Any-Time RRT. We believe Any-Time SPRT uses fewer average nodes due its aggressive pruning strategy. In addition to removing a particular node from further consideration, pruning also prevents irrelevant new nodes from being added as its descendants. Also, by minimizing path wandering, Any-Time SPRT finds low *bstln* values early on, tightening the search envelope and enabling even more pruning. Small search envelopes tend to decrease node count because the maximum number of tree-nodes is proportional to the hyper-volume of the search envelope.

Node usage is irrelevant if an algorithm fails to find decent shortest-paths. Luckily, Any-Time SPRT finds shorter paths than the comparison methods in all environments—although, results are only slightly better in Experiment 3. We note that in Experiment 2 with 5 robots Any-Time RRT struggles to find a single solution, while Any-Time SPRT quickly finds one and refines it (Figures B.8 and B.9). We believe Experiments 1 and 2 favor Any-Time SPRT because they are highly-non-convex. Shortest-paths must go around many obstacles, and RRT wanders the wrong way around many of them (recall the case discussed in Section C.4, Figure B.4). In contrast, Any-Time RRT performs just about as well as Any-Time SPRT in the nearly-convex environment of Experiment 3, and requires less nodes per tree. Indeed, both methods require very few nodes in Experiment 3.

We believe another reason Any-Time SPRT performs well in highly-non-convex spaces is because it can leverage all previous work, by maintaining nodes that might help it find a better path, instead of regrowing a new tree from scratch every time the goal is achieved. In such spaces, simply reaching the goal is difficult and requires substantial effort. Algorithms that regrow a new

tree must invest this effort before each better solution is found. In contrast, our algorithm puts that effort into modifying the old tree to yield better-and-better paths. Modifying the tree requires substantially less effort than regrowing an entire tree. The effects of this are mainly observable in Figure B.6 as path-lengths that quickly decrease little-by-little (Any-Time SPRT) vs. long jumps between larger improvements (Any Time RRT). When results are averaged over many runs, we believe this behavior translates into path-lengths that asymptote more quickly toward optimality (Figures B.7, B.8, and B.10).

Another interesting observation is that path-shortening reduces path-length for both Any-Time RRT algorithms, but does little to improve Any-Time SPRT. While this illustrates Any-Time RRT should be used with path-shortening, it also provides evidence that Any-Time SPRT is already finding close-to-optimal paths.

A main assumption of Any-Time SPRT is operation in a metric space. This allows the heuristic pruning of unhelpful nodes. Whether or not similar ideas can be modified for non-metric spaces is beyond the scope of this paper—but an interesting possibility for future work. Another assumption is the multi-rover planning paradigm. This allows collision detection vs. obstacles by projecting a potential path into R^2 (or R^3) per each particular robot. Runtime may be negatively affected if collision were performed in a higher dimensional space. Many high dimensional collision detection algorithms have runtime dependent on path segment length, and our method uses longer path-segments to reach the goal as quickly as possible. Therefore, it remains to be seen whether or not Any-Time SPRT is applicable to other the higher dimensional path-search problems (e.g. manipulators).

B.5 SPRT conclusions

The main contributions of our work are the presentation, explanation, analysis, and experimental evaluation of a new algorithm for multi-rover mutual shortest-path planning in non-convex high-dimensional metric spaces. We call our algorithm the Any-Time Shortest Path Random Tree (or Any-Time SPRT). Any-Time SPRT is unique from but inspired by Any-Time RRT, and devel-

oped in parallel to RRT*. Any-Time RRT is built directly on RRT, which was designed to perform coverage in non-convex high-dimensional spaces. We observe that a side effect of the latter property is that RRT based algorithms tend to wander, sometimes the wrong way around obstacles, which increases the resulting path-lengths. Any-Time SPRT attempts to correct these issues, and also to reuse as much work as possible between individual any-time solutions.

The two main differences between Any-Time SPRT and Any-Time RRT are: (1) Any-Time SPRT adds nodes such that their distance-to-root is minimized, while Any-Time RRT minimizes distance to tree. (2) Any-Time SPRT maintains and improves a single tree, while Any-Time RRT grows a new tree each time the previous tree reaches a goal. In addition to connecting new nodes such that they receive the least possible distance-to-root, Any-Time SPRT performs three other operations designed to gradually improve the search tree over time, with respect to distance-to-root. The first is to reroute old nodes through newly added nodes if doing so will improve the distance-to-root of the rerouted nodes. The second is to periodically reroute randomly selected nodes through their best parent. The third is to prune all nodes that cannot possibly lead to better solutions (after each solution is found).

The main difference vs. RRT* is that RRT* depends on using a kd-tree as a subroutine, and therefore is only suited to operate in configuration spaces in which Euclidean distance can be used to determine nearest neighbors. In contrast, Any-Time SPRT can be used in any configuration space that obeys the triangle inequality but is arguably less well suited for use in Euclidean configuration spaces due to its greater asymptotic runtime. To combat the greater runtime of Any-Time SPRT vs. tree size we actively attempt to reduce the latter. This is done by aggressive pruning of nodes that cannot possibly lead to a better solution.

Given a sequence of random nodes to insert, we prove Any-Time SPRT always finds a path at least as short as Any-Time RRT. However, Any-Time SPRT has $O(mn)$ run-time while Any-Time RRT has expected runtime $O(m \log n)$, where n is the number of nodes in an average tree. Therefore, it is possible Any-Time RRT may outperform Any-Time SPRT with respect to planning time. To evaluate the practical performance of Any-Time SPRT vs. Any-Time RRT, we perform 3

experiments.

We observe Any-Time SPRT finds better solutions in less time than Any-Time RRT. In highly-non-convex environments (Experiments 1 and 2) we find Any-Time SPRT maintains less n—*by at least an order of magnitude*—which makes up for its larger per-node runtime. We believe Any-Time SPRT uses fewer nodes, on average, for two reasons: (1) It actively prunes nodes that cannot lead to better solutions. (2) By minimizing path wandering early on, it quickly shrinks the hyper-volume of the search envelope and therefore avoids considering many irrelevant nodes. On the other hand, we observe Any-Time SPRT requires more nodes, on average, in nearly-convex metric spaces (Experiment 3)—although both algorithms require very few nodes (and Any-Time SPRT still finds marginally better solutions than Any-Time RRT). We believe Any-Time RRT performs relatively well in this case because there are no obstacles for the search-tree to wander the wrong way around.

As a secondary contribution, we evaluate path-shortening as a post-processing step for Any-Time RRT (after each individual RRT is found). We find path-shortening significantly improves path quality because it reduces the natural wandering of RRT, and enforces tighter search envelopes on subsequent trees. Path-shortening does not significantly improve our method—evidence it already finds close-to-optimal paths.

Appendix C

On the expected length of greedy paths through random graphs

As was the case with Appendix-B, Appendix-C, contains a previously unpublished paper, written by myself under the direction of my advisor Nikolaus Correll, that is self-contained and retains the original paper's nomenclature and voice—*unique* from the rest of my dissertation. This appendix describes work that I have done on the expected lengths of random paths. The theory that is presented here is used as a rationalization for the particular parameter selection I use to tune the random trees in Chapters 3-5.

Much work has gone into inventing and empirically testing randomized path planning algorithms. Meanwhile, theoretical analysis is usually limited to proving guarantees on completeness, coverage, feasibility, and/or asymptotic optimality. While these are important contributions, much less work has gone into finding theoretical answers to questions like:

- How does the convergence rate of one algorithm compare to another?
- How does an algorithm's performance change vs. the dimensionality of the search-space?
- How do algorithmic parameters such as edge length affect convergence?

Although many algorithms exist that can find a solution, we do not know *a priori* which algorithm is likely to work best in a particular domain. Further, there is usually only a limited theoretical understanding of how parameter tuning will affect performance, given a particular algorithm. These two problems are currently addressed in practice via empirical evaluation. That is,

by running repeated experiments comparing algorithms and/or parameters for whatever targeted problem domain the author is currently investigating.

We believe that a better theoretical understanding of path planning algorithms will lead to the development of better algorithms and improve the use of current algorithms. In this paper we present an analytic technique that can be used to answer targeted questions about an algorithm's performance. In particular, we show how to obtain bounds on the expected performance of randomized path planning algorithms using statistics derived from an algorithm's geometry. For example, finding the expected path length returned by an algorithm after it has sampled N random nodes from the environment.

The main contribution of this paper is the analytical tool itself. The algorithms used to demonstrate the tool's use are relatively simple. While we hope the technique will be applied to more complex algorithms in the future, the current demonstration indeed provides useful insights despite the simplicity of the algorithms studied.

The rest of this paper is organized as follows: In the remainder of this section we outline related work. In Sections C.1, C.2, and C.3 we provide problem formulation, define our nomenclature, and provide a high-level overview of our technique, respectively. Section C.4 is devoted to applying our technique to simple problems, and Section C.5 we extend analysis to more complex problems. In Section C.6 we examine the applications and implications of the analytical results, with respect to commonly used algorithms. In Section C.7 we perform a series of experiments to demonstrate the accuracy of our analytical methods. Discussion and conclusions are given in Section C.8.

C.0.1 Related Work

Early graph algorithms found optimality guarantees with respect to a graph [34, 52], while other planning algorithms were shown to have theoretical convergence to a goal state [83, 110]. More recently, work on proving bounds on the probability of failure for probabilistic road-maps (PRMs) has been done in [58] using the concept of expansiveness. Completeness, as well as an inverse exponential bound on the chance of failure for PRM is given by [71]. Random graph reachability is

covered by [66]. The Rapidly Exploring Random Tree (RRT) algorithm is presented in [75], where guarantees on its coverage are given. The RRT* algorithm is presented in [64], along with proof of its asymptotic optimality. Much work, too numerous to mention, has focused on experimental evaluation of algorithm performance. [25] provides a good survey, while a summary containing many useful lessons for single-query planning is given by [122].

C.1 Random path problem formulation

We assume that the configuration space is Euclidean, and that random sampling is uniform and i.i.d.. We believe our technique can be modified to handle many non-Euclidean spaces, and/or non-uniform sampling. However, the current paper is intended as an introduction, and leaves these more difficult cases for future work.

A graph G is defined by a set of nodes (i.e., vertices) G_V and a set of edges G_E. Let v_i represent the i-th node. An edge is an ordered pair of nodes $[v_i, v_j]$ that signifies connectivity between v_i and v_j. In a directed graph, edge connectivity is unidirectional and the ordering of an edge pair $[v_i, v_j]$ means that the edge goes from v_i to v_j. In an undirected graph, edges allow movement in either direction. Often each edge is defined to have a cost $c(v_i, v_j)$.

A path P is any sequence of N_p nodes connected by $N_p - 1$ edges with the property that for $i = 1...N_p - 1$ the edge $[v_i, v_i + 1] \in P$. The cost of a path c_P is found by summing the cost of its edges.

$$c_P = \sum_{[v_i, v_{i+1}] \in P} c(v_i, v_j)$$

With respect to a set of paths, the minimum cost path P_{min} (often called the best path P^*), is the particular path that has the least total cost. That is $P_{min} = \arg\min_P (c_P)$.

In this paper we are concerned with graphs that model the connectivity of a Euclidean configuration space. Cost is defined as Euclidean distance. We are interested in finding paths that move between a starting state and a goal state, where state is defined as either a point or region in the configuration space (or a set of points and/or regions).

The graph is a tool we use to find a path through the configuration space, and is only an approximation to reality. In general, movement through a configuration space is not restricted to the edges of a graph; however, by using a graph to solve the problem we are doing exactly that. The main point here is that an optimal path with respect to the graph may be (and probably is) suboptimal with respect to the configuration-space. Therefore, we are interested in path length relative to space optimality—not graph optimality—because the former is more relevant to the real world.

Complicating matters is the fact that graph creation and path extraction are often combined in modern planning algorithms. Paths are dependent on the algorithm used to extract them from a graph, as well as the underlying properties of graph. The latter are dependent on the process used to build the graph. For example, the original RRT algorithm causes all edges to have a length less than r, for some $r > 0$. The same is true of RRT*, except edges are also further limited to be less than $\min(r, r_N)$, where N is the number of nodes in the graph, and r_N is a decreasing function of N.

C.2 Random path nomenclature

This section is intend to provide a quick reference for the variables we use in the rest of the paper. The reader may opt to skip this section, as each variable is also described as it is introduced in the text.

v_i is a node (vertex). $[v_i, v_j]$ is an edge between v_i and v_j, and has edge cost $c(v_i, v_j)$. The Euclidean distance between two nodes is $d(v_i, v_j)$. A graph G is defined by a set of vertices G_V and a set of edges G_E. n_i is the number of neighbors of v_i. The number of nodes in a graph is N.

A path P is defined as a connected sequence of $N_p = \ell_P + 1$ nodes (by ℓ_P edges). The best path, or one found using a particular algorithm, is P^* (context dependent). The best space path is S^* (and has $\ell^*_S + 1$ nodes). The total path cost is c_P. The best total path cost is c^*_P, and its expectation is $\mathrm{E}_{c^*_P}$. Analogous quantities exist for S^*. Often we will abbreviate c_P and c^*_P by c and c^*, respectively, when it is clear from context that we are talking about a path or a partial-path.

v_{start} and v_{goal} are the start and goal nodes, respectively, assuming that we are defining these as points.

v_S^* and v_G^* are nodes that are space-optimal and graph-optimal, respectively, where optimality is context dependent. $\vec{\mathbf{v}}_{i,j}$ is the vector from v_i to v_j, and $g(v_i, v_j) = \|\vec{\mathbf{v}}_{i,j}\|$ is its length (Euclidean distance). $\hat{\mathbf{v}}_S^*$ is a unit vector that points in the space-optimal direction (i.e., along $\vec{\mathbf{v}}_{i,goal}$ if there are no obstacles). ϕ measures angle away from $\hat{\mathbf{v}}_S^*$, $\phi_{i,j}$ is the ϕ as defined from v_i to its neighbor v_j. $\phi_{i,j}$ is shortened to ϕ_j when it is clear we are starting at v_i. ϕ^* denotes the best (minimum) ϕ_j.

When talking about a specific path, the shorthand $\mathbf{g}_i = \vec{\mathbf{v}}_{i,j}$ denotes the i-th edge in the path, and g_i its length (or $\mathbf{s}_i$ and s_i, if the path is space-optimal). Similarly, $\mathbf{o}_i$ is the projection of $\mathbf{g}_i$ onto the x axis, and o_i is the length of $\mathbf{o}_i$. When referring to concepts in general, edge length is denoted g, and the length of a best edge g^* (context dependent). o and o^* represent the x components of g, and g^*, respectively. $\mathrm{E}(g^*)$ and $\mathrm{E}(o^*)$ represent the expected values of g^* and o^*, respectively.

A particular algorithm (i.e., combination of graph creation and path extraction) is denoted A. The probability density of c using A is denoted f_c, while F_c is the corresponding distribution function. f_{c^*} is the probability density function of c^*. The expected value of c^* is E_{c^*}. Analogous functions and expectations can be found by substituting ϕ, $\sec\phi$, $\cos\phi$, etc. for c.

C is the configuration space, C_{free} is the obstacle free portion of C. The dimensionality of the configuration space is D. The x-axis is located along the D-th dimension. y is used to denote an axis perpendicular to the x-axis (context dependent). r is the maximum edge length and the radius of the D-ball B. The particular half of B for which $x > 0$ is denoted $B_{1/2}$. η is the number of non-v_i nodes in $B_{1/2}$. Φ is the hypersector described by B and ϕ. L is the distance between start and goal, where $L = d(v_{start}, v_{goal})$. Ψ is the Subset of $B_{1/2}$ bounded by the ellipsoid traced by v_i, where $c = d(v_{start}, v_i) + d(v_i, v_{goal})$. x_c is the x coordinate of the intersection of the ellipsoid with $B_{1/2}$. The Lebesgue measures of C_{free}, $B_{1/2}$, Φ, and Ψ are denoted $\lambda_{C_{free}}$, $\lambda_{B_{1/2}}$, λ_Φ, and λ_Ψ, respectively. The ratio of free space to total space is $\tau = \lambda_{C_f}/\lambda_C$.

ι_i denotes the runtime function of the node insertion function in terms of i nodes already in the graph. Time itself is denoted t. Various subscripts are added to ι_i and t to distinguish between

different runtime-order insertion functions (e.g., $\iota_{N,\text{linear}}$ or $t_{constant}$.

C.3 Outline of technique

In order to evaluate algorithmic properties, we would like to know how a particular algorithm A might be expected to perform in a particular environment over many trials. Let c quantify a particular property of the path returned by an algorithm (e.g., c might represent path length). Let f_c be the probability density function of c, assuming algorithm A is used in a particular environment. Given f_c, it is straightforward to calculate E_c the expected value of c for A in that environment [104]. Let F_c represent the corresponding distribution function of c. Note that f_c is the derivative of F_c with respect to c.

Often we are more interested in c^*, the best (as defined by the minimum or maximum) c. For example, the length of a shortest-path with respect to G, start, and goal. Let f_{c^*} denote the probability density function of c^*, over all graphs through a particular environment using algorithm A. The function f_{c^*} can be calculated from f_c and F_c using order statistics [11]. Therefore, E_{c^*} the expected value of c^* can also be found.

Given our assumption that sampling is uniform and i.i.d., the chance of picking a point out of any region of space is proportional to its Lebesgue measure (e.g., 'hypervolume'). This provides a straightforward way to calculate F_c in Euclidean space. Namely, $F_c = q$ is given by the Lebesgue measure of space associated with paths that have cost $c \leq q$.

The analytical technique that we present uses the geometry of an algorithm to calculate the expected value of $c*$. It is summarized as follows:

- Find F_c by analyzing the Lebesgue measure of space associated with $c \leq q$.
- Find f_c by differentiating F_c with respect to c.
- Use order statistics to find f_{c^*} from f_c and F_c.
- Find E_{c^*} using f_{c^*} and c^*.

RandomGraph()

```
1: Add v_start and v_goal to G
2: for i = 3 to N do
3:     add random point v_i to G
4:     for all v_j ∈ G s.t. d(v_i, v_j) ≤ r do
5:         add edge [v_i, v_j] to G.
```

GreedyPath(G, v_{start}, v_{goal})

```
1: v_i = v_start
2: while not(v_i = v_goal do
3:     j = arg min_j(φ_j) s.t. [v_i, v_j] ∈ G
4:     v* = v_j
5:     if φ_j ≤ π/2 then
6:         v_i = v*
7:     else
8:         no solution
```

Figure C.1: **RandomGraph**() constructs graph G, while **GreedyPath**() navigates G from v_{start} to v_{goal}, the start and goal nodes, respectively. The subroutine $d(v_i, v_j)$ returns the Euclidean distance between v_i and v_j. The quantity r represents the maximum edge length.

- Use E_{c^*} for simple algorithms to find bounds on E_{c^*} for more complex algorithms.

C.4 Expected quantities of simple algorithms

We now demonstrate the application of our method on a few simple examples. Assume a graph of N nodes exists in an obstacle-free, D dimensional, Euclidean configuration space (c-space) such that each node v_i is connected to all other nodes v_j for which $d(v_i, v_j) \leq r$, where r represents the maximum edge length, and v_{start} and v_{goal} are the start and goal nodes, respectively. This is the graph that would be created using algorithm **RandomGraph**() in Figure C.1.

Let a *graph*-optimal path be a path through the graph that minimizes the Euclidean distance traveled through the graph. Let a *space*-optimal path be a path through the configuration space that minimizes the Euclidean distance traveled through the configuration space. In general, these two types of optimality are not the same—while we would like to have a space-optimal path, we must settle for a graph-optimal path (since G is a subset of the entire c-space). A space-optimal

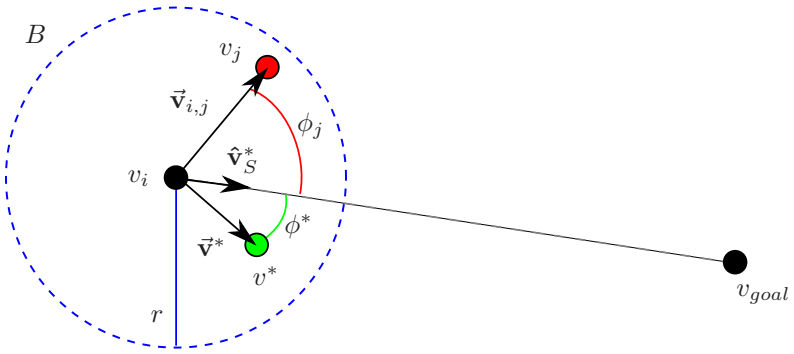


Figure C.2: v^* (green) is the 'best' (locally-optimal) neighbor of v_i, another neighbor is v_j (red). The unit vector $\hat{\mathbf{v}}_S^*$ points at the goal. Angles away from $\hat{\mathbf{v}}_S^*$ and vectors to v^* and v_j are also depicted. All neighbors of v_i are within D-ball B of radius r (blue).

path is the best graph-optimal path over all possible graphs through the configuration-space.

C.4.1 Expected angle to a desired heading E_{ϕ^*}

Assuming a graph has been created using **RandomGraph**(), we consider movement near a particular node v_i. We assume the goal is further than r from v_i so that it is not a direct neighbor, $d(v_i, v_{goal}) > r$. Let v_S^* denote a locally space-optimal neighbor of v_i. Since there are no obstacles, v_S^* is any neighbor of v_i that is along $\vec{\mathbf{v}}_{i,goal}$, the vector from v_i to v_{goal}. We shall refer to this direction as the locally-optimal heading, and denote it with the unit vector $\hat{\mathbf{v}}_S^*$ (See Figure C.2). Similarly, let the vector from v_i to its neighbor v_j be denoted $\vec{\mathbf{v}}_{i,j}$.

Assume that a greedy algorithm is used to extract a path from the graph, such that movement from v_i is to the neighbor v^*, where:

$$v^* = \arg\min_{v_j} \left(\arccos \left(\hat{\mathbf{v}}_S^* \cdot \frac{\vec{\mathbf{v}}_{i,j}}{\|\vec{\mathbf{v}}_{i,j}\|} \right) \right)$$

Where '$\cdot$' denotes the dot product and $\|\vec{\mathbf{v}}_{i,j}\|$ is the length of $\vec{\mathbf{v}}_{i,j}$. Let ϕ denote the angular distance away from $\hat{\mathbf{v}}_S^*$. Let ϕ_j denote the angle between $\hat{\mathbf{v}}_S^*$ and $\vec{\mathbf{v}}_{i,j}$, and let ϕ^* denote the angle between $\hat{\mathbf{v}}_S^*$ and $\vec{\mathbf{v}}^*$.

$$\phi^* = \min(\phi_j) = \arccos \left(\hat{\mathbf{v}}_{S*} \cdot \frac{\vec{\mathbf{v}}^*}{\|\vec{\mathbf{v}}^*\|} \right)$$

In other words, the greedy algorithm attempts to minimize ϕ, subject to the movement constraints

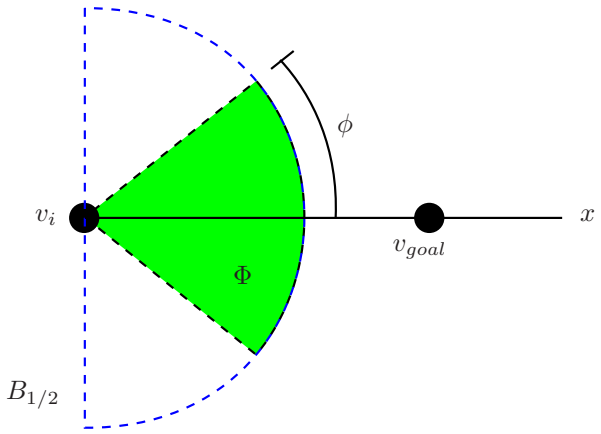


Figure C.3: Cross-section of problem. Φ is the hypersector (green) defined by ϕ measured from the x axis, the goal v_{goal} is located along the x axis, and v_i is located at $x = 0$. $B_{1/2}$ is the portion of the D-Ball located in the positive x direction.

imposed by graph edges.

C.4.1.1 Finding the distribution function F_ϕ

We would like to know E_{ϕ^*} the expected ϕ^*. Following our procedure outlined in Section C.3, the first step is to use our knowledge of the problem's geometry to find the distribution function F_ϕ.

We know that all neighbors of v_i exist within the D-ball of radius r centered at v_i. Let B refer to the D-ball. Note that a D-ball is the D-dimensional analog of a ball, it represents all points contained within a hypersphere of radius r. Without loss of generality, we assume that v_i is at the origin, and the optimal heading points along the D-th dimension, which we will also call the x-axis (see Figure C.3). We consider only the case where movement occurs in the positive x direction—In other words, if no neighbor of v_i is closer to the goal than v_i, with respect to the x-axis, then path extraction algorithm returns 'no solution.' Let $B_{1/2}$ refer to the half of the D-ball that exists in the positive x direction.

Note that ϕ is measured as the angular distance from the x-axis, where $0 \leq \phi \leq \pi/2$. Let Φ denote the hypersector of B that is bounded by the revolution of ϕ around the x-axis (e.g., if $d = 2$ then Φ is a sector and if $d = 3$ then Φ is a spherical cone). Let $\lambda_{B_{1/2}}$ and λ_Φ represent the

Lebesgue measure of $B_{1/2}$ and Φ, respectively. Since the probability a point is sampled from any particular region of space is proportional to the Lebesgue measure of that region, the distribution function F_ϕ is given by:

$$F_\phi = \frac{\lambda_\Phi}{\lambda_{B_{1/2}}} \tag{C.1}$$

From [60], we know an equation for $\lambda_{B_{1/2}}$ is given by:

$$\lambda_{B_{1/2}} = \frac{r^D \pi^{D/2}}{2\Gamma(D/2+1)}$$

where $\Gamma(\cdot)$ is the well-known gamma function. Note $\lambda_B = 2\lambda_{B_{1/2}}$. Similarly, [80] describes λ_Φ as follows:

$$\lambda_\Phi = \lambda_{B_{1/2}} I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right)$$

Where $I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right)$ is the regularized incomplete beta function $I_z\left(\frac{D-1}{2}, \frac{1}{2}\right)$ evaluated at $z = \sin^2\phi$.

$$I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right) = \frac{\mathsf{B}\left(\sin^2(\phi); \frac{D-1}{2}, \frac{1}{2}\right)}{\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right)}$$

Where $\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right)$ and $\mathsf{B}\left(\sin^2(\phi); \frac{D-1}{2}, \frac{1}{2}\right)$ are the corresponding beta function and incomplete beta function, respectively [96]. Substituting the integral form of the beta functions yeilds:

$$I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right) = \frac{\int_0^{\sin^2(\phi)} t^{(D-3)/2}(1-t)^{-1/2}dt}{\int_0^1 t^{(D-3)/2}(1-t)^{-1/2}dt}$$

When performing this calculation it is convenient to notice that ϕ ranges from 0 to $\pi/2$, with the consequences that $|\sin(\phi)| = \sin(\phi)$ and $|\cos(\phi)| = \cos(\phi)$.

C.4.1.2 Finding f_ϕ the pdf of ϕ

The probability density function of ϕ is given by:

$$f_\phi = F'_\Phi$$

where F'_Φ is the derivative of f_Φ with respect to ϕ.

d	E_{ϕ^*}	$E_{sec(\phi^*)}$	$E_{cos(\phi^*)}$
2	$\frac{\pi}{2(1+\eta)}$		
3	$\frac{(\eta-1)!!}{\eta!!}$ if η is odd $\frac{\pi}{2}\frac{(\eta-1)!!}{\eta!!}$ if η is even	$\frac{\eta}{\eta-1}$	$\frac{\eta}{\eta+1}$

Table C.1: Special cases of E_{ϕ^*}, $E_{sec(\phi^*)}$, and $E_{cos(\phi^*)}$

C.4.1.3 Finding $f_{\phi*}$ the pdf of $\phi*$

It is now possible to use order statistics to find the probability density function of $\phi*$.

Let C denote the configuration space, and let C_{free} denote the obstacle free portion of the configuration space. Let $\lambda_{C_{free}}$ denote the Lebesgue measure of C_{free}. Let η denote the number of nodes (other than v_i) in $B_{1/2}$. Nodes are uniformly random and i.i.d. in C_{free}, so η is expected to be:

$$\eta = N\frac{\lambda_{B_{1/2}}}{\lambda_{C_f}} - \frac{1}{2} \tag{C.2}$$

Where the negative 1/2 term accounts for the existence of v_i on the flat boundary of $B_{1/2}$ at $x = 0$.

Since ϕ^* represents the minimum ϕ over a set of size η, we are interested in the first order statistic of ϕ, given by:

$$f_{\phi^*} = \eta(1 - F_\phi)^{\eta-1} f_\phi \tag{C.3}$$

C.4.1.4 Finding E_{ϕ^*} the expected ϕ^*

The expected value for ϕ^* can now be computed as follows:

$$E_{\phi^*} = \int_0^{\pi/2} \phi f_{\phi^*} d\phi \tag{C.4}$$

This appears to be the simplest form of E_{ϕ^*} that is generally applicable over all D and η (due to the η exponent in Equation C.3). However, we have been able to find special cases for $D = \{2, 3\}$ that are applicable over all η. These appear in Table C.1. For $D > 3$ the integral form of $E_{\phi*}$ can be solved separately for each combination of D and η. As η gets large, we have found

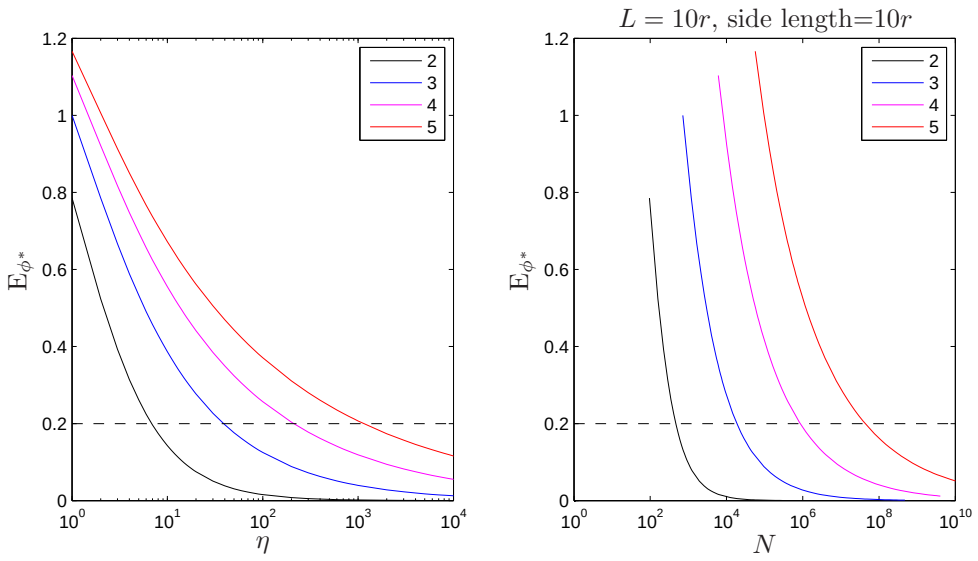


Figure C.4: Expected angle to desired heading E_{ϕ^*} vs. η and N (number of nodes in $B_{1/2}$ and graph, respectively), left and right, respectively. Different colors represent different configuration space dimensionality D.

numerical integration to become a practical necessity. Figure C.4-Left displays values of E_{ϕ^*} vs. η and $D = \{1, ..., 5\}$.

Figure C.4-Left shows us what to expect assuming that each $B_{1/2}$ contains η nodes. However, in practice the number of nodes N required to achieve a particular η is exponentially dependent on the dimensionality of the c-space (Further discussion is postponed until Section C.6). We plot E_{ϕ^*} vs. N in Figure C.4-Left assuming that the configuration space is bounded by a hyper-cube that spans $10r$ in each dimension.

C.4.2 Expected secant of angle to a desired heading $\mathrm{E}_{\mathrm{sec}(\phi^*)}$

Although E_{ϕ^*} (calculated in the previous section) is of academic interest, it is not the most useful quantity for evaluating algorithmic performance. A more useful quantity is $\mathrm{E}_{\mathrm{sec}(\phi*)}$, the expected secant of ϕ^*. $\mathrm{E}_{\mathrm{sec}(\phi^*)}$ is interesting because it represents the expected ratio between the distance that is actually traveled, and the component of that movement in the space-optimal direction of travel.

It is important to note that $\mathrm{E}_{\mathrm{sec}(\phi^*)} \neq \mathrm{sec}\left(\mathrm{E}_{\phi^*}\right)$ due to the functional non-invariance of the

expectation operator. In fact, we know $\mathrm{E}_{\sec(\phi^*)} \geq \sec\left(\mathrm{E}_{\phi^*}\right)$ from Jensen's inequality, and the convexity of the secant function on the range $[0, \pi/2]$.

Fortunately, $\sec(\phi)$ is monotonically increasing vs. ϕ on the range $\phi = [0, \pi/2]$. Since '*' is being used to denote the minimum value over a set of η values, it follows that:

$$\sec\left(\phi^*\right) = \sec\left(\phi\right)^* \tag{C.5}$$

In other words, the secant of the minimum angle in a set is the same as the minimum secant over all angles in the same set. In order to calculate $\mathrm{E}_{\sec(\phi^*)}$ we follow the same steps as in the previous section.

C.4.2.1 Finding the distribution function $F_{\sec(\phi)}$

Since $\sec(\phi)$ is a monotonically increasing function of ϕ over the range we are considering, 0 to $\pi/2$, the distribution functions for $\sec(\phi)$ and ϕ are identical.

$$F_{\sec(\phi)} = F_{\phi}$$

C.4.2.2 Finding $f_{\sec(\phi)}$ the pdf of $\sec(\phi)$

The probability density function of $\sec(\phi)$ is given by:

$$f_{\sec(\phi)} = F'_{\sec(\phi)}$$

where $F'_{\sec(\phi)}$ is the derivative of $F_{\sec(\phi)}$ with respect to ϕ. Note that $f_{\sec(\phi)} = f_{\phi}$.

C.4.2.3 Finding $f_{\sec(\phi^*)}$ the pdf of $\sec(\phi^*)$

It is now possible to use order statistics to find the probability density function of $\sec(\phi^*)$. From Equation C.5 we see that $f_{\sec(\phi^*)}$ is equivalent to $f_{\sec(\phi)^*}$, which is given by the first order statistic of $\sec(\phi)$ as follows:

$$f_{\sec(\phi^*)} = f_{\sec(\phi)^*} = \eta(1 - F_{\sec(\phi)})^{\eta-1} f_{\sec(\phi)}$$

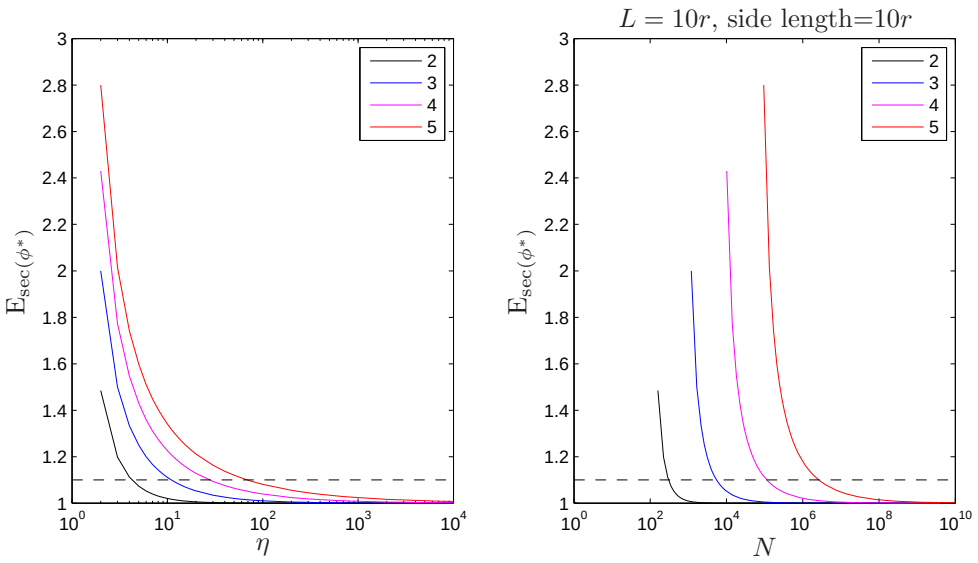


Figure C.5: Expected secant of angle to desired heading $\mathrm{E}_{\sec(\phi^*)}$ vs. η and N (number of nodes in $B_{1/2}$ and graph, respectively), left and right, respectively. Different colors represent different configuration space dimensionality D.

C.4.2.4 Finding $\mathrm{E}_{\sec(\phi^*)}$ the expected $\sec(\phi^*)$

The expected value of $\sec(\phi^*)$ over $\phi = [0, \pi/2]$ can now be calculated as follows:

$$\mathrm{E}_{\sec(\phi^*)} = \int_0^{\pi/2} \sec(\phi)\, f_{\sec(\phi^*)} d\phi \tag{C.6}$$

As with E_ϕ we are unable to simplify the integral forms of $\mathrm{E}_{\sec(\phi^*)}$ in a way that is generally applicable over all D and η. However, we have found a simplified version for the special case $d = 3$, which is presented in Table C.1. We also note that $\mathrm{E}_{\sec(\phi*)} = \infty$ for the special case $\eta = 1$ (for any D). Indeed, it is still possible to solve $\mathrm{E}_{\sec(\phi^*)}$ separately for each combination of D and η. In our personal experience, we have found numerical integration techniques to become necessary much earlier (with respect to D and η) than in the previous section. Values of $\mathrm{E}_{\sec(\phi^*)}$ for $D = \{1, ..., 5\}$ are plotted vs. η and N in Figure C.5-Left and -Right, respectively.

C.4.3 Expected cosine of angle to a desired heading $\mathrm{E}_{\cos(\phi^*)}$

Another interesting and related quantity is $\mathrm{E}_{\cos(\phi^*)}$, the expected cosine of ϕ. Although $\cos(\phi^*) = 1/\sec(\phi^*)$, we note that $\mathrm{E}_{\cos(\phi^*)} \neq 1/\mathrm{E}_{\sec(\phi^*)}$.

In this case, the function $\cos(\phi)$ is monotonically decreasing vs. ϕ on the range $\phi = [0, \pi/2]$. This mean that while '*' is being used to denote the minimum value with respect to ϕ, it will denote the maximum value with respect to $\cos(\phi)$. We again apply our procedure to obtain $\mathrm{E}_{\cos(\phi^*)}$.

C.4.3.1 Finding the distribution function $F_{\cos(\phi)}$

Since $\cos(\phi)$ is a decreasing increasing function of ϕ over the range we are considering, the distribution functions for $\cos(\phi)$ and ϕ must sum to one.

$$F_{\cos(\phi)} = 1 - F_{\phi}$$

C.4.3.2 Finding $f_{\cos(\phi)}$ the pdf of $\cos(\phi)$

The probability density function of $\cos(\phi)$ is given by:

$$f_{\cos(\phi)} = F'_{\cos(\phi)}$$

where $F'_{\cos(\phi)}$ is the derivative of $F_{\cos(\phi)}$ with respect to ϕ.

C.4.3.3 Finding $f_{\cos(\phi^*)}$ the pdf of $\cos(\phi^*)$

As before, we now use order statistics to find the probability density function of $\cos(\phi^*)$. Note that $f_{\cos(\phi^*)}$ is equivalent to $f_{\cos(\phi)^*}$, although '*' represents the minimum and maximum, respectively. The maximum value $f_{\cos(\phi)^*}$ is given by the η-th order statistic of $\cos(\phi)$ as follows:

$$f_{\cos(\phi^*)} = f_{\cos(\phi)^*} = \eta(F_{\cos(\phi)})^{\eta-1} f_{\cos(\phi)}$$

C.4.3.4 Finding $\mathrm{E}_{\cos(\phi^*)}$ the expected $\cos(\phi^*)$

The expected value of $\cos(\phi^*)$ over $\phi = [0, \pi/2]$ can now be calculated as follows:

$$\mathrm{E}_{\cos(\phi^*)} = \int_0^{\pi/2} \cos(\phi)\, f_{\cos(\phi^*)} d\phi \tag{C.7}$$

As with the other expected quantities, the integral form of $\mathrm{E}_{\mathrm{sec}(\phi^*)}$ cannot be simplified in general way for all D and η. A simplified version for the special case $d = 3$ is presented in

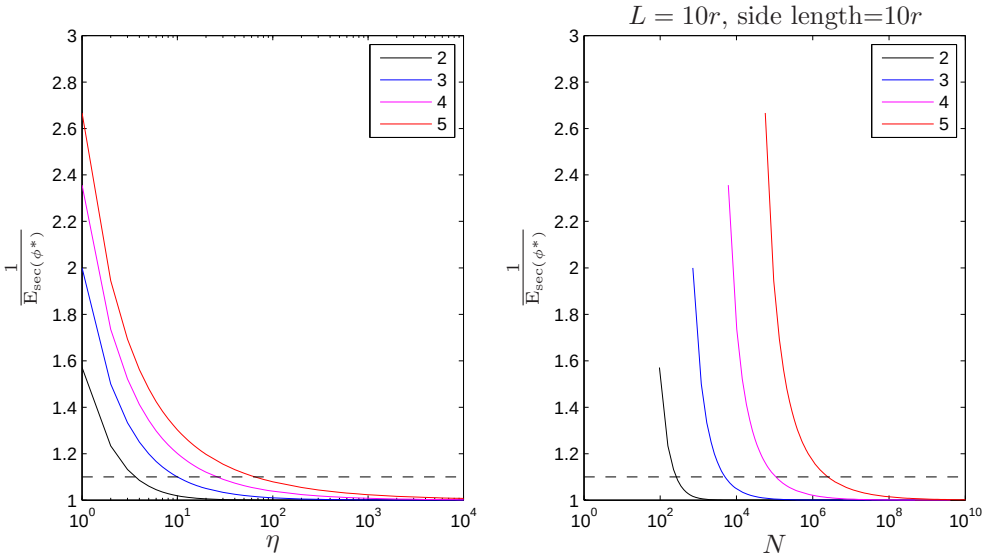


Figure C.6: The inverse of the expected cosine of angle to desired heading $\frac{1}{\mathrm{E}_{\cos(\phi^*)}}$ vs. η and N (number of nodes in $B_{1/2}$ and graph, respectively), left and right, respectively. Different colors represent different configuration space dimensionality D. Note that the *inverse* of $\mathrm{E}_{\cos(\phi^*)}$ is plotted.

PostProcess()

1: **for all** $[v_{start}, v_i] \in G$ **do**
2: add edge $[v_i, v_{goal}]$ to G.

Figure C.7: A post processing routine that links all neighbors of v_{start} to v_{goal}

Table C.1. In contrast to $\mathrm{E}_{\sec(\phi*)}$, $1/\mathrm{E}_{\cos(\phi*)}$ has finite values for $\eta = 1$ (for any finite D). As before, it is still possible to solve $\mathrm{E}_{\cos(\phi^*)}$ separately for each combination of D and η, and we find numerical integration necessary for moderately large D and η. Values of $1/\mathrm{E}_{\cos(\phi^*)}$ for $D = \{1, ..., 5\}$ are plotted vs. η and N in Figure C.6-Left and -Right, respectively. It is important to note that the inverse of $\mathrm{E}_{\cos(\phi^*)}$ is plotted (the reasons for this will become apparent in the next section).

C.4.4 Minimum bound on expected path length with 2 edges

Assume that a random graph is created using **RandomGraph**() in Figure C.1, and then the post processing procedure **PostProcess**() is used to connect all neighbors of v_{start} to the goal. Let c denote the length of any path between start and goal. Let c^* denote the minimum c over G. Running **PostProcess**() will never increase c^*. This means that any c^* found over the

post-processed graph is a minimum bound on the c^* of the original graph.

Lemma C.1: *After using* **PostProcess**(), *the best path in G contains (only) two edges:* $[v_{start}, v_G^*]$ *and* $[v_G^*, v_{goal}]$.

By construction, any path with c^* must contain v_{start}, v_{goal}, and at least one other node. Let $v_{G,1}^*$ denote the neighbor of v_{start} in the shortest path before post processing. Running post processing will not increase the graph distance between $v_{G,1}^*$ and v_{goal}. Further, it may cause paths through other neighbors of v_{start} to decrease such that the path with c^* goes through $v_{G,2}^*$ a different neighbor of n_{start} than $v_{G,1}^*$. Regardless, the best path must go through either $v_{G,1}^*$ or $v_{G,2}^*$ since **PostProcess**() only attaches neighbors of v_{start} to v_{goal}. □

We now apply our procedure to find E_{c^*} the expected length of the shortest path over all graphs created using **RandomGraph**() followed by **PostProcess**() in obstacle free Euclidean configuration spaces.

Without loss of generality, we make the same assumptions as in the previous three sections. Further, we assume that the start is further than r away from the goal, $d(v_{start}, v_{goal}) > r$ (this is done to ensure that the best path is non-trivial).

C.4.4.1 Finding the distribution function F_c

Let $L = d(v_{start}, v_{goal})$ be the Euclidean distance between the start and goal. The shortest graph distance between start and goal is c^*. Note that $c^* \geq L$ by the triangle inequality. By Lemma C.1 we know that

$$c = d(v_{start}, v_i) + d(v_i, v_{goal}) \tag{C.8}$$

where v_i is some neighbor of v_{start}. We also know two things about the geometry of the problem that can help us find E_c. The first is that v_i exist in the half D-ball defined by r (by construction of the first edge). The second is that Equation C.8 describes an ellipsoid in the configuration space with foci located at v_{start} and v_{goal} (See Figure C.8). The major semi-axis of the ellipsoid is along the x-axis, all minor semi-axes are orthogonal to the major semi-axis, and located along the axes

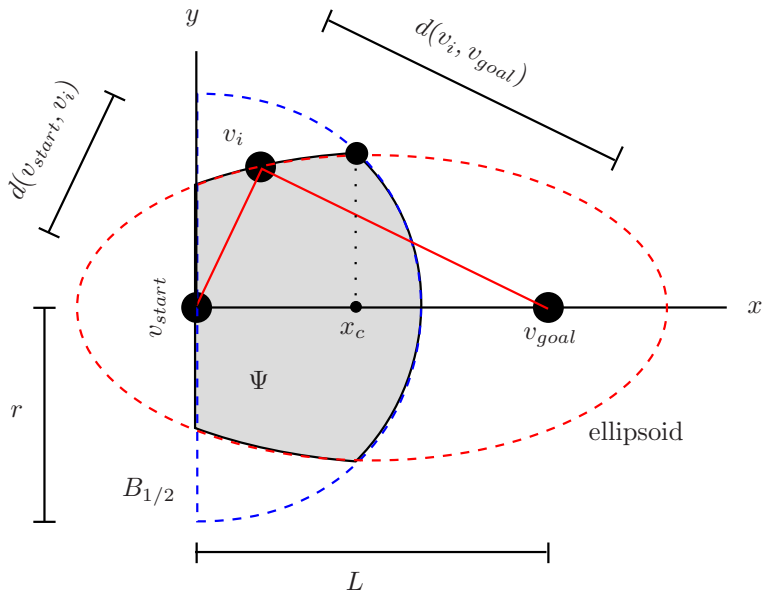


Figure C.8: Cross-section of problem. Ψ (grey) is the region defined by the intersection of $B_{1/2}$ with the region contained in the ellipsoid defined by $c = d(v_{start}, v_i) + d(v_i, v_{goal})$ with v_{start} and v_{goal} as its foci. Their respective bounding surfaces intersect at x coordinate x_c. L is the distance between v_{start} and v_{goal}. The radius of $B_{1/2}$ is r.

of the remaining dimensions.

Let the intersection of the space inside the ellipsoid with $B_{1/2}$ be denoted Ψ. Let λ_Ψ be the Lebesgue measure of Ψ. Assuming that L and r are constant, λ_Ψ is a monotonically increasing function of c. Therefore, F_c is given by:

$$F_c = \frac{\lambda_\Psi}{\lambda_{B_{1/2}}}$$

Ψ is symmetric about the x axis, and so a function for λ_Ψ is obtained by integrating the cross-section of Ψ along x. The integration must be performed in two steps to account for the different bounding surfaces of Ψ (i.e., due to the ellipsoid and $B_{1/2}$, respectively).

Let the x coordinate of the intersection manifold be denoted x_c. Given a particular c, the entire intersection manifold has the same x_c due to symmetry.

$$x_c = \frac{L^2 + 2cr - c^2}{2L}$$

Let y represent the perpendicular distance from the x axis to the bounding manifold at a particular

x. The Lebesgue measure of the cross-section is given by $\lambda_{B_{D-1,y}}$, where $B_{D-1,y}$ is the $(D-1)$-ball of radius y (i.e., D-ball in one lower dimension than D). The equation for y in terms of x and c will be different depending on if we are considering limits imposed by the ellipsoid or d-ball. We denote the former as y_1 and the latter as y_2, and their equations are:

$$y_1 = \frac{1}{2}\left(\left(L^2 - c^2\right)\left(1 - \frac{(c-2x)^2}{L^2}\right)\right)^{1/2} \tag{C.9}$$

and

$$y_2 = (r^2 - x^2)^{1/2} \tag{C.10}$$

respectively. Note that $\lambda_{B_{D-1,y_1}}$ and $\lambda_{B_{D-1,y_2}}$ are found by substituting Equations C.9 and C.10 for the radius variable in $\lambda_{B_{D-1}}$. For example, $\lambda_{B_{1,y_2}} = (r^2 - x^2)^{1/2}$, while $\lambda_{B_{2,y_2}} = \pi(r^2 - x^2)$ and $\lambda_{B_{3,y_2}} = \frac{4}{3}\pi(r^2 - x^2)^{3/2}$. The hyper-volume λ_Ψ is calculated as:

$$\lambda_\Psi = \int_0^{x_c} \lambda_{B_{D-1,y_1}} dx + \int_{x_c}^{r} \lambda_{B_{D-1,y_2}} dx$$

C.4.4.2 Finding f_c the pdf of c

The probability density function of c is given by:

$$f_c = F_c'$$

where F_c' is the derivative of f_c with respect to c. Note the derivative is taken with respect to c and not x.

C.4.4.3 Finding f_{c^*} the pdf of c^*

Order statistics can now be used find the probability density function of c^*. Since c^* is the smallest c in a set of size η, we care about the first order statistic.

$$f_{c^*} = \eta(1 - F_c)^{\eta-1} f_c$$

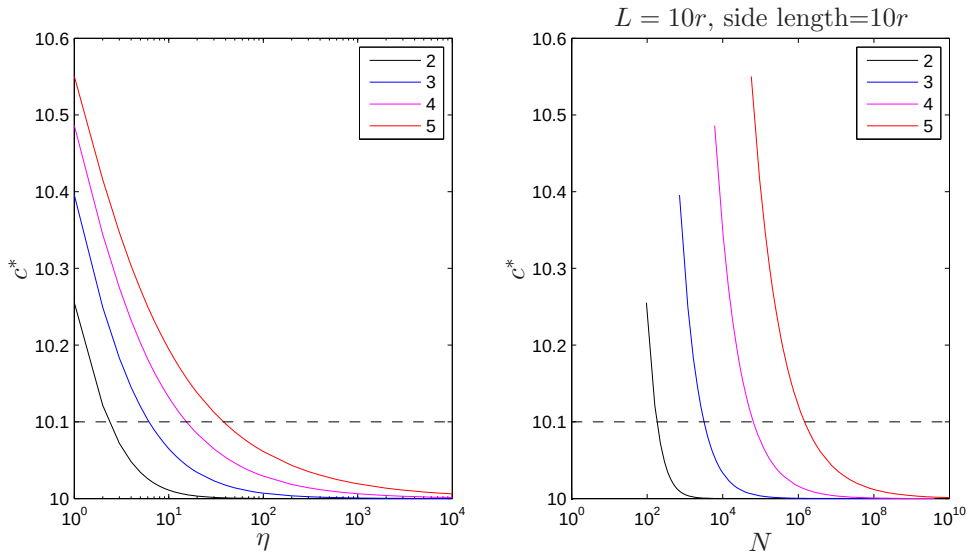


Figure C.9: The expected path length of a greedy 2-node path E_{c^*} vs. η and N (number of nodes in $B_{1/2}$ and graph, respectively), left and right, respectively. Different colors represent different configuration space dimensionality D. The distance between start and goal is $L = 10r$.

C.4.4.4 Finding E_{c^*} the expected c^*

By its definition F_c must range from 0 to 1. This happens when λ_Ψ is between 0 and $\lambda_{B_{1/2}}$, due to the geometry of the problem. Therefore, the limits of integration are given by the c that cause this to happen: L and $L + (r^2 + L^2)^{1/2}$, respectively. This can be understood intuitively as the minimum and maximum path lengths that can possibly exist given the problem that we have defined. L is the straight-line distance between start and goal, while $r + (r^2 + L^2)^{1/2}$ is the distance found by moving as far as possible (r) perpendicular to the desired direction of travel, before moving to the goal. We now have all the quantities necessary to find E_{c^*} the expected c^*:

$$\mathrm{E}_{c^*} = \int_L^{r+(r^2+L^2)^{1/2}} c f_{c^*} dc \tag{C.11}$$

We are unable to find a more simple form of Equation C.11. As with the other expected values we have found, Equation C.11 can be solved given particular values of r, L, η, and d. Again, numerical integration is a practice necessity. Figure C.9 depicts E_{c^*} vs. η and N for $d = \{1, ...5\}$ for the particular problem where $r = 1$ and $L = 10$.

C.4.5 Checking our work

To verify that our expressions for E_{ϕ^*}, $E_{\sec(\phi^*)}$, $E_{\cos(\phi^*)}$, and E_{c^*} are giving us reasonable values, we can check our work by performing Monte Carlo experiments using a particular d and η (and L in the case of E_{c^*}). In each trial we draw η points uniformly at random and i.i.d. from within $B_{1/2}$, and record the observed values of ϕ^*, $\sec(\phi^*)$, $\cos(\phi^*)$, or c^*. Averaging the results over t trials gives estimates of E_{ϕ^*}, $E_{\sec(\phi^*)}$, $E_{\cos(\phi^*)}$, and E_{c^*}, respectively. The Monte Carlo estimated values appear to approach values obtained using Equations C.4, C.6, C.7, and C.11, respectively, as $t \to \infty$. For example, the average values of ϕ^*, $\sec(\phi^*)$, $\cos(\phi^*)$, and c^* over t experiments are displayed in Figure C.10 top to bottom, respectively. For c^* we assume that $L = 10$.

C.5 Bounds on more complex algorithms

We begin with a few proofs that will help us later. Let g represent edge length, in general, and let g^* represent the edge length of the locally optimal edge (the one that is ϕ^* away from the desired direction of travel), in general. Let o and o^* represent the x components of g and g^*, respectively.

Lemma C.2: *The distributions of ϕ and g are statistically independent.*

Each half-hypersphere shell of $B_{1/2}$ is associated with a particular g, and has the same distribution of ϕ as any other half-hypersphere shell of $B_{1/2}$ associated with a different g. Therefore, all g have exactly the same distribution of ϕ. □

Corollary C.1: *The distributions of ϕ^* and g^* are statistically independent.*

Lemma C.3: *The distributions of $\cos(\phi)$ and g are statistically independent.*

Each g is associated with a particular half-hypersphere shell of $B_{1/2}$, and all shells have identical distributions of ϕ and $\cos(\phi)$. □

Corollary C.2: *The distributions of $\cos(\phi^*)$ and g^* are statistically independent.*

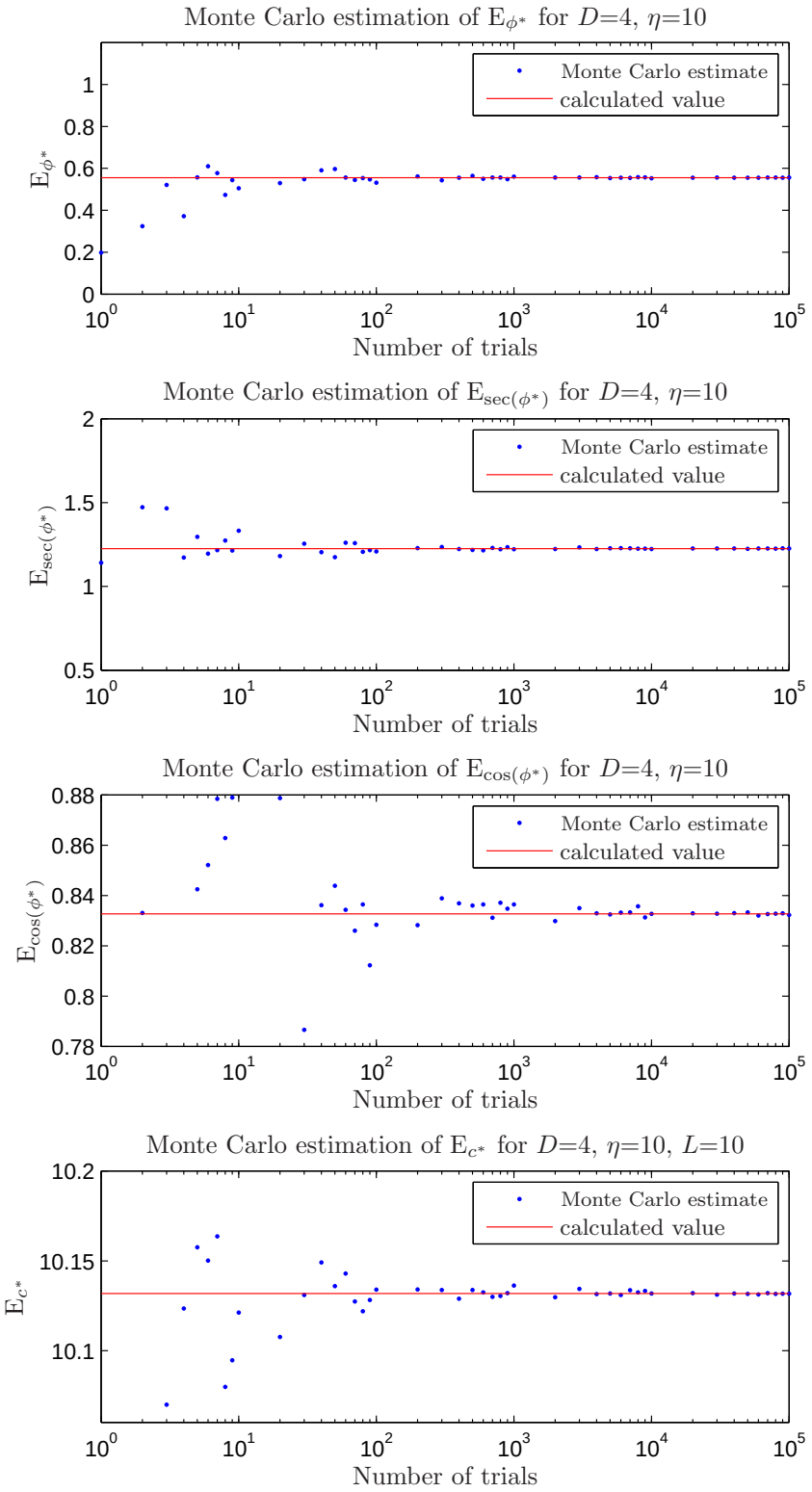


Figure C.10: Monte Carlo estimation of E_{ϕ^*}, $E_{\sec(\phi^*)}$, $E_{\cos(\phi^*)}$, and E_{c^*} (Top to Bottom, respectively) for $d = 4$ and $\eta = 10$ (and $L = 10$ for c^*). The horizontal axes denote number of trials.

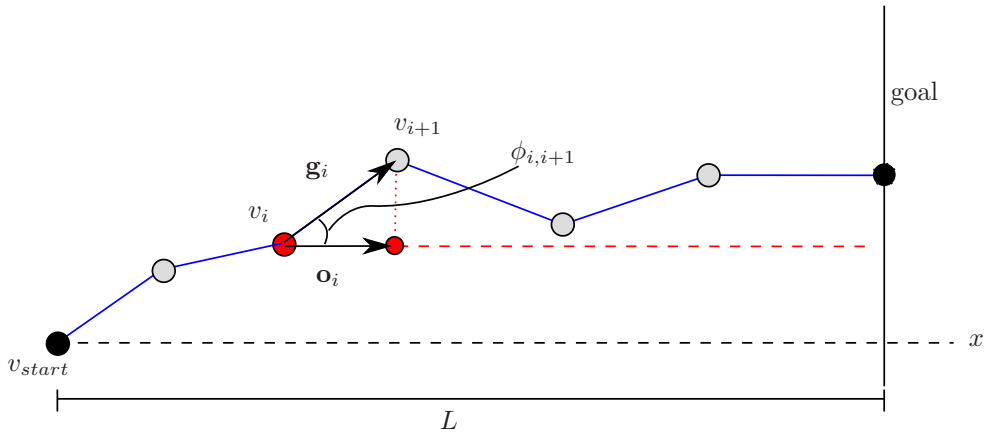


Figure C.11: Path between v_{start} and a goal defined by the plane at $x = L$. $\mathbf{g}_i$ is the vector from v_i to v_{i+1}, $\mathbf{o}_i$ is the projection of this vector onto the x axis.

Theorem C.1: $\mathrm{E}(g^* \cos(\phi^*)) = \mathrm{E}(g^*)\mathrm{E}(\cos(\phi^*))$.

The expectation operator supports multiplicativity between statistically independent variables. Corollary C.2 shows that g^* and ϕ^* are statistically independent. □

Corollary C.3: $\mathrm{E}(o^*) = \mathrm{E}(g^*)\mathrm{E}(\cos(\phi^*))$.

This follows directly from the definition of $\cos(\phi^*) = o^*/g^*$. □

C.5.1 Lower bound on greedy algorithm from point to plane in an obstacle free environment

Assuming a random graph has been created as in the previous section (by making two nodes neighbors if they are within r), we now investigate the particular problem of moving from a node v_{start} located at the origin to a goal that is defined by the hyperplane at $x = L$. In this case, both the globally and locally space-optimal directions of travel are parallel to the x axis. Therefore, the greedy algorithm moves to the neighbor that has ϕ^*, where ϕ^* measured as the angle between an outgoing edge and the x axis.

Let P^* represent the path found using the greedy algorithm, and let c_P^* be the cost of this path. We now calculate a lower bound on $\mathrm{E}_{c_P^*}$ the expected c_P^*. Although this can be done with

the use of E_{c^*} from Section C.4.4, the resulting bound becomes quite loose as L/r increases. We can achieve a much tighter bound by using $\mathrm{E}_{\cos(\phi^*)}$.

Let $\mathbf{g}_i$ represent the vector defined by the i-th edge $[v_i, v_{i+1}]$ along P^*. Let $\mathbf{o}_i$ represent the projection of $\mathbf{g}_i$ onto the x axis (See Figure C.11). Let g_i and o_i represent the magnitude of $\mathbf{g}_i$ and $\mathbf{o}_i$, respectively. $\phi_{i,i+1}$ is the angle between $\mathbf{g}_i$ and $\mathbf{o}_i$. Note that $\cos(\phi_{i,i+1}) = o_i/g_i$. Let ℓ_{P^*} be the number of edges in P^*. By construction we have:

$$c^*_P = \sum_{i=1}^{\ell_{P^*}} g_i$$

Taking the expectation of each side gives:

$$\mathrm{E}_{c^*_P} = \mathrm{E}\left(\sum_{i=1}^{\ell_{P^*}} g_i\right)$$

It is possible to move the expectation operator inside the summation due to its linearity property.

$$\mathrm{E}_{c^*_P} = \sum_{i=1}^{\ell_{P^*}} \mathrm{E}\left(g_i\right) \tag{C.12}$$

We now make three assumptions that we will discuss in-depth later. For now, assume that they hold. The first is that the expected $\phi_{i,i+1}$ of each edge except the last, is equal to the expected $\phi*$. That is $\mathrm{E}\left(\phi_{i,i+1}\right) = \mathrm{E}_{\phi^*}$, for $1 \leq i < \ell_{P^*}$. The second and third are similar, but for edge length g_i and o_i. Namely, $\mathrm{E}\left(g_i\right) = \mathrm{E}\left(g^*\right)$ and $\mathrm{E}\left(o_i\right) = \mathrm{E}\left(o^*\right)$, for $1 \leq i < \ell_{P^*}$. We assume that the last edge (i.e., $i = \ell_{P^*}$) is parallell to the x axis so that it moves toward the goal with the least ammount of length possible. This means that $\phi_{\ell_{P^*},goal} = 0$, and so $g_{\ell_{P^*}} = o_{\ell_{P^*}}$

Theorem C.2: *The expected path length from the greedy algorithm between a point and a plane is, with the previous assumptions,* $\mathrm{E}_{c^*_P} = \mathrm{E}(g_{\ell_{P^*}}) + \frac{L - \mathrm{E}(g_{\ell_{P^*}})}{\mathrm{E}_{\cos(\phi^*)}}$.

Using the second assumption along with Equation C.12, and moving the last edge outside the summation gives:

$$\mathrm{E}_{c^*_P} = \mathrm{E}(g_{\ell_{P^*}}) + \sum_{i=1}^{\ell_{P^*}-1} \mathrm{E}\left(g^*\right)$$

Substituting $\mathrm{E}\left(g^*\right) = \mathrm{E}\left(o^*\right) / \mathrm{E}\left(\cos(\phi^*)\right)$, by Corrilary C.3:

$$\mathrm{E}_{c^*_P} = \mathrm{E}(g_{\ell_{P^*}}) + \sum_{i=1}^{\ell_{P^*}-1} \frac{\mathrm{E}\left(o^*\right)}{\mathrm{E}\left(\cos(\phi^*)\right)}$$

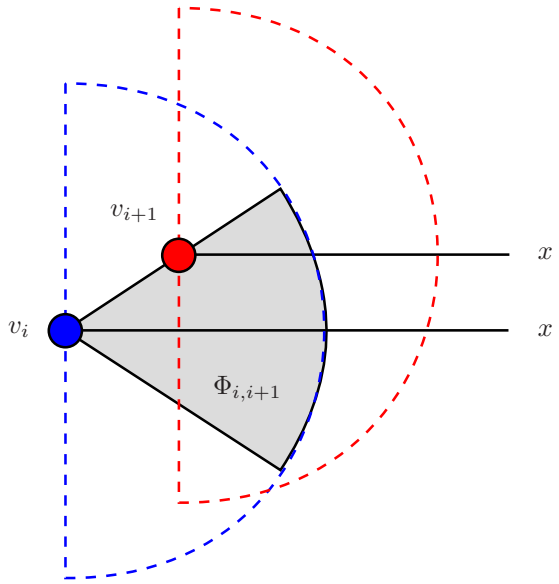


Figure C.12: Overlap between the $B_{1/2}$ of nodes v_i and v_{i+1}. By construction of the greedy algorithm, we know no nodes exist in $\Phi_{i,i+1}$ (grey).

Factoring $1/\mathrm{E}\left(\cos(\phi^*)\right)$, and substituting $\mathrm{E}\left(o_i\right) = \mathrm{E}\left(o^*\right)$ by the third assumption above gives:

$$\mathrm{E}_{c_P^*} = \mathrm{E}(g_{\ell_{P^*}}) + \frac{\sum_{i=1}^{\ell_{P^*}-1} \mathrm{E}\left(o_i\right)}{\mathrm{E}\left(\cos(\phi^*)\right)}$$

Realizing $\sum_{i=1}^{\ell_{P^*}-1} \mathrm{E}\left(o_i\right) = \mathrm{E}\left(\sum_{i=1}^{\ell_{P^*}-1} o_i\right) = L - \mathrm{E}(o_{\ell_{P^*}})$, by definition, and replacing $\mathrm{E}(o_{\ell_{P^*}}) = \mathrm{E}(g_{\ell_{P^*}})$, by construction, finishes the proof. □

Corollary C.4: $\mathrm{E}_{c_P^*} \geq r + \frac{L-r}{\mathrm{E}_{\cos(\phi^*)}}$

We know $\mathrm{E}(g_{\ell_{P^*}}) \leq r$. We also know that $\mathrm{E}_{\cos(\phi^*)} \leq 1$, which means that $\mathrm{E}(g_{\ell_{P^*}}) \leq \frac{\mathrm{E}(g_{\ell_{P^*}})}{\mathrm{E}_{\cos(\phi^*)}}$. Thus, by inspection we can see that replacing $\mathrm{E}(g_{\ell_{P^*}})$ with r leads to a lower bound on $\mathrm{E}_{c_P^*}$. □

We now examine whether or not it is fair to make the assumptions that lead to Theorem C.2. At first glance they seem reasonable, reflecting a belief that the local properties of nodes in the greedy-path are similar to the local properties of nodes in the entire graph. However, it turns out that this is not accurate. Consider Figure C.12. We see the two $B_{1/2}$ that are associated with neighboring nodes in P^*. By definition, the right-most point v_{i+1} is located along the best heading from the left-most node v_i. This means that we know no nodes exist within $\Phi_{i,i+1}$ defined by revolving $\phi_{i,j}$ around the x axis at v_i. Thus all three of our assumptions are violated! However, we see

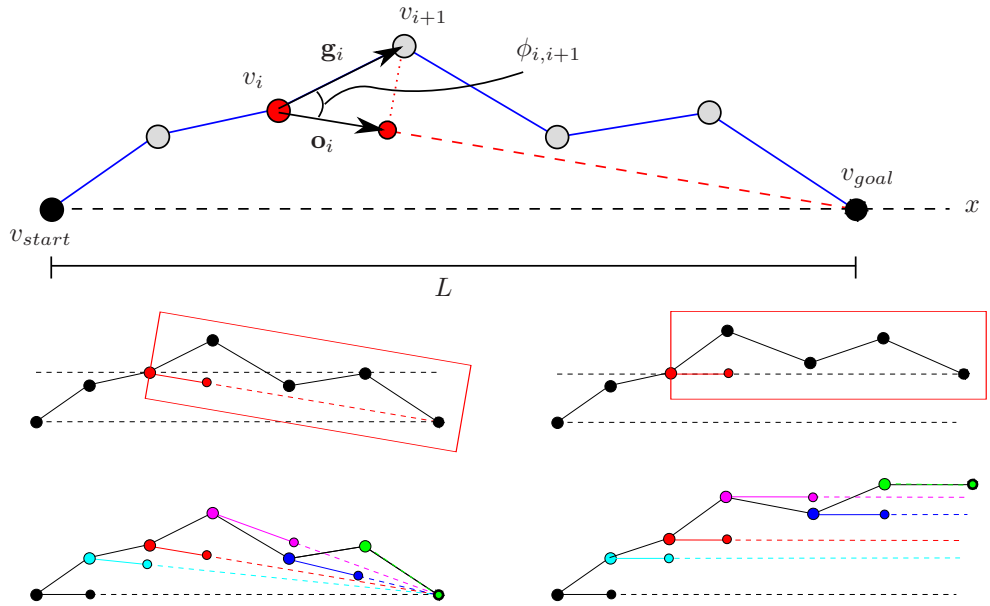


Figure C.13: Paths between v_{start} and v_{goal}. Top: $\mathbf{g}_i$ is the vector from v_i to v_{i+1}, $\mathbf{o}_i$ is the projection of this vector onto the locally optimal path (at v_{goal}). Middle: rotation of the sub-path between v_i and v_{goal} does not increase path length. Bottom: performing rotation for all nodes such that $\mathbf{o}_i$ is parallel to the x axis for all i.

that they are violated in a special way. Namely, for $1 < i < \ell_{P^*}$ we expect $\mathrm{E}(\cos(\phi_{i,i+1})) \leq \cos(\phi^*)$, with the inequality decreasing with D due to a higher concentration of volume near the surface of B. This means that we can expect paths to be longer than what is predicted by theorem C.2 (since dividing by $\mathrm{E}_{\cos(\phi^*)}$ instead of $\mathrm{E}(\cos(\phi_{i,i+1}))$ gives too small of a result. Therefore, we have the following theorem.

Theorem C.3: *A lower bound on the expected path length from the greedy algorithm between a point and a plane is given by* $\mathrm{E}_{c_P^*} \geq r + \frac{L-r}{\mathrm{E}_{\cos(\phi^*)}}$.

C.5.2 Lower bound on greedy algorithm from point to point

We now consider the case of planning between two points in an obstacle free environment. Without loss of generality, assume that both points are located on the x-axis and separated by L. In this case the locally optimal direction $\mathbf{o}_i$ from node v_i is directly at v_{goal}.

Lemma C.4: *Rotating the subpath* $v_i ... v_{goal}$ *around* v_i *will not change* $\sum_{i=1}^{\ell_{P^*}} o_i$.

Rotation of $\mathbf{g}_i$ does not change g_i. Since the goal is also rotated the quantity $\phi_{i,i+1}$ does not change, and so neighter does o_i. □ (see Figure C.13-middle)

Theorem C.4: *A lower bound on the expected path length from the greedy algorithm between two points is given by* $\mathrm{E}_{c^*_P} \geq r + \frac{L-r}{\mathrm{E}_{\cos(\phi^*)}}$.

This is a consequence of the triangle inequality and can be observed by examining a series of problems that are equivalent with respect to path length. Starting at node v_1 and working forward, each successive equivalent problem is obtained by rotating the rest of the path (including the goal) around node v_i (where $0 < i < \ell_{P^*}$) so that $\mathbf{o}_i$ is parallel to the x-axis, and points in the positive direction (see Figure C.13-bottom). Since rotations are performed around nodes, path length remains unchanged by Lemma C.4. Each rotation moves the goal in a non-decreasing manner with respect to the x-axis, so $\sum_{i=1}^{\ell_{P^*}} o_i \geq L$. By construction, we have modified the problem so that the desired direction of movement is parallel to the x-axis from any node v_i. This is exactly what was done when moving between a point and a plane (the requirement last edge moves directly to the goal parallel to the x-axis is met). Therefore, the lower bound described in Theorem C.3 is applicable, except that L must be replaced by $\sum_{i=1}^{\ell_{P^*}} o_i$. Using L instead of $\sum_{i=1}^{\ell_{P^*}} o_i$ gives a slightly looser lower bound. □

A more intuitive but less rigorous proof (that is also applicable to optimal paths) is that, movement to a single point constrains the problem more than moment to a plane. Since there are less 'good' movement options from any particular node, with respect to the goal, overall path length tends to increase.

C.5.3 Lower bounds with obstacles

So far we have considered only obstacle free configuration spaces. We now examine how adding obstacles can effect path length. Let M represent the number of nodes that we attempt to add to C. Nodes are invalid if they intersect with obstacles. $M \geq N$ since invalid nodes are not added to the graph.

Lemma C.5: *Adding obstacles (vs. no obstacles) will not decrease expected path length, with respect to M.*

This is true by the triangle inequality. Obstacles invalidate possible neighbors, with respect to a non-obstacle case. While paths may have increased length or become invalid, their length cannot be expected to increase. □

Given an environment with obstacles, paths fall into different homotopy classes depending on the relative direction they move around each obstacle. For example, in a 2D space, going above or below an obstacle produces two different homotopy classes. Each homotopy class has its own minimum path length associated with it. A path of minimum length, with respect to a particular homotopy class, can be described by placing nodes as close to the limiting surfaces of the obstacle as possible, and then connecting those nodes with edges.

Convergence to an optimal path can be broken into three separate categories as follows:

(1) Discovering paths in better and better homotopy classes, until a path is found in the homotopy class of the optimal path.

(2) Finding nodes that are closer and closer to the bounding surfaces of obstacles.

(3) Connecting distant nodes with more direct sub-paths.

In practice all of these things happen simultaneously.

This provides intuition as to how we might achieve a lower bound on the expected path length in obstacle environments, given a lower bound in obstacle free environments. Namely, assume that we are provided with the nodes on the limiting surfaces *a priori*, but that they are not—and can never be—directly connected. Next, we build the graph in the standard way. The starting nodes on the limiting surfaces can be connected via neighbors. If we re-sample the location of a starting node, then the new node is connected with distance 0 to that starting node. Note that the latter case effectively allows connections between a re-sampled starting node positions and other starting nodes that happen to exit within r of them.

This procedure removes the first two types of convergence, replacing them with *a priori* optimality. Thus, the only form of convergence that the new problem has is that of converging toward straight sub-paths between each seed node.

Lemma C.6: *Seeding the graph with nodes on the limiting surfaces will not increase the expected path length, vs. a non-seeded problem, with respect to M.*

This follows from the fact that we have removed two of the three convergence modalities and replaced them with optimality. □

Let S^* be the space optimal path that contains the fewest possible nodes Let ℓ_S^* be the number of edges in S^*. The number of nodes in S^* is $\ell_S^* + 1$. Let $\mathbf{s}_i$ be the vector between the i-th and $(i+1)$-th nodes in the space optimal path. Let s_i be the length of $\mathbf{s}_i$. The length c_S^* of a space optimal path is given by:

$$c_S^* = \sum_{i=1}^{\ell_S^*} s_i \tag{C.13}$$

Lemma C.7: *A lower bound on the expected sub-path found by the greedy algorithm between v_i and v_{i+1} in the new problem is given by $E(c_i) \geq r + \frac{s_i - r}{\mathrm{E}_{\cos(\phi^*)}}$*

First, we consider the reduced problem of planning between the two points v_i and v_{i+1} in obstacle free space. Using Theorem C.4, we obtain the lower bound $E(c_i) \geq r + \frac{s_i - r}{\mathrm{E}_{\cos(\phi^*)}}$. Next, Lemma C.6 allows us to use this as a lower bound for the case with obstacles. □

This leads to the following theorem.

Theorem C.5: *A lower bound on the expected path length from the greedy algorithm between two points in a Euclidean space is given by $\mathrm{E}_{c_P^*} \geq r\ell_S^* + \frac{c_S^* - r\ell_S^*}{\mathrm{E}_{\cos(\phi^*)}}$.*

Lemma C.6 tells us that a lower bound on the actual graph path is found by the new problem described above.

$$\mathrm{E}_{c_P^*} \geq \mathrm{E}\left(\sum_{i=1}^{\ell_S^*} c_i\right)$$

We move the expectation inside the summation (allowed by linearity of expectation), and replace $\mathrm{E}(c_i)$ according to Lemma C.7 to obtain a slightly looser bound:

$$\mathrm{E}_{c_P^*} \geq \sum_{i=1}^{\ell_S^*} \left(r + \frac{s_i - r}{\mathrm{E}_{\cos(\phi^*)}} \right) = r\ell_S^* + \sum_{i=1}^{\ell_S^*} \frac{s_i - r}{\mathrm{E}_{\cos(\phi^*)}}$$

We can factor the $1/\mathrm{E}_{\cos(\phi^*)}$, since it is the same for each term:

$$\mathrm{E}_{c_P^*} \geq r\ell_S^* + \frac{\sum_{i=1}^{\ell_S^*}(s_i - r)}{\mathrm{E}_{\cos(\phi^*)}} = \frac{\left(\sum_{i=1}^{\ell_S^*} s_i\right) - r\ell_S^*}{\mathrm{E}_{\cos(\phi^*)}}$$

Substituting by Equation C.13 completes the proof □.

C.6 Applications and implications

In the previous section we showed that for many different problems, a lower bound on the expected path length found by the greedy algorithm has the following form:

$$\mathrm{E}(c_P^*) \geq r\ell_S^* + \frac{c_S^* - r\ell_S^*}{\mathrm{E}_{\cos(\phi^*)}} \tag{C.14}$$

where c_P^*, is the length of the current path, c_S^* is the length of a space-optimal path that solves the problem, and $\mathrm{E}_{\cos(\phi^*)}$ measures the expected ratio between the lengths of the 'best' possible local edge and the best edge that we are able to find given the current graph. ℓ_S^* is the number of edges in the particular space-optimal with the least edges. In other words, $\ell_S^* - 1$ is the minimum number of times an optimal path must change heading to avoid obstacles. We now discuss possible applications and implications of this lower bound.

C.6.1 Estimation of optimal path length

Noting that $\mathrm{E}(c_S^*) = c_S^*$, we can rewrite Equation C.14 as follows:

$$\mathrm{E}(c_S^*) \leq r\ell_S^* + (\mathrm{E}(c_P^*) - r\ell_S^*)\mathrm{E}_{\cos(\phi^*)}$$

Substituting, $c_P^* \approx \mathrm{E}(c_P^*)$ gives an approximation to the upper bound on the expected optimal path length in terms of the current path length. This can be useful for estimating the expected minimum improvement if we continue planning forever. Although perhaps not as useful as an estimation for the maximum improvement, this can still be used as a possible stopping criterion.

C.6.2 Evaluation of algorithmic performance vs. D

As noted in the introduction, one of the main motivations for pursuing analytical algorithmic bounds is for improved understanding of how algorithmic performance is effected by the properties of the configuration space. This section is dedicated to doing this for D, η, and N. The only quantity in Equation C.14 that depends on these things is $\mathrm{E}_{\cos(\phi^*)}$. Therefore we now examine the behavior of $\mathrm{E}_{\cos(\phi^*)}$.

We begin by noticing that $\lambda_{B_{1/2}}$ cancels from λ_Φ in the distribution function F_ϕ. Therefore, F_ϕ is equal to the incomplete regularized beta function evaluated at $\sin^2(\phi)$.

$$F_\phi = I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right)$$

Taking the derivitive gives, for the range $0 \leq \phi \leq \pi/2$:

$$f_\phi = \frac{2\left(\sin^2(\phi)\right)^{(d-1)/2}}{\sin(\phi)\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right)}$$

As noted in Section C.4.3, $F_{\cos(\phi)} = 1 - F_\phi$. Which which implies $f_{\cos(\phi)} = -f_\phi$. Substituting into Equation C.7 and rearranging gives the following expression for $\mathrm{E}_{\cos(\phi^*)}$:

$$\int_{\pi/2}^{0} \frac{2\cos(\phi)\eta\left(1 - I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right)\right)^{\eta-1}(\sin(\phi))^{d-2}}{\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right)} d\phi$$

Note that we have switched the limits of integration to remove the negative sign introduced by $f_{\cos(\phi)}$

We desire to find Bachmann-Landau bounds of this quantity in terms of D (see [30] for details on Bachmann-Landau notation, the original German versions are [10, 72]). This is straightforward for $D = \{2, 3\}$, because the expression inside the integral reduces to a relatively simple formula in these cases. However, we would like to know a bound for all D. We observer that the main difficulty in evaluating the integral is the $\left(1 - I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right)\right)^{\eta-1}$ factor. However, for the range of ϕ that we are considering we know that $0 \leq \sin(\phi) \leq 1$, which implies that $0 \leq \sin^2(\phi) \leq 1$, and therefore $0 \leq I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right) \leq 1$. It follows that $0 \leq \left(1 - I_{\sin^2\phi}\left(\frac{D-1}{2}, \frac{1}{2}\right)\right)^{\eta-1} \leq 1$, for all η. Therefore we can replace the problematic factor with 1 to obtain an upper bound on the entire integral that is

applicable over all η.

$$\mathrm{E}_{\cos(\phi^*)} \leq \int_{\pi/2}^{0} \frac{2\eta \cos(\phi)(\sin(\phi))^{D-2}}{\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right)} d\phi$$

Now the integral can be performed with relative ease:

$$\mathrm{E}_{\cos(\phi^*)} \leq \frac{2\eta}{(D-1)\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right)} \tag{C.15}$$

Theorem C.6: $\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right) = \frac{k\sqrt{2}\sqrt{\pi}\sqrt{D}}{(D-1)}$, *where k is a constant.*

Using the gamma form of the beta function [9] leads to:

$$\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right) = \frac{\Gamma\left(\frac{D}{2} - \frac{1}{2}\right)\Gamma\left(\frac{1}{2}\right)}{\Gamma\left(\frac{D}{2}\right)}$$

By definition $\Gamma\left(\frac{D}{2} - \frac{1}{2}\right) = \sqrt{\pi}$. Substituting, and also multiplying by a convenient form of 1 gives:

$$\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right) = \frac{\left(\frac{D}{2} - \frac{1}{2}\right)}{\left(\frac{D}{2} - \frac{1}{2}\right)} \frac{\Gamma\left(\frac{D}{2} - \frac{1}{2}\right)\sqrt{\pi}}{\Gamma\left(\frac{D}{2}\right)}$$

By construction the gamma function satisfies the relation $(x-1)\Gamma(x-1) = \Gamma x$. Using this property and rearranging:

$$\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right) = \frac{2\sqrt{\pi}}{(D-1)} \frac{\Gamma\left(\frac{D}{2} + \frac{1}{2}\right)}{\Gamma\left(\frac{D}{2}\right)}$$

In [47] it is shown that $\frac{\Gamma(J+1/2)}{\Gamma(J)} = \sqrt{J}k$, where k is the sum of an asymptotic series. This allow is the following substitution:

$$\mathsf{B}\left(\frac{D-1}{2}, \frac{1}{2}\right) = \frac{2\sqrt{\pi}}{(D-1)} \sqrt{\frac{D}{2}} k$$

Rearranging finishes the proof. □

Substituting the result from Therorm C.6 into Equation C.15, and rearranging:

$$\mathrm{E}_{\cos(\phi^*)} \leq \frac{2\eta}{k\sqrt{2}\sqrt{\pi}\sqrt{D}}$$

This implies the following bounds:

$$\mathrm{E}_{\cos(\phi^*)} = O\left(\frac{\eta}{\sqrt{D}}\right)$$

and

$$\frac{1}{\mathrm{E}_{\cos(\phi^*)}} = \Omega\left(\frac{\sqrt{D}}{\eta}\right) \tag{C.16}$$

Recall that η is the number of nodes within $B_{1/2}$, the half D-ball of radius r. For algorithmic analysis purposes, we are more interested in how an algorithm behaves vs. N, the number of nodes in the graph. By Equation C.2 we see that η is related to N by the ratio $\lambda_{B_{1/2}}/\lambda_{C_f}$, which means that the relationships η vs. N and $\mathrm{E}_{\phi_{G*}}$ vs. N are dependent on both the radius r and the particular configuration space being used.

Recall that we assumed the radius of the D-ball was fixed at a particular r. In practice r is often defined to be small (e.g., with respect to the size of the c-space in any direction)—for now, we assume this is the case. Let B_{little} denote the largest D-ball that can fit in the c-space (ignoring obstacles), let $\lambda_{B_{little}}$ and r_{little} be the Lebesgue measure and radius of B_{little}, respectively. Let τ be the ratio of obstacle free c-space to total c-space, $\tau = \lambda_{C_f}/\lambda_C$. Note that by definition:

$$\lambda_{B_{little}} \leq \lambda_C$$

Dividing λ_{C_f} by either side gives:

$$\frac{\lambda_{C_f}}{\lambda_{B_{little}}} \geq \frac{\lambda_{C_f}}{\lambda_C} = \tau$$

Rearranging gives:

$$\tau\lambda_{B_{little}} \leq \lambda_{C_f}$$

which leads to the following inequality:

$$\frac{\lambda_{B_{1/2}}}{\tau\lambda_{B_{little}}} \geq \frac{\lambda_{B_{1/2}}}{\lambda_{C_f}}$$

Substituting in equations for volume, and simplifying:

$$\frac{1}{2\tau}\left(\frac{r_{edge}}{r_{little}}\right)^D \geq \frac{\lambda_{B_{1/2}}}{\lambda_{C_f}}$$

Substituting into Equation C.2:

$$\eta \leq \frac{N}{2\tau}\left(\frac{r_{edge}}{r_{little}}\right)^D - \frac{1}{2} \tag{C.17}$$

Similarly, Let B_{big} denote the smallest d-ball that completely surrounds the c-space.

$$\lambda_{B_{big}} \geq \lambda_C$$

Dividing λ_{C_f} by either side gives:

$$\frac{\lambda_{C_f}}{\lambda_{B_{big}}} \leq \frac{\lambda_{C_f}}{\lambda_C} = \tau$$

Rearranging gives:

$$\tau\lambda_{B_{big}} \geq \lambda_{C_f}$$

which leads to the following inequality:

$$\frac{\lambda_{B_{1/2}}}{\tau\lambda_{B_{big}}} \leq \frac{\lambda_{B_{1/2}}}{\lambda_{C_f}}$$

Substituting in equations for volume, and simplifying:

$$\frac{1}{2\tau}\left(\frac{r_{edge}}{r_{big}}\right)^D \leq \frac{\lambda_{B_{1/2}}}{\lambda_{C_f}}$$

Substituting into Equation C.2:

$$\eta \geq \frac{N}{2\tau}\left(\frac{r_{edge}}{r_{big}}\right)^D - \frac{1}{2} \tag{C.18}$$

Let $k_1 = r_{little}/r_{edge}$ and $k_2 = r_{big}/r_{edge}$. Given our assumption that r_{edge} is small, with respect to the c-space we know that $k_1 \geq 1$ and $k_2 \geq 1$. Substituting into Equations C.17 and C.18 leads to bounds on η as follows:

$$\frac{N}{2\tau k_2^D} - \frac{1}{2} \leq \eta \leq \frac{N}{2\tau k_1^D} - \frac{1}{2} \tag{C.19}$$

Alternatively:

$$\eta = O\left(\frac{N}{k_1^D}\right) \tag{C.20}$$

and

$$\eta = \Omega\left(\frac{N}{k_2^D}\right)$$

Combining Equation C.20 with Equation C.16 leads to the following:

$$\frac{1}{\mathrm{E}_{\cos(\phi^*)}} = \Omega\left(\frac{D^{1/2}k_1^D}{N}\right)$$

Combining with Equation C.14 gives a lower bound on the expected path length of the greedy algorithm.

$$\mathrm{E}(c_P^*) = \Omega\left(r\ell_S^* + (c_S^* - r\ell_S^*)\frac{D^{1/2}k_1^D}{N}\right) \tag{C.21}$$

Examining Equation C.21 we find a number of intuitive trends. As we might expect, increasing the number of nodes N tends to decrease path length. It is increasingly hard to find a good path as the number of dimensions increases. The factor ℓ_S^* can be thought of as a measure of obstacle congestion, and as ℓ_S^* increase, using large r becomes less advantageous. The next section is devoted to the study both r and the relationship between N and time.

C.6.3 Evaluation of algorithmic performance vs. time and r

By inspection of Equation C.21 we can see that the lower bound on $\mathrm{E}(c_P^*)$ decreases as r increases. This is due to the construction of scenario from which we constructed the lower bound. Specifically, because the last edge travels directly to the goal, or to each seed point in environments with obstacles . Obviously the graph is not seeded with optimal points in practice; however, assuming the current best path is located in the same homotopy class as the space-optimal path, there will still be nodes located near the corners of obstacles, and these will approach the optimal points as $N \to \infty$ (in fact, a result from [insert ref] shows that as N increases, the chances that a particular region of space is not sampled decrease exponentially). The main point here is that increasing r to enable direct connection between distant nodes will never increase path length, with respect to N, and it may decrease it.

Based on this logic, it seems natural to define $r = \infty$. However, there are logical reasons this is not done, and they differ depending on if we are considering a single or multi-query algorithm. For multi-query planners like PRM, the average branching factor of the graph is directly related to r. Therefore, increasing r can exponentially increase the time required to solve the optimal search problem. So although we will find better paths per query, each query will take longer.

The branching factor is less of an issue for single query planners like RRT and RRT*. This is because back pointers are updated as the graph is built, thus eliminate the need for post-processed graph search. However, in single query planners r actually effects the growth rate of N vs. time.

Focusing on single query algorithms, we believe that the most important evaluation metric is solution quality vs. time. New nodes are added via an insert function. Therefore, in order to

attach a new node v_i, one must first locate the appropriate neighbors (i.e., all other nodes within r of v_i). Using kd-trees this takes time $O(n_i \log N)$, where n_i is the number of potential neighbors we wish to locate. Modifying Equation C.2 to return the number of nodes in B instead of $B_{1/2}$, for a fixed r we expect n_i to be:

$$n_i = N \frac{\lambda_B}{\lambda_{C_f}}$$

Where $\lambda_B \leq \lambda_{C_f}$, and we assume that N is the number of nodes in the graph before adding v_i. Assuming that λ_{C_f} is static, and noting $\lambda_B = k_3 r^D$, where k_3 is a constant dependent on D, we get $n_i = N k_4 r^D$, where k_4 is a constant dependent on D. It is also important to note that $n_i \leq N$, since we can at most look at all of the nodes in the graph. Therefore, the insertion time per node for the graph building algorithm that we have been working with is:

$$\min(N, k_4 N r^D) \log N$$

We can immediately see why many algorithms actively try to keep r small. In particular, RRT* defines r such that the n_i is bounded in order to maintain the same order insertion time as RRT (which only needs to locate a single node per insertion).

To reiterate, for single-query planners, we have the following conflicting views about the ideal size of r.

- Larger r correlates to better path quality.
- Larger N correlates to better path quality.
- The growth of N is inhibited by the size of r.

As we have already mentioned, previous investigation into this problem by [rrt*] attempt to keep the runtime of the insertion function logarithmic. The underlying assumption here is that a logarithmic insertion function causes the best possible interaction between N and r, with respect to path quality. Based on our analysis in the previous sections, we believe there is evidence to support a contrary interpretation, at least for the greedy algorithm that we are currently investigating.

We now focus on the interplay between N, r, and $\mathrm{E}(c_P^*)$. The first step is to find the relationship between N and time. We assume that the insertion function has a runtime given by ι_N per node insertion, where ι_N is a function of the N nodes already in the graph.

The time directly after the first node is inserted is given by $t_1 = \iota_0$. The time directly after the second node is inserted is $t_2 = \iota_0 + \iota_1$. Following this pattern, the time t directly after the N-th node is inserted is:

$$t = \sum_{i=0}^{N-1} \iota_i \tag{C.22}$$

Therefore, for constant insertion time functions $\iota_{N,\mathrm{const}} = k$, we have:

$$t_{\mathrm{const}} = kN$$

In terms of runtime bounds we get:

$$t_{\mathrm{const}} = \Theta(N)$$

For linear insertion time functions $\iota_{N,\mathrm{linear}} = kN$ we get:

$$t_{\mathrm{linear}} = k\frac{N(N-1)}{2} = \Theta(N^2)$$

For logarithmic insertion time functions $\iota_{N,\log} = \log(N+1)$, where we add 1 so that $t \geq 0$ for $N \geq 1$, this is:

$$t_{\log} = \sum_{i=0}^{N-1} \log(i+1) = \log\left(\prod_{i=1}^{N} i\right) = \Theta(\log(N!))$$

And for the case of searching for a set of n neighbors, assuming that n is small enough that the set size is less than N (and being a bit sloppy with the fact that for $i < n$ we find only i neighbors). $\iota_{N,\mathrm{nlog}} = n\log(N+1)$

$$t_{\mathrm{nlog}} = \Theta(n\log(N!))$$

However, a more pertinent case for the greedy algorithm we have been considering is finding all neighbors within radius r. Again assuming that r is small enough that the set size is less than N

(and again being sloppy with $i < n$). $\iota_{N,\mathrm{r}} = k_4 N r^D \log(N+1)$, where $k_4 N r^D < 1$.

$$\begin{aligned} t_{\mathrm{r}} &= k_4 r^D \textstyle\sum_{i=0}^{N-1} i \log(i+1) = k_4 r^D \sum_{i=0}^{N-1} \log((i+1)^i) \\ &= k_4 r^D \log\left(\textstyle\prod_{i=1}^{N} i^{i+1}\right) = k_4 r^D \log\left(N! \prod_{i=1}^{N} i^i\right) \\ &= k_4 r^D \log\left(N! K(N+1)\right) \\ &= k_4 r^D \left(\log(N!) + \log(K(N+1))\right) \\ &= \Theta(r^D \log(K(N+1)) \end{aligned}$$

Where $K()$ represents the K-function. Note that $K(N+1) = \Omega(N!)$ and also $K(N+1) = o((N!)^N)$, where the later is a 'little-o.'

Solving the preceding runtime bounds for N in terms of the various t gives:

$$N = \Theta(t_{\mathrm{const}})$$

$$N = \Theta(\sqrt{t_{\mathrm{linear}}})$$

$$N = \Theta\left(\Gamma^{-1}(2^{t_{\log}})\right)$$

$$N = \Theta\left(\Gamma^{-1}(2^{t_{\mathrm{nlog}}/n})\right)$$

$$N = \Theta\left(K^{-1}\left(2^{t_{\mathrm{r}}/(r^D)}\right)\right)$$

Where Γ^{-1} and K^{-1} are the inverse Gamma- and K-functions, respectively. Although, it should be noted that simple closed forms (i.e., non-numerical solutions) of these two functions are unknown to the authors.

Regardless, one can obtain a theoretical lower bound on expected solution length vs. time, by substituting the appropriate form of N into Equation C.21 (depending on the type of insertion function used).

When studying the preceding expressions for N it is important to remember that they are all located somewhere on the spectrum between the constant and linear cases. Thus, despite their non-intuitive forms, $\iota_{N,\log}$, $\iota_{N,\mathrm{nlog}}$, and $\iota_{N,\mathrm{r}}$ will never take longer than $\iota_{N,\mathrm{linear}}$ case or less time than $\iota_{N,\mathrm{constant}}$ to build a graph of N nodes. One important assumption here is that r is small. An alternative assumption is that the insertion algorithm knows when $n \geq N$ (for $\iota_{N,\mathrm{nlog}}$) and when r

is larger than the environment (for $\iota_{N,\mathrm{r}}$), so that it can simply use a list of all nodes in those cases (i.e., instead of extracting them from the kd-tree and incurring an additional logarithmic cost per node).

Another observation is the runtime dependency that t_{r} has on r^D.

Turning now to look again at Equation C.21, but substituting $N = t(N)$ to represents the number of nodes in terms of the cumulative time required to insert them.

$$\mathrm{E}(c_P^*) = \Omega\left(r\ell_S^* + (c_S^* - r\ell_S^*)\frac{D^{1/2}k_1^D}{f(N)}\right) \tag{C.23}$$

Equation C.23 shows that a change in runtime order can have a significant effect on expected path quality. However, for all the insertion functions we have considered $\Theta(\sqrt{t_{\mathrm{linear}}}) \leq t(N) \leq \Theta(t_{\mathrm{const}})$, so although the benefit is significant, it is limited. On the other hand, we see that by increasing r until $r = c_S^*/\ell_S^*$, we get $\mathrm{E}(c_P^*) = \Omega(c_S^*)$, thus wiping out all other variables of the convergence lower bound.

It must be noted that much of this is due to our specific construction of the lower bound. Specifically, the fact that in the scenario we used to define it, we have removed two types of convergence and replaced them with optimality. Because the third type of convergence does not apply to the last edge, making that edge as long as possible can be beneficial. For this to apply to practical problems the goal must be sampled with more than 0 probability. Therefore, if a point goal is used, then it must be explicitly sampled—for instance, a small percentage of the time. This is already common practice in many algorithms.

We believe that the specific type of convergence that we have evaluate here is the one that tends to limit path quality the most in practice. However, we have no proof of this belief, as of yet, and hope in future work to evaluate how the other two types of convergence compare to the one we are currently examining.

Regardless, we are certain that the latter plays a significant role in the overall convergence—due to the fact that the overwhelming majority of all paths contain sub-paths that move between points unobstructed by obstacles. For this reason, we believe that the analysis provided can lead to

useful insight and better algorithm design. The next section is devoted to experimentally validating what we have obtained analytically.

C.7 Random path experiments

In order to verify that the predicted lower bound is correct, we perform experiments in which we run the greedy algorithm in $D = \{2...4\}$. The environment is euclidean and obstacle free, and each dimension spans 10 meters. We evaluate the lower bounds for edge lengths $r = \{1...10\}$. Regardless of dimension, the robot is required to move 10.5 meters. Note that this means that goal will not be a direct neighbor of the goal for the r that we are using. 50 runs of each parameter combination are run. Experimental results appear in Figure C.14

Although these plots are admittedly cluttered, the important thing to realize is that the predicted lower bounds are indeed less than the corresponding mean path lengths for all experiments. They are, in fact, lower than the mean minus one standard deviation in most cases (not shown). While the bounds are relatively loose, they do appear to track relatively well, and appear to become tighter as either r decreases or D decreases.

C.8 Random path discussion and conclusions

The main contribution of this paper is the introduction of an analytical method that can be used to calculate expected algorithmic performance. It is our hope that this tool will enable better algorithms to be developed and current algorithms to be more useful.

The method can be summarized as follows: use the geometry of a problem to find the distribution function F_c of the relevant quantity c, then use that to find the probability density function F_c. Assuming that we want to know about c^*, the best c^*, we can use f_c and F_c to compute the probability density function of f_{c^*}. This can be used to find the expected value of $c*$. Finally, we use the expected c^* to prove bound on more complex quantities.

Doing this allows us to answer question such as: how can we expect different algorithmic

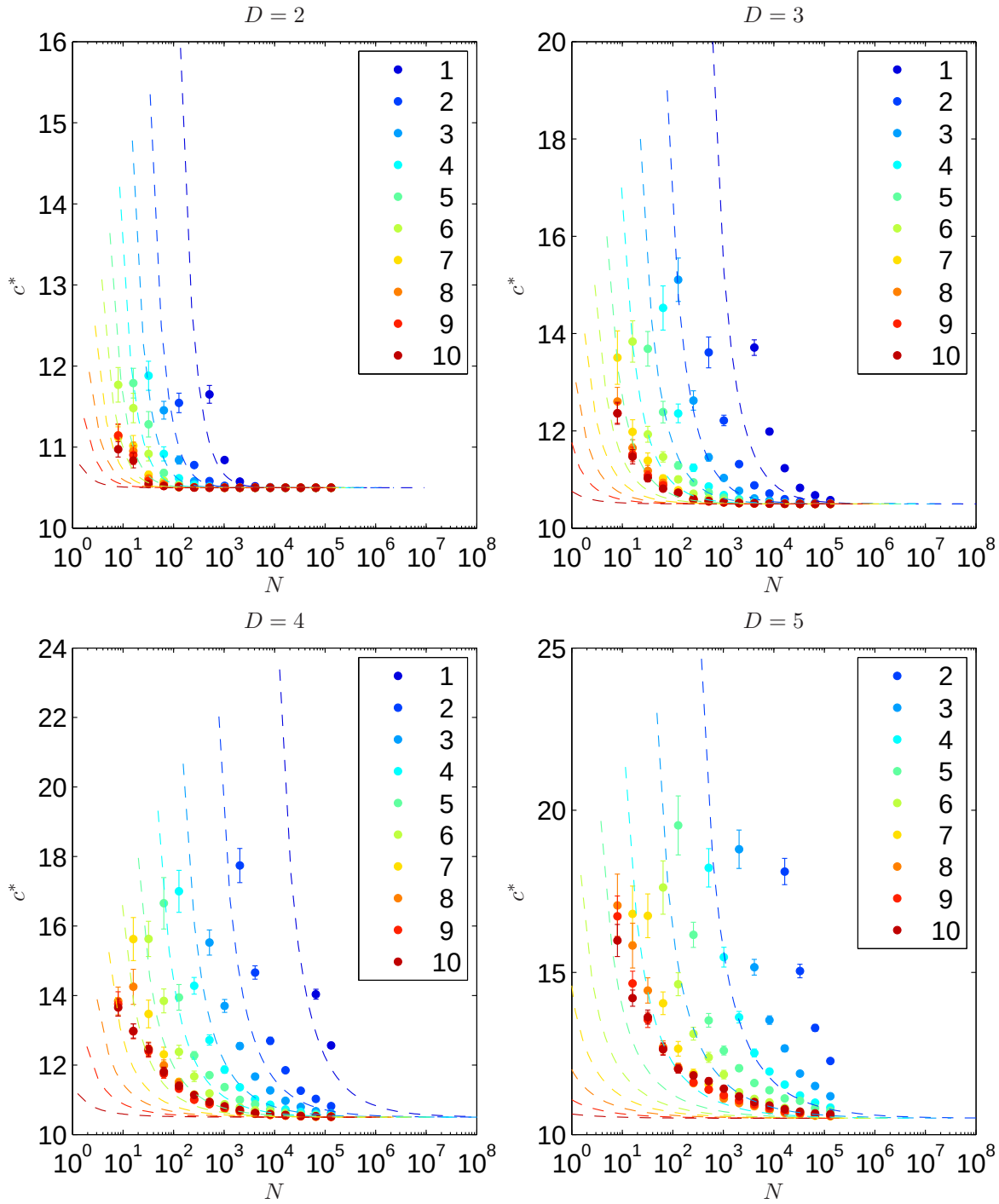


Figure C.14: Path lengths returned by the greedy algorithm in an obstacle-free environment. Different colors correspond to different maximum edge lengths r. Sub-figures correspond to dimensionality D. Mean and standard error over 50 runs appear as points and error bars, respectively. The calculated lower bounds appear as dashed lines. The start is 10.5 meters away from the goal. The predicted lower bounds are less than the actual path lengths (as expected). Points are not shown if less than half of the runs failed to find a solution.

parameters to perform vs. each other? and how might the dimensionality of the search space effect an algorithms performance?

In Section C.4.5 we use Monte Carlo techniques to verify that simple expected quantities we predict are accurate. In Section C.5 we use these simple quantities to prove bound on a greedy algorithm that always moves to the locally optimal neighbor—where local-optimality is defined as moving as close to the direction of the goal as possible. Finally, in Section C.7 we verify experimentally that the lower bound we predict are accurate.

Both the theory provided by our analytical method and the experiments that we perform show that, for the greedy algorithm we evaluate, performance tends to increase vs. decreasing maximum edge length r. This effect becomes more important in high dimensions, and is due to the fact that larger r allow the algorithm to short-cut unnecessary movement. This is a departure from previous work that attempts to keep r small to decrease insertion tome and accelerate graph growth. While we do find that adding more nodes to a graph tends to decrease path length, our analysis also suggests that by using the nodes we already have better, we can often achieve better paths than by simply adding more nodes to the graph.

The obstacle-free example we highlight in our experiments is more relevant than one might suspect at first glance, since it demonstrate a pure best-case scenario. In other words, the bounds we achieve on the obstacle free case are immediately applicable to cases with obstacles because the latter still require movement through free-space.

Finally, we note that in the course of our investigations we obtained a bound on a heretofore unknown quantity. In particular, the secondary effect that dimensionality has on path quality is exhibited by the $D^{1/2}$ factor in Equation C.14. This factor appears because adding more dimensions gives paths more of a chance to meander. We believe we are the first to quantify this effect for any algorithm, although it has been documented in the past.