Completing Manipulation Tasks Efficiently in Complex Environments

Christopher M. Dellin
September 30, 2016

CMU-RI-TR-16-53
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Siddhartha Srinivasa, CMU RI (Chair)
Anthony Stentz, CMU RI
Maxim Likhachev, CMU RI
Lydia Kavraki, Rice University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Copyright © Christopher M. Dellin

Abstract

An effective autonomous robot performing dangerous or menial tasks will need to act under significant time and energy constraints. At task time, the amount of effort a robot spends planning its motion directly detracts from its total performance. Manipulation tasks, however, present challenges to efficient motion planning. Tightly coupled steps (e.g. choices of object grasps or placements) allow poor early decisions to render subsequent steps difficult, which this encourages longer planning horizons. However, an articulated robot situated within a geometrically complex and dynamic environment induces a high-dimensional configuration space in which it is expensive to test for valid paths. And since multi-step plans require paths in changing valid subsets of configuration space, it is difficult to reuse computation across steps.

This thesis proposes an approach to motion planning well-suited to articulated robots performing recurring multi-step manipulation tasks in complex, semi-structured environments. The high cost of edge validation in roadmap methods motivates us to study a lazy approach to pathfinding on graphs which decouples constructing and searching the graph from validating its edges. This decoupling raises two immediate questions which we address: (a) how to allocate precious validation computation among the unevaluated edges on the graph, and (b) how to efficiently solve the resulting dynamic pathfinding problem which arises as edges are validated. We next consider the inherent tradeoff between planning and execution cost, and show that an objective based on utility functions is able to effectively balance these competing goals during lazy pathfinding. Lastly, we define the family motion planning problem which captures the structure of multi-step manipulation tasks, and propose a related utility function which allows our motion planner to quickly find efficient solutions for such tasks.

We assemble our algorithms into an integrated manipulation planning system, and demonstrate its effectiveness on motion and manipulation tasks on several robotic platforms. We also provide open-source implementations of our algorithms for contemporary motion planning frameworks. While the motivation for this thesis originally derived from manipulation, our pathfinding algorithms are broadly applicable to problem domains in which edge validation is expensive. Furthermore, the underlying similarity between lazy and dynamic settings also renders our incremental algorithms applicable to conventional dynamic problems such as traffic routing.

Acknowledgements

Thanks first and foremost to Sidd, whose advice and guidance were indispensable for this thesis – countless whiteboards replete with vertices and edges, $\mathcal{C}$'s and queues, each captured by a smartphone photo at the end of a meeting. I was also exceedingly lucky to have such a distinguished committee – Tony, Max, and Lydia provided not only an extraordinary body of work on which to build, but also an array of helpful comments and suggestions which greatly improved this dissertation.

Robotics research thrives on real hardware and real applications, and my experiences on the CHIMP and ARM-S projects at the NREC were pivotal to motivating and grounding my work. Thanks especially to Tony, Drew Bagnell, Clark Haynes, Mike Van de Weghe, Kyle Strabala, Jordan Brindza, David Stager, Eric Meyhofer, Brian Zajac, Sean Hyde, Jean-Sébastien Valois, Tom Galluzzo, Moslem Kazemi, and everyone else who made those projects such a fun and rewarding time for me.

I could not have asked for a better working environment (or a bigger desk) than what I found at the Personal Robotics Lab – but more importantly, the people are motivated, amazingly smart, and always available for a brainstorming session, technical discussion, code review, practice talk, or free lunch. Thanks especially to Mike Koval, Jen King, Aaron Johnson, and Shushman Choudhury for your help and critical comments, and to Clint, Gilwoo, Laura, Rachel, Pras, Pyry, Shervin, and Stefanos, for being such great colleagues and friends.

I am eternally grateful to my parents, who provided boundless love and support – and also very persistent encouragement while this dissertation on lazy coloring was in the pipeline. And to my fiancée Anca, I cannot imagine a more compassionate critic, a better friend, or a more dedicated partner. Thank you for giving me your enthusiastic support and your insightful advice. I certainly could not have finished without you!

This dissertation would not have been possible without funding from National Science Foundation IIS (#1409003), Toyota Motor Engineering & Manufacturing (TEMA), and the Office of Naval Research.

Contents

1 Introduction 9

1.1 Problem Characterization 9
1.2 Outline of Approach 10
1.3 Summary of Contributions 12
1.4 Review of Experimental Platforms and Problem Instances 13

2 Roadmap Methods for Motion Planning 15

2.1 The Motion Planning Problem 15
2.2 Motion Planning by Discretizing C-Space 17
2.3 Obstacle Sensitivity 19
2.4 Roadmap Classes 22

3 Fast Pathfinding on Graphs via Lazy Evaluation 25

3.1 The Shortest Path Problem 25
3.2 Lazy Shortest Path Algorithm 27
3.3 Edge Equivalence to A* Variants 30
3.4 Novel Edge Selectors 35
3.5 Experiments 38
3.6 Discussion 39

4 Incremental Bidirectional Search 43

4.1 Problem Definition 44

4.2 *Review of Pathfinding with Distance Functions* 45
4.3 *Incremental Bidirectional Search* 53
4.4 *Heuristic Search* 56
4.5 *Experimental Results* 62
4.6 *Available Implementations* 66

5 *Maximizing Utility in Motion Planning* 67
5.1 *Motivation and Related Work* 67
5.2 *Utility in Motion Planning* 70
5.3 *Marginal Utility on Roadmaps* 75
5.4 *Experiments* 80
5.5 *Discussion* 83

6 *Planning over Configuration Space Families* 85
6.1 *Related Work* 85
6.2 *Motivation: Families in Manipulation Tasks* 87
6.3 *The Family Motion Planning Problem* 92
6.4 *Approach: A Utility Model over Family Beliefs* 93
6.5 *Application: Multi-Step Manipulation Tasks* 97
6.6 *Implementation Details* 100

7 *Conclusion* 101
7.1 *Summary and Contributions* 101
7.2 *Future Directions* 105
7.3 *Lessons Learned* 115
7.4 *Concluding Remarks* 116

A *Appendix: LazySP Proofs and Timing Results* 119
A.1 *LazySP Proofs* 119
A.2 *LazySP Timing Results* 122

B Appendix: Incrementally Calculating Partition Functions on Graphs 125

C Appendix: IBiD Proofs 129

D Appendix: LEMUR Results 133

E Appendix: Family Motion Planning 135

List of Figures 137

List of Tables 147

F Bibliography 149

1

Introduction

Technology has automated an increasing variety of difficult, dangerous, or menial tasks previously performed by humans. Computer algorithms now trade our stocks, route our telephone calls and packages, and fly our planes, while simple machines clean our clothes and wash our dishes. Recent advances may soon enable real-world navigation applications such as autonomous automobiles and drones.

More complex tasks require robots with many degrees of freedom. Manipulation tasks, in particular, present challenges in many areas including perception, symbolic reasoning, and motion planning. Successful applications have so far been largely confined to large-scale manufacturing domains whose prescribed and structured environments allow these challenges to be overcome.

But what of applications such as home assistance, disaster response, and small-batch manufacturing? The robots of tomorrow will be required to plan high-dimensional motions in the face of geometrically complex and changing environments, and do so under significant resource constraints.

> *This thesis proposes an efficient motion planning approach well-suited to articulated robots performing recurring multi-step manipulation tasks in complex, semi-structured environments.*

We begin by characterizing the challenges inherent in motion planning for manipulation tasks. We then detail a concerted set of insights which inform our approach. The result is a collection of complementary algorithms which together outperform the current state-of-the-art.

1.1 Problem Characterization

Planning motions for articulated robots performing manipulation tasks presents a number of challenges, which we survey here.

A performant motion planner is only one component in a larger robotic system, which for many tasks must consider uncertainty, constraints, motion control, and error recovery.

High Dimensionality. The continuous configuration spaces induced by robotic arms often have higher dimensionality than typical vehicle navigation problems. Depending on the class of approach used, this manifests itself as high branching factors or costly nearest-neighbor queries, and calls for intelligent discretization schemes.

Expensive Validity Checking. The robot and its environment are geometrically complex. Homes and disaster areas are cluttered, and arms with revolute joints render corresponding configuration space obstacles intractable to consider explicitly. Further, since manipulation tasks require motions that begin and end close to collisions with objects, geometric models must have sufficiently high fidelity to disambiguate feasible paths from colliding ones. Testing candidate motions for collision therefore entails a large computational cost during planning.

Planning vs. Execution Effort. The robot must allocate its limited resources (e.g. time and energy) between planning motions and executing them. To maximize task efficiency, it is essential that motions be neither under-optimized (leading to costly execution) nor over-optimized (resulting in long planning pauses). This balance is especially important for tasks with or around humans, who are particularly intolerant of both unpredictable motion and planning pauses.

Semi-Structured Environments. The robot must continually generate planned motions in partially changing environments. This is especially evident when planning for manipulation tasks, where each object grasp and placement changes the collision-free subset of the robot's configuration space.

1.2 *Outline of Approach*

This dissertation develops an approach to motion planning for articulated robots which addresses these four challenges. We provide an outline of the thesis in Figure 1.1.

Roadmap Methods for Motion Planning (Chapter 2). We begin by reviewing the definition of the motion planning problem in terms of the robot's configuration space. We examine several existing approaches from the literature, including commonly-used sampling-

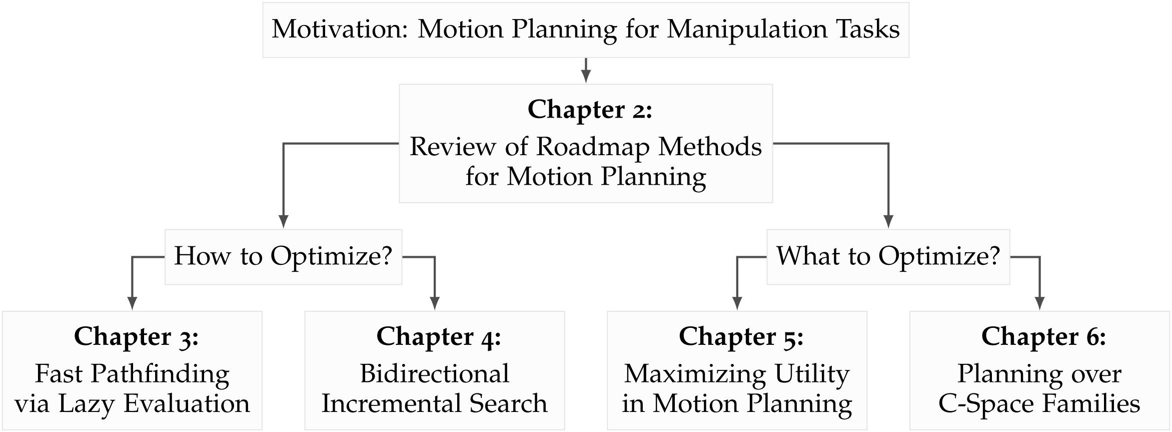

Motivation: Motion Planning for Manipulation Tasks
Chapter 2:
Review of Roadmap Methods for Motion Planning
How to Optimize?
What to Optimize?
Chapter 3:
Fast Pathfinding via Lazy Evaluation
Chapter 4:
Bidirectional Incremental Search
Chapter 5:
Maximizing Utility in Motion Planning
Chapter 6:
Planning over C-Space Families

Figure 1.1: Outline of the dissertation document. Motivated by recurring manipulation tasks, we consider two questions: (a) how to optimize a motion planning objective over a C-space roadmap, and (b) which objective best captures the task's planning/execution tradeoff?

based algorithms, and discuss their various completeness and optimality properties. We motivate our focus on the broad class of *roadmap methods* for solving the motion planning problem, which allows it to be solved via pathfinding on a sequence of progressively densified graphs with a particular choice of edge weight function. We also discuss the amenability of roadmap methods to caching and parallelization.

Fast Pathfinding on Graphs via Lazy Evaluation (Chapter 3). While roadmap methods effectively reduce the motion planning problem to one of graph search, the properties of the graph representation (in particular, that evaluating an edge's weight entails costly collision checks) motivates a lazy approach to pathfinding. Inspired by foundational algorithms such as D* [115] and the Lazy PRM [10], we introduce the *Lazy Shortest Path (LazySP)* class of algorithms, which relies on an inner incremental search algorithm and can leverage an edge weight estimator. Importantly, LazySP enables the a choice of *edge selector*, which strongly influences pathfinding performance. We introduce and compare various simple and novel selectors, and show equivalences with existing algorithms such as A* [47] and Lazy Weighted A* [20].

Incremental Bidirectional Search (Chapter 4). LazySP relies on an inner incremental search algorithm, such as DynamicSWSF-FP [98] or Lifelong Planning A* [67], to nominate candidate paths on each iteration. However, the choices of selector and estimator strongly influence the location of updated edges and the vertex heuristic that can be used. This motivates the development of a new search algorithm, IBiD, which combines bidirectional, heuristic, and incremental search into a single generalized algorithm.

Ch.	Problem	Algorithm	Description
3	Shortest Path (SP)	LazySP	Lazy search with edge selectors (induces an inner Dynamic SP problem)
4	Dynamic Shortest Path	IBiD	Incremental Bidirectional Dijkstra's algorithm
5	Motion Planning	LEMUR	Lazily Evaluated Marginal Utility Roadmaps (accepts a domain-specific cost model)
6	Family Motion Planning	Family LEMUR	Express a C-space family as a LEMUR cost model

Table 1.1: Table of four computational problems and the respective algorithms developed in this dissertation.

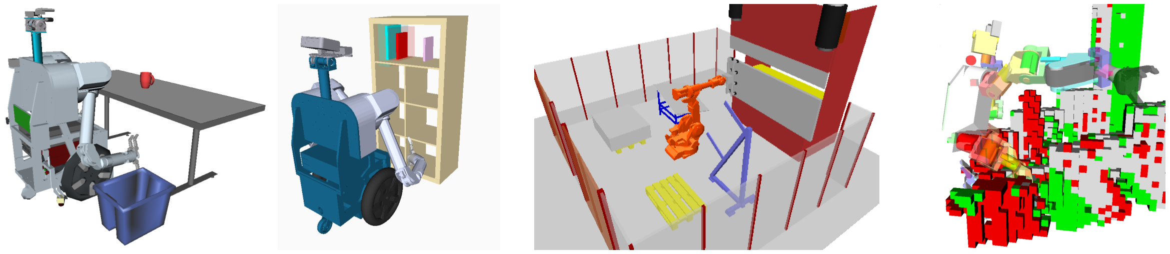
Maximizing Utility in Motion Planning (Chapter 5). Motion tasks expose a fundamental tradeoff between planning and execution cost. We show how conventional motion objective models can be augmented to include a planning cost term, and how such terms can be combined into a single measure of *utility*, which we can then optimize explicitly via particular weight and estimator functions using LazySP. The resulting algorithm, *Lazily Evaluated Marginal Utility Roadmaps (LEMUR)*, demonstrates improved performance across a range of motion planning queries when compared to state-of-the-art planners.

Planning over Configuration Space Families (Chapter 6). While utility maximization shows benefits in single-query settings, many motion tasks (such as manipulation problems) comprise multiple queries in a family of related configuration space subsets. We show how such problems can be formulated naturally via augmented cost models with custom planning cost components. We demonstrate improved performance in multi-step problem applications.

1.3 Summary of Contributions

We summarize the contributions of this dissertation here. A table of the developed algorithms is presented in Table 1.1.

- The LazySP pathfinding algorithm for graphs with expensive edge evaluations, along with accompanying novel edge selectors.
- The IBiD bidirectional, heuristic, incremental search algorithm.
- The LEMUR algorithm for maximizing utility in motion planning problems.
- A formulation of the C-space family motion planning problem, and an application of LEMUR to this formulation.



(a) The HERB home robot experimental platform.

(b) An ABB IRB 440 robot in an industrial workcell.

(c) The CHIMP disaster response robot.

Figure 1.2: The three robotic platforms considered in this dissertation.

- An experimental evaluation of the above algorithms against a selection of motion planning problems drawn from manipulation tasks.

- Open-source implementations[1] of the above algorithms for the Boost Graph Library (BGL) [108], Open Motion Planning Library (OMPL), [119] and OpenRAVE [28] software packages.

1.4 *Review of Experimental Platforms and Problem Instances*

We consider a number of motion and manipulation planning instances across three robotic platforms. The tasks performed by each robot are described in more detail along with the experimental results in Chapters 5 and 6.

- Figure 1.2(a): HERB, the Home Exploring Robot Butler [112]. HERB is composed of 7-DOF Barrett WAM arms mounted on a mobile base, and performs home tasks such as table clearing and retrieving objects.

- Figure 1.2(b): IRB 4400, a compact industrial robot from ABB. We consider a sheet metal bending application example in an industrial workcell, which is reproduced from the Lazy PRM paper [10].

- Figure 1.2(c): CHIMP, the CMU Highly Intelligent Mobile Platform [117]. We include results on planning data from the DRC Robotics Challenge Trials in December, 2013.

[1] `https://github.com/personalrobotics/lemur`

2

Roadmap Methods for Motion Planning

Robots are fundamentally agents that move through the world, and so the question of how to best deliberate about motion is a central problem in robotics. Different tasks and problem domains expose a vast array of qualities and properties of motion that are important – for example, motion that is expressive, efficient, or expected – and algorithms can rely on a rich and growing set of models and methodologies in order to generate such motion.

Underlying this rich tapestry of robotic motion is the most fundamental abstraction of motion planning – finding a path for a system which accomplishes a motion task without colliding with obstacles. While this might at first seem straightforward, finding such paths for articulated robotics in complex semi-structured environments is no easy feat, and finding such paths quickly and efficiently is of utmost importance.

This chapter includes a brief introduction to the motion planning problem. Due to its paramount importance to robotics, variations on this problem have deservedly enjoyed a considerable amount of attention over the past 40 years. We tailor our investigation towards aspects of the problem that are relevant to this thesis – for a comprehensive treatment, we refer to LaValle [71] and Choset et. al. [18]. In particular, we provide an overview of a class of approaches called roadmap methods that are well-suited to motion planning for articulated robots.

2.1 The Motion Planning Problem

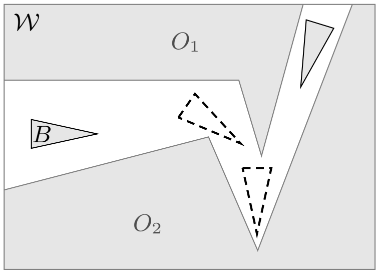


Figure 2.1: The original mover's problem [106] entails finding a collision-free path for a geometric body amongst obstacles, or finding that no path exists.

The earliest studies of motion planning considered a single rigid body moving within a Euclidean environment consisting of fixed geometric obstacles (Figure 2.1). Termed the *FindPath* or *piano mover's* problem, this representation and variations thereof were extensively studied [84, 106], for small dimensionalities (e.g. two or three) and obstacle representations (e.g. polygonal regions). Importantly, these earliest problems exhibited two fundamental properties inherent in all motion planning problems: (a) that solutions, called *motions* or *paths*, are continuous, and (b) that the fundamental feasibility objective is binary, with any prospective path either infeasible (in collision) or feasible (collision-free). The problem entails finding a feasible solution path if one exists, or returning with failure otherwise.

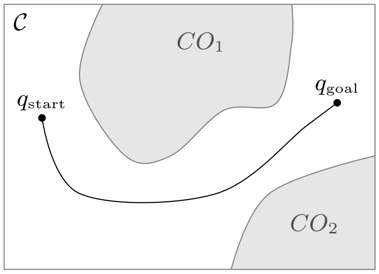


Figure 2.2: The motion planning problem entails finding a continuous path among obstacles in an abstract configuration space.

The Configuration Space. A generalization of the piano mover's problem commonly called the *motion planning problem* entails an abstraction of the single rigid body to an arbitrary *configuration space* (C-space) [83]. Any point q in this abstract space $\mathcal{C}$ (see Figure 2.2) corresponds to a full configuration of the system, such as the position and orientation of a rigid body. Importantly, $\mathcal{C}$ can also capture the full configuration (or joint) space of an articulated robotic manipulator.

Any point $q \in \mathcal{C}$ corresponds to a particular geometric configuration of the robot within its fixed environment. If this configuration results in a geometric collision in the workspace (either between the robot and the environment, or the robot with itself), this point lies within a *configuration-space obstacle*, e.g. CO_1 in Figure 2.2. The union of these configurations comprises the set of obstacle configurations $\mathcal{C}_{\text{obs}}$, and a configuration q is collision free if it is contained within the complement of this set $\mathcal{C}_{\text{free}} = \mathcal{C} \setminus \mathcal{C}_{\text{obs}}$.

Paths in $\mathcal{C}$. A continuous function $\xi : [0,1] \rightarrow \mathcal{C}$ of bounded variation is called a *path*, and let the set Ξ denote the set of such paths. We can then establish path validity:

$$\text{path } \xi \text{ is valid } \iff \xi(t) \in \mathcal{C}_{\text{free}} \; \forall t \in [0,1] \tag{2.1}$$

We can capture this definition for use in an optimization framework via the functional $x_{\text{valid}} : \Xi \rightarrow \mathbb{R}$ shown in Figure 2.3.

$$x_{\text{valid}}[\xi] = \begin{cases} 0 & \text{if } \xi(t) \in \mathcal{C}_{\text{free}} \; \forall t \\ \infty & \text{otherwise} \end{cases}$$

Figure 2.3: Path cost model for the (feasible) motion planning problem.

The Motion Planning Problem. Consider a single-pair motion planning query u consisting of start and destination vertices $q_{\text{start}}, q_{\text{dest}} \in \mathcal{C}$ as well as a collision-free subset $\mathcal{C}_{\text{free}}$. The motion planning problem consists of finding a valid path ξ^* with $\xi^*(0) = q_{\text{start}}$ and $\xi^*(1) = q_{\text{dest}}$ if one exists.

2.1.1 *Optimal Motion Planning*

$$x_{\text{int}}[\xi] = L_2[\xi]$$

Figure 2.4: Example dynamical path cost function for the optimal motion planning problem. Here, the arc length of ξ using the L_2 norm cost is used.

The *optimal motion planning problem* is a generalization of the motion planning problem which includes an additional objective over solution paths $x_{\text{int}} : \Xi \rightarrow \mathbb{R}$ intended to measure properties intrinsic to the path. Common choices for x_{int} include the path's arc length, the total time or energy necessary to execute the path, or the squared time derivatives averaged over the path. See [61] for a comprehensive treatment of sampling-based methods for the optimal motion planning problem.

An optimal motion planner typically considers both feasibility dynamical properties of candidate paths – this objective is simply $x_{\text{opt}} = x_{\text{valid}} + x_{\text{int}}$. We treat them separately because they exhibit fundamentally different computational profiles.

2.1.2 *Further Generalizations*

There are a great number of further generalizations of the motion planning problem that we will not explicitly consider. For example, the kinodynamic planning problem includes the derivatives of the configuration variables, and a solution to the kinodynamic problem is called a trajectory. The objective used in optimal variants of the kinodynamic problem often include terms which depend on the control inputs (e.g. torques) required to execute the trajectory.

Constrained motion planning constitutes a different generalization. A holonomic constraint on the configuration space, for example, induces a subset (often a manifold) of the configuration space on which planning is restricted. Various motion planners [7] have been proposed to accommodate such constraints.

2.2 *Motion Planning by Discretizing C-Space*

While the generalized mover's problem is computationally hard [99], a large array of algorithms have been proposed that perform well on a large range of typical instances. We give a brief review of approaches to the motion planning problem that rely principally on discretizations of the configuration space.

2.2.1 *Building Graphs in C-Space*

Testing Validity of Configurations. How can we establish the validity of a configuration q? This question is especially delicate when the geometry is complex and the configuration-space obstacles cannot be represented explicitly. In this case, it is necessary to test prospective paths for membership in $\mathcal{C}_{\text{free}}$ using a predicate that we express as a binary-valued *indicator function*,

$$\mathbf{1}_{\mathcal{C}_{\text{free}}} : \mathcal{C} \to \{\text{True}, \text{False}\} \quad \text{s.t.} \quad \mathbf{1}_{\mathcal{C}_{\text{free}}}(q) = \text{True iff } q \in \mathcal{C}_{\text{free}}. \tag{2.2}$$

Note that implementing $\mathbf{1}_{\mathcal{C}_{\text{free}}}$ for an articulated robot entails computing the forward kinematics of the robot at the query configuration, and conducting a collision check in the workspace.

Testing Validity of Paths. How can we establish the validity of a path ξ according to (2.1)? There are many approaches that can be applied, including C-space bubbles [97] and computing a path discretization resolution and corresponding workspace obstacle padding [4]. We will generally assume that a functional variant of the indicator function is available:

$$\mathbf{1}_{\Xi\mathrm{free}} : \Xi \to \{\mathrm{True}, \mathrm{False}\}. \tag{2.3}$$

The experiments in this document commit to a particular collision checking resolution for motion validity.

Motion Planning as Pathfinding. Importantly, two paths ξ_{ab} and ξ_{bc} with $\xi_{ab}(1) = \xi_{bc}(0)$ can be concatenated to a path ξ_{ac}, with

$$\mathbf{1}_{\Xi\mathrm{free}}(\xi_{ab}) \,\wedge\, \mathbf{1}_{\Xi\mathrm{free}}(\xi_{bc}) \iff \mathbf{1}_{\Xi\mathrm{free}}(\xi_{ac}). \tag{2.4}$$

This motivates approaches which maintain a graph structure of smaller valid motions in the configuration space. Consider a graph $G = (V, E)$, with each vertex $v \in V$ a configuration $q_v \in \mathcal{C}$, and each edge $e_{uv} \in E$ a path $\xi_e \in \Xi$ s.t. $\xi(0) = q_u$ and $\xi(1) = q_v$, and consider vertices $v_{\mathrm{start}}, v_{\mathrm{dest}} \in V$ corresponding to query vertices $q_{\mathrm{start}}, q_{\mathrm{dest}}$. Then a path p through G from v_{start} to v_{dest}, on which each edge e has $\mathbf{1}_{\Xi\mathrm{free}}(\xi_e) = \mathrm{True}$, corresponds to a valid solution to the motion planning problem.

Optimal Motion Planning as Pathfinding. Furthermore, if the intrinsic optimization objective x_{int} from Figure 2.4 is *additive*, then the optimal motion planning problem can be solved on G via a shortest path algorithm (at least up to the discretization afforded by the graph). This is accomplished by defining an edge weight function $w : E \to \mathbb{R}$ by $w(e) = x_{\mathrm{opt}}(\xi_e)$.

A wide variety of approaches exist in the literature for constructing, searching, and validating this graph for motion planning problems. We will broadly review a selection of different algorithms later in this chapter (Section 2.3).

Other Approaches to Motion Planning. The motion planning problem affords a number of different classes of algorithms, many of which do not construct graphs of local motions in the problem's configuration space. Potential field methods [64] and navigation functions [100] construct fields whose gradients can guide a path around local obstacles. Trajectory optimization approaches are especially well-suited to planning instances in which the basins of attraction are large. Approaches such as CHOMP [122] and TrajOpt [105]) commit to a trajectory representation for $\xi^{(1)}$ and use require stronger world descriptions (SDFs, convex obstacle decompositions) to make local modifications to produce a new path $\xi^{(2)}$. Also, optimizers are commonly used in lower levels to react to local changes (e.g. [97]).

2.3 *Obstacle Sensitivity*

Generally, graph-based motion approaches as described in Section 2.2.1 require the following three processes: (a) constructing the graph, (b) searching the graph, and (c) validating the graph. One of the key differentiators between the many graph-building techniques for motion planning is how they interleave these three processes. This section considers one important factor: the degree of dependence of the graph structure on the distribution of obstacles in the scene. We broadly categorize algorithms into two groups: those that are *obstacle-sensitive*, and those that are *obstacle-insensitive*.

2.3.1 *Obstacle-Sensitive Approaches*

In an obstacle sensitive approach, the graph is built incrementally, and construction of new elements (e.g. vertices and edges) is directly interleaved with validating those elements in response to the distribution of valid or costly states. These approaches might be best understood as constructing an approximation to $\mathcal{C}_{\text{free}}$ as directly as possible.

Exact Algorithms. The earliest work on the motion planning problem studied *exact* or *semi-algebraic* algorithms which worked directly on an explicit representation of the obstacles (e.g. polygons) in the configuration space [83]. This formulation of motion planning was shown to be PSPACE-hard [99, 16]. These approaches can guarantee optimal solutions, and may be even be able to guarantee the validity of certain edges by construction. However, they are difficult to apply to problems on articulated robots for two reasons. First, many approaches are only applicable to problems in two or three dimensions, or to robots with only translational degrees of freedom [62]. Second, an explicit representation of obstacles in the configuration space is exceedingly difficult to achieve due to the nonlinearity in the forward kinematics function. While some approaches are able to construct explicit analytical or approximate configuration space obstacles from workspace geometry (e.g. [88]), they are often only applicable to simple kinematics.

Treebuilding Algorithms. Many approaches do treat the configuration space implicitly (i.e. using indicator functions for validity checking as described in Section 2.2.1). The most common algorithms simultaneously grow and validate trees, either unidirectionally from the v_{start} or bidirectionally from both start and goal configurations. At each iteration, these algorithms use a sampling strategy to propose a new extension, and then augment their tree(s) if the new motion is found to be valid. Key examples of this approach include Expansive Space Trees (EST) [51] and Rapidly-exploring Random Trees (RRT) [70, 69]. The SBL planner [104] is a bidirectional variant of EST with lazy edge validity checking.

Planning vs. Execution Cost. The foundational algorithms in this category are primarily concerned with path *feasibility* – that is, the objective x_{valid} from Figure 2.3; there is often no mechanism explicitly biasing them to select low-cost paths. Therefore, solutions found tend to be of low quality, and they are customarily optimized using a path shortcutting algorithm in a post-processing phase before they can be executed. More recent work has also focused on asymptotically optimal variants of these planners [60, 61, 41, 48] which do address more general cost objectives.

We talk about this tradeoff between planning and execution cost in detail in Chapter 5.

Advantages of Obstacle Sensitivity. One key advantage to obstacle-sensitive approaches is that they only construct the graph structure in the parts of the configuration space that are relevant to the planning query. The Voronoi sampling bias of the RRT or the importance sampling of the EST attempt to focus graph construction in areas of the configuration space which tend to find feasible paths quickly. These techniques can also take account of the current distribution and connectivity of the discretization in order to focus new samples to their advantage (e.g. to address the narrow passage problem), such as visibility tests [110] or the expansion phase of the original PRM [63]. These heuristics can also be incorporated into sampling-based planners as obstacle-based [11] or hybrid [52] sampling strategies.

An additional advantage of these approaches is that they handle densification naturally. The graph structure is augmented automatically, so the resolution of the discretization need not be explicitly increased as the instance reveals itself to be more difficult.

2.3.2 *Obstacle-Insensitive Approaches*

In contrast to obstacle-sensitive approaches, the approaches we describe here commit to a particular discretization of the configuration space a priori, independent of the distribution of obstacles. While this simpler method forgoes some of the advantages of obstacle sensitivity discussed previously, this independence does confer some advantages of its own.

Roadmap Methods. The foundational obstacle-insensitive methods particular to the motion planning problem are called *roadmap methods*.

The term roadmap usually connotes a graph embedded in a continuous ambient space in the presence of obstacles. Vertices in the graph, which correspond to configurations in the configuration space, are also called milestones. The first roadmap methods such as the Probabilistic RoadMap (PRM) [63] initialized the milestones arrangement from a uniform distribution over the free space.

Key to roadmap methods to motion planning is the concept of the *local planner* which governs the validity of roadmap edges. Given two configurations $q_a, q_b \in \mathcal{C}$, the local planner considers a candidate path ξ_{ab} between them. (A common implementation simply uses the straight-line path in configuration space.) Edges are only attempted when they meet certain restrictions, such as distance with respect to some metric.

There are some variants of roadmap planners that are not independent of the obstacle distribution. First, many roadmap planners make use of a heuristic "expansion" step wherein additional samples are added to the roadmap in order to increase its connectivity. Second, edge connection rules that can forgo connections within the same connected component.

Search-based Methods. Graph pathfinding algorithms such as A* [47] are applicable to the motion planning problem if a suitable discretization of the continuous space is considered. While roadmaps can serve as this discretization, application of search-based methods often represents the space implicitly via a set of operators or motion primitives which reproduce a regular lattice over the configuration space [92]. When the vertices comprising the lattice are rectangular (also called a Sukharev point set [120]), search-based planning is also called "grid search." Such search methods can exploit a vertex heurstic, and many heuristics have been proposed for articulated motion planning – the most common of which compute a decomposition over the lower-dimensional workspace to guide the search. Some methods [93] exploit synergies between these different levels of planning to achieve improved efficiency.

Lazy Validity Checking. Because obstacle-insensitive approaches decouple graph construction from validity checking, it is often advantageous to defer the latter until it is necessary for solving the query at hand. Lazy validity checking has been exploited in roadmap methods such as the Lazy PRM [10, 48], search methods such as Lazy Weighted A* [20], and methods which bridge the two such as Fast Marching Trees (FMT*) [57] and Bidirectional FMT* (BFMT*) [114]. We discuss laziness in the context of graph search more comprehensively in Chapter 3

Adaptive Densification. One difficulty with these obstacle-insensitive approaches is that they commit to a particular discretization, and once their search is exhausted, they must return the best path found, or report failure if no path was found. This property is known as resolution completeness [17].

Many instances of prior work endeavor to progressively densify their discretization until a suitable solution is found. Early examples of this approach include hierarchical cell decompositions [34] and workspace and C-space bitmap pyramids for search over potential fields [4]. More recent asymptotically optimal motion planners such as Lazy PRM* [48]. and BIT* [41] take a similar approach to progressive densification, which we also adopt as described in Chapter 5 (see Figure 2.5).

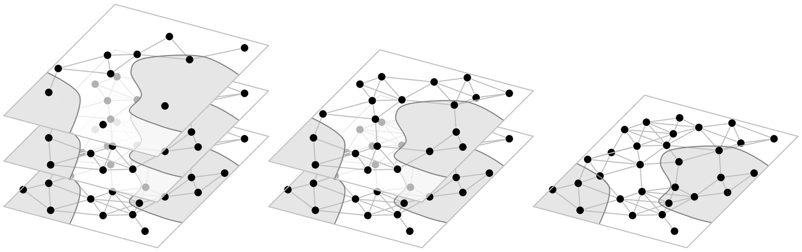


Figure 2.5: A stack of progressively densified roadmaps over a given configuration space $\mathcal{C}$. Each deeper roadmap constitutes a more accurate approximation to the true continuous problem space.

Advantages of Obstacle Insensitivity. The obstacle sensitivity question represents an underlying efficiency tradeoff. Sensitive approaches may be able to adapt their discretization more directly to local obstacle distributions. On the other hand, insensitivity to obstacles admits a number of efficiency advantages. Much of the nearest-neighbor computation often required during graph construction can be cached and amortized over all planning queries. Other properties of the discretization that are constant between similar planning instances can also be shared as described in Chapter 6. This also allows these approaches to be more easily parallelized. For these reasons, this thesis focusses on obstacle-insensitive roadmap methods.

2.4 *Roadmap Classes*

Roadmap methods operate over a graph constructed in the robot's configuration space. Roadmap milestones are commonly generated by a number of different types of point sequences, and edges are considered between vertices pairs that meet certain constraints. While most algorithms methods are generally agnostic to the class of roadmap that they operate on, choosing a suitable class and its parameters has a large effect on both theoretical and practical performance.

Random Sequences. The original roadmap algorithms [63] operated primarily over sets of vertices drawn uniformly at random from the configuration space. Many more recent algorithms such as RRT* [60], PRM* [61], FMT* [57], RRT$^{\#}$ [3], and BIT* [41] make use of the same uniform obstacle distribution. Such a distribution is attractive both because it is simple to implement it because it allows for theoretical properties such as completeness and optimality to be demonstrated in probability.

Deterministic Sequences. Many researchers have examined whether randomness is a necessary (or even beneficial) aspect of treebuilding and roadmap planners [13]. This is especially relevant in the context of comparing roadmap planners with search-based methods that conventionally operate over lattices [72].

One of the most straightforward disadvantages of using a non-deterministic sequence for each motion planning query is that the constructed graph is different for each query. This makes it impractical to pre-compute and amortize nearest neighbor queries across problem instances, as well as to pre-compute and cache edge states as described in Chapter 6.

Further, theoretical properties of roadmap methods such as resolution completeness and asymptotic optimality depend on properties of the underlying point set such as its *dispersion*. Not only can the dispersion of a set of randomly sampled points only be established in expectation, but it is demonstrably larger than the dispersion of other well-known point sequences, such as the Halton or Hammersley sets, or the Sukharev grid. For an in-depth analysis of different point sets for roadmap planning, see Janson et. al. [56].

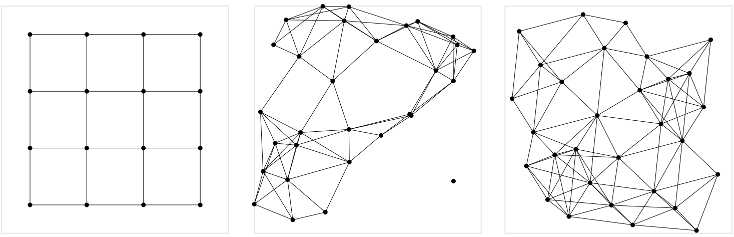


Figure 2.6: Examples of three roadmap types over the unit square: an axis-aligned lattice (left), random geometric graph (middle), and Halton graph with primes $(2,3)$ (right). All three roadmaps have a connection radius of 0.3. The latter two roadmaps have 30 vertices.

Connection Radii. Previous work [61] has established that a roadmap planner which uses a uniformly randomly generated point set is asymptotically optimal almost surely if the edge connection radius r is sufficiently large w.r.t. the number the number of samples n. In particular, it defines the critical value $\gamma^*_{\mathcal{C}}$:

$$\gamma^*_{\mathcal{C}} = 2 \left(\left[1 + \frac{1}{d} \right] \frac{\lambda_d(\mathcal{C})}{\zeta_d} \right)^{1/d}, \tag{2.5}$$

where d is the dimensionality of the configuration space, $\lambda_d(\cdot)$ represents the Lebesgue measure (e.g. volume), and ζ_d is the Lebesgue measure of the d-dimensional unit ball. Note that while other publications consider n to be the number of milestones in the collision-free subset of the space (and therefore also rely on the measure of that subset), we measure the volume of the entire C-space, and let n correspond to the total number of vertices. We do this because we will be determining the validity of roadmap edges in a separate step, as described in Chapter 3.

Asymptotic optimality under uniform sampling requires an edge connection radius $r_{\log}$:

$$r_{\log}(n) = \gamma^*_{\mathcal{C}}\, \eta \left(\frac{\log(n)}{n} \right)^{1/d} \tag{2.6}$$

for some tuning parameter $\eta > 1$. Point sets with tighter dispersion bounds, such as the Halton sequence, can exploit a smaller connection radius r_{loglog}, as described in detail in [56]:

$$r_{\mathrm{loglog}}(n) = \gamma_{\mathcal{C}}\, \eta \left(\frac{\log(\log(n))}{n} \right)^{1/d}. \tag{2.7}$$

Roadmaps for Articulated Robots. We apply roadmap methods to the three robot platforms from Chapter 1 in this thesis. We build roadmaps directly in the configuration space of the robot. See Table 2.1 for the roadmap parameters that we use.

	HERB	CHIMP	IRB 4400
d	7	7	6
$\gamma^*_{\mathcal{C}}$	7.67	7.97	7.74
$r_{\log}$	2.83	2.93	2.41
r_{loglog}	2.31	2.40	1.90

Table 2.1: Table of roadmap connection radii parameters for various scaling rates across the different robot platforms considered in this thesis. Radii presented are for $n = 10000$ and $\eta = 1$, and are given in radians.

3

Fast Pathfinding on Graphs via Lazy Evaluation

The roadmap methods described in Chapter 2 create a discretization of the configuration space using a graph $G = (V, E)$. This allows the motion planning problem to be solved by way of a pathfinding algorithm on G – and there are a large variety of such algorithms available in the literature to choose from. However, the computational efficiency of suitable algorithms depends intimately on the underlying problem domain.

In this chapter, we consider the general shortest path problem with a particular focus on domains (such as robot motion planning) where evaluating the edge weight function dominates algorithm running time. Inspired by lazy approaches in robotics, we define and investigate the *Lazy Shortest Path* class of algorithms which is differentiated by the choice of an *edge selector* function. We show that several algorithms in the literature are equivalent to this lazy algorithm for appropriate choice of this selector. Further, we propose various novel selectors inspired by sampling and statistical mechanics, and find that these selectors outperform existing algorithms on a set of example problems.

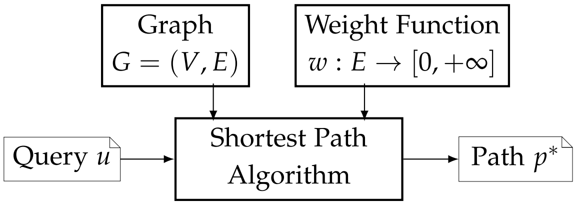


Figure 3.1: While solving a shortest path query, a shortest path algorithm incurs computation cost from three sources: examining the structure of the graph G, evaluating the edge weight function w, and maintaining internal data structures.

3.1 *The Shortest Path Problem*

Graphs provide a powerful abstraction capable of representing problems in a wide variety of domains from computer networking to puzzle solving to robotic motion planning. In particular, many important problems can be captured as *shortest path problems* (Figure 3.1), wherein a path p^* of minimal length is sought between two query vertices through a graph G with respect to an edge weight function w.

Despite the expansive applicability of this single abstraction, there exist a wide variety of algorithms in the literature for solving the shortest path problem efficiently. This is because the measure of computational efficiency, and therefore the correct choice of algorithm, is inextricably tied to the underlying problem domain.

The computational costs incurred by an algorithm can be broadly categorized into three sources corresponding to the blocks in Figure 3.1. One such source consists of queries on the structure of the graph G itself. The most commonly discussed such operation, *expanding* a vertex (determining its successors), is especially fundamental when the graph is represented implicitly, e.g. for domains with large graphs such as the 15-puzzle or Rubik's cube. It is with respect to vertex expansions that A* [47] is optimally efficient.

A second source of computational cost consists of maintaining ordered data structures inside the algorithm itself, which is especially important for problems with large branching factors. For such domains, approaches such as partial expansion [121] or iterative deepening [68] significantly reduce the number of vertices generated and stored by either selectively filtering surplus vertices from the frontier, or by not storing the frontier at all.

The third source of computational cost arises not from reasoning over the structure of G, but instead from evaluating the edge weight function w (i.e. we treat discovering an out-edge and determining its weight separately). Consider for example the problem of articulated robotic motion planning using roadmap methods [63]. While these graphs are often quite small (fewer than 10^5 vertices), determining the weight of each edge requires performing many collision and distance computations for the complex geometry of the robot and environment, resulting in planning times of multiple seconds to find a path.

As described in Chapter 2, we consider problem domains in which evaluating the edge weight function w dominates algorithm running time. In this chapter, we investigate the following research question:

> How can we minimize the number of edges we need to evaluate to answer shortest-path queries?

We make three primary contributions. First, inspired by lazy collision checking techniques from robotic motion planning [10], we formulate a class of shortest-path algorithms that is well-suited to problem domains with expensive edge evaluations. Second, we show that several existing algorithms in the literature can be expressed as special cases of this algorithm. Third, we show that the extensibility afforded by the algorithm allows for novel edge evaluation strategies, which can outperform existing algorithms over a set of example problems.

3.2 *Lazy Shortest Path Algorithm*

We describe a lazy approach to finding short paths which is well-suited to domains with expensive edge evaluations.

3.2.1 *Problem Definition*

A path p in a graph $G = (V, E)$ is composed of a sequence of adjacent edges connecting two endpoint vertices. Given an edge weight function $w : E \to [0, +\infty]$, the length of the path with respect to w is then:

$$\text{len}(p, w) = \sum_{e \in p} w(e). \tag{3.1}$$

Given a single-pair planning query $u : (v_{\text{start}}, v_{\text{goal}})$ inducing a set of satisfying paths P_u, the *shortest-path problem* is:

$$p^* = \arg\min_{p \in P_u} \text{len}(p, w). \tag{3.2}$$

A shortest-path algorithm computes a satisfying solution p^* given (G, u, w). Many such algorithms have been proposed to efficiently accommodate a wide array of underlying problem domains. The well-known principle of best-first search (BFS) is commonly employed to select vertices for expansion so as to minimize such expansions while guaranteeing optimality. Since we seek to minimize edge evaluations, we apply BFS to the question of selecting candidate paths in G for evaluation. The resulting algorithm, Lazy Shortest Path (LazySP), is presented in Algorithm 1, and can be applied to graphs defined implicitly or explicitly.

3.2.2 *The Algorithm*

We track evaluated edges with the set E_{eval}. We are given an estimator function w_{est} of the true edge weight w. This estimator is inexpensive to compute (e.g. edge length or even 0). We then define a *lazy* weight function w_{lazy} which returns the true weight of an evaluated edge and otherwise uses the inexpensive estimator w_{est}.

At each iteration of the search, the algorithm uses w_{lazy} to compute a candidate path $p_{\text{candidate}}$ by calling an existing solver SHORTEST-PATH (note that this invocation requires no evaluations of w). Once a candidate path has been found, it is returned if it is fully evaluated. Otherwise, an *edge selector* is employed which selects graph edge(s)

Algorithm 1 Lazy Shortest Path (LazySP)

```
function LazyShortestPath($G, u, w, w_{\text{est}}$)
    $E_{\text{eval}} \leftarrow \emptyset$
    $w_{\text{lazy}}(e) \leftarrow w_{\text{est}}(e) \quad \forall e \in E$
    loop
        $p_{\text{candidate}} \leftarrow$ ShortestPath($G, u, w_{\text{lazy}}$)
        if $p_{\text{candidate}} \subseteq E_{\text{eval}}$ then
            return $p_{\text{candidate}}$
        $E_{\text{selected}} \leftarrow$ Selector($G, p_{\text{candidate}}$)
        for $e \in E_{\text{selected}} \setminus E_{\text{eval}}$ do
            $w_{\text{lazy}}(e) \leftarrow w(e)$                ▷ Evaluate (expensive)
            $E_{\text{eval}} \leftarrow E_{\text{eval}} \cup e$
```

for evaluation. The true weights of these edges are then evaluated (incurring the requisite computational cost), and the algorithm repeats.

LazySP is complete and optimal:

Proof of all theorems are available in Appendix A.

Theorem 1 (Completeness of LazySP) *If the graph G is finite,* ShortestPath *is complete, and the set $E_{selected}$ returned by* Selector *returns at least one unevaluated edge on $p_{candidate}$, then* LazyShortestPath *is complete.*

Theorem 2 (Optimality of LazySP) *If w_{est} is chosen such that $w_{est}(e) \leq \epsilon\, w(e)$ for some parameter $\epsilon \geq 1$ and* LazyShortestPath *terminates with some path p_{ret}, then $len(p_{ret}, w) \leq \epsilon\, \ell^*$ with ℓ^* the length of an optimal path.*

The optimality of LazySP depends on the admissibility of w_{est} in the same way that the optimality of A* depends on the admissibility of its goal heuristic h. Theorem 2 establishes the general bounded suboptimality of LazySP w.r.t. the inflation parameter ϵ. While our theoretical results (e.g. equivalences) hold for any choice of ϵ, for clarity our examples and experimental results focus on cases with $\epsilon = 1$.

3.2.3 *The Edge Selector: Key to Efficiency*

The LazySP algorithm exhibits a rough similarity to optimal replanning algorithms such as D* [115, 116] which plan a sequence of shortest paths for a mobile robot as new edge weights are discovered during its traverse. D* treats edge changes passively as an aspect of the problem setting (e.g. a sensor with limited range).

The key difference is that our problem setting treats edge evaluations as an active choice that can be exploited. While any choice

Algorithm 2 Various Simple LazySP Edge Selectors

```
function SelectExpand($G, p_{\text{candidate}}$)
    $e_{\text{first}} \leftarrow$ first unevaluated $e \in p_{\text{candidate}}$
    $v_{\text{frontier}} \leftarrow G$.source($e_{\text{first}}$)
    $E_{\text{selected}} \leftarrow G$.out_edges($v_{\text{frontier}}$)
    return $E_{\text{selected}}$
function SelectForward($G, p_{\text{candidate}}$)
    return {first unevaluated $e \in p_{\text{candidate}}$}
function SelectReverse($G, p_{\text{candidate}}$)
    return {last unevaluated $e \in p_{\text{candidate}}$}
function SelectAlternate($G, p_{\text{candidate}}$)
    if LazySP iteration number is odd then
        return {first unevaluated $e \in p_{\text{candidate}}$}
    else
        return {last unevaluated $e \in p_{\text{candidate}}$}
function SelectBisection($G, p_{\text{candidate}}$)
    return {unevaluated $e \in p_{\text{candidate}}$ furthest from nearest evaluated edge}
```

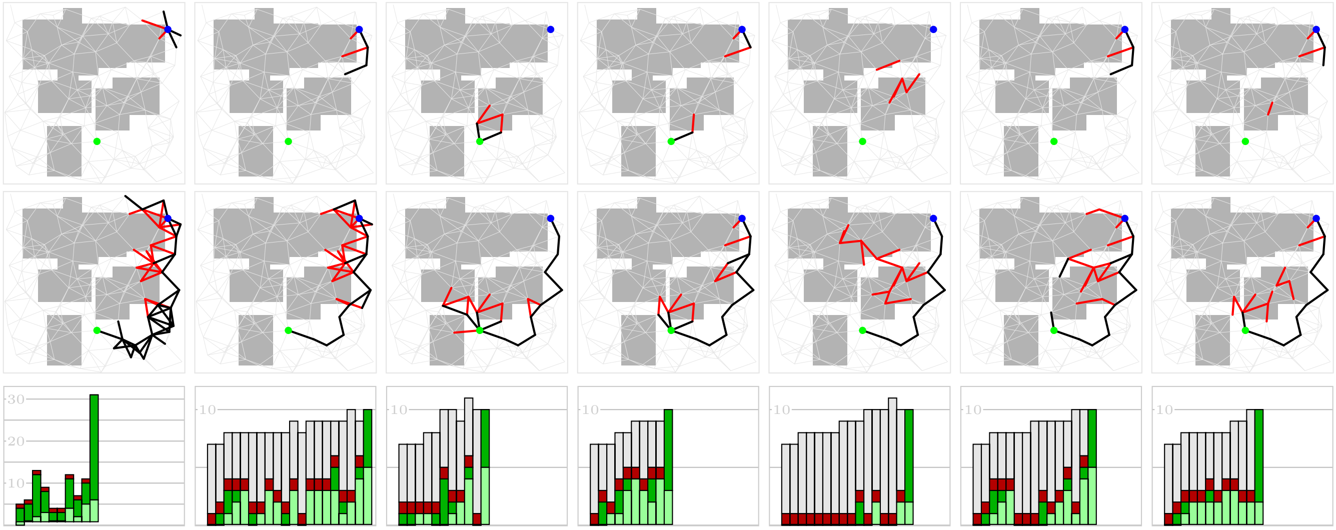
of edge selector that meets the conditions above will lead to an algorithm that is complete and optimal, its *efficiency* is dictated by the choice of this selector. This motivates the theoretical and empirical investigation of different edge selectors in this chapter.

Simple selectors. We codify five common strategies in Algorithm 2. The Expand selector captures the edge weights that are evaluated during a conventional vertex expansion. The selector identifies the first unevaluated edge e_{first} on the candidate path, and considers the source vertex of this edge a *frontier* vertex. It then selects all out-edges of this frontier vertex for evaluation. The Forward and Reverse selectors select the first and last unevaluated edge on the candidate path, respectively (note that Forward returns a subset of Expand).

The Alternate selector simply alternates between Forward and Reverse on each iteration. This can be motivated by both bidirectional search algorithms as well as motion planning algorithms such as RRT-Connect [69] which tend to perform well w.r.t. state evaluations.

The Bisection selector chooses among those unevaluated edges the one furthest from an evaluated edge on the candidate path. This selector is roughly analogous to the collision checking strategy employed by the Lazy PRM [10] as applied to our problem on abstract graphs.

In the following section, we demonstrate that instances of LazySP using simple selectors yield equivalent results to existing vertex algo-



(a) Expand(77) (b) Forward(34) (c) Reverse(24) (d) Alternate(23) (e) Bisect(25) (f) WeightSamp(22) (g) Partition(22)

Figure 3.2: Snapshots of the LazySP algorithm using each edge selector discussed in this chapter on the same obstacle roadmap graph problem, with start (●) and goal (●). At top, the algorithms after evaluating five edges (evaluated edges labeled as / valid or / invalid). At middle, the final set of evaluated edges. At bottom, for each unique path considered from left to right, the number of edges on the path that are 🟩 already evaluated, 🟩 evaluated and valid, 🟥 evaluated and invalid, and ☐ unevaluated. The total number of edges evaluated is noted in brackets. Note that the scale on the Expand plot has been adjusted because the selector evaluates many edges not on the candidate path at each iteration.

rithms. We then discuss two more sophisticated selectors motivated by weight function sampling and statistical mechanics.

3.3 *Edge Equivalence to A* Variants*

In the previous section, we introduced LazySP as the path-selection analogue to BFS vertex-selection algorithms. In this section, we make this analogy more precise. In particular, we show that LazySP-Expand is edge-equivalent to a variant of A* (and Weighted A*), and that LazySP-Forward is edge-equivalent to a variant of Lazy Weighted A* (see Table 3.1). It is important to be specific about the conditions under which these equivalences arise, which we detail here.

Table 3.1: LazySP equivalence results. The A*, LWA*, and BHFFA algorithms use reopening and the dynamic h_{lazy} heuristic (3.4).

LazySP Selector	Existing Algorithm	Result
Expand	(Weighted) A*	Edge-equivalent (Theorems 3, 4)
Forward	Lazy Weighted A*	Edge-equivalent (Theorems 5, 6)
Alternate	Bidirectional Heuristic Front-to-Front Algorithm	Conjectured

Edge equivalence. We say that two algorithms are *edge-equivalent* if they evaluate the same edges in the same order. We consider an algorithm to have evaluated an edge the first time the edge's true weight is requested.

Arbitrary tiebreaking. For some graphs, an algorithm may have multiple allowable choices at each iteration (e.g. LazySP with multiple shortest candidate paths, or A* with multiple vertices in OPEN with lowest f-value). We will say that algorithm A is equivalent to algorithm B if for any choice available to A, there exists an allowable choice available to B such that the same edge(s) are evaluated by each.

A* with reopening. We show equivalence to variants of A* and Lazy Weighted A* that do not use a CLOSED list to prevent vertices from being visited more than once.

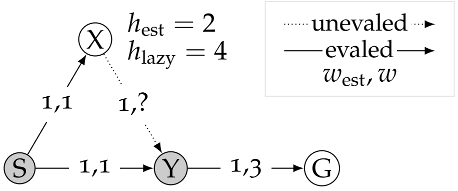


Figure 3.3: A* comparison between the static goal heuristic h_{est} (3.3) and the dynamic goal heuristic h_{lazy} (3.4) on a simple graph from start S to goal G. The values of both the edge weight estimate w_{est} and the true edge weight w (for evaluated edges) are shown. Using either goal heuristic, the A* algorithm first expands vertices S and Y, evaluating three edges in total and leaving X and G on OPEN. After finding that edge YG has $w = 3$, the dynamic heuristic value $h_{\text{lazy}}(\text{X})$ is updated from 2 to 4. While the A* using the static h_{est} would next expand X, the A* using the dynamic h_{lazy} would next expand G and terminate, having never evaluated edge XY.

A* with a dynamic heuristic. In order to apply A* and Lazy Weighted A* to our problem, we need a goal heuristic over vertices. The most simple may be

$$h_{\text{est}}(v) = \min_{p : v \rightarrow v_g} \text{len}(p, w_{\text{est}}). \tag{3.3}$$

Note that the value of this heuristic could be computed as a preprocessing step using Dijkstra's algorithm [29] before iterations begin. However, in order for the equivalences to hold, we require the use of the lazy heuristic

$$h_{\text{lazy}}(v) = \min_{p : v \rightarrow v_g} \text{len}(p, w_{\text{lazy}}). \tag{3.4}$$

This heuristic is dynamic in that it depends on w_{lazy} which changes as edges are evaluated. Therefore, heuristic values must be recomputed for all affected vertices on OPEN after each iteration.

3.3.1 Equivalence to A*

We show that the LazySP-Expand algorithm is edge-equivalent to a variant of the A* shortest-path algorithm. We make use of two invariants that are maintained during the progression of A*.

Proof of all invariants are available in Appendix A.

Invariant 1 *If v is discovered by A* and v' is undiscovered, with v' a successor of v, then v is on OPEN.*

Invariant 2 *If v and v' are discovered by A*, with v' a successor of v, and $g[v] + w(v, v') < g[v']$, then v is on OPEN.*

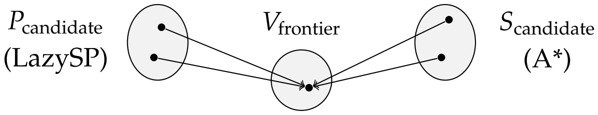


Figure 3.4: Illustration of the equivalence between A* and LazySP-Expand. After evaluating the same set of edges, the next edges to be evaluated by each algorithm can both be expressed as a surjective mapping onto a common set of unexpanded frontier vertices.

When we say a vertex is *discovered*, we mean that it is either on OPEN or CLOSED. Note that Invariant 2 holds because we allow vertices to be reopened; without reopening (and with an inconsistent heuristic), later finding a cheaper path to v (and not reopening v') would invalidate the invariant.

We will use the goal heuristic h_{lazy} from (3.4). Note that if an admissible edge weight estimator $\hat{w}$ exists (that is, $\hat{w} \leq w$), then our A* can approximate the Weighted A* algorithm [95] with parameter ϵ by using $w_{\text{est}} = \epsilon\, \hat{w}$, and the suboptimality bound from Theorem 2 holds.

Equivalence. In order to show edge-equivalence, we consider the case where both algorithms are beginning a new iteration having so far evaluated the same set of edges.

LazySP-Expand has some set $P_{\text{candidate}}$ of allowable candidate paths minimizing $\text{len}(p, w_{\text{lazy}})$; the Expand selector will then identify a vertex on the chosen path for expansion.

A* will iteratively select a set of vertices from OPEN to expand. Because it is possible that a vertex is expanded multiple times (and only the first expansion results in edge evaluations), we group iterations of A* into *sequences*, where each sequence s consists of (a) zero or more vertices from OPEN that have already been expanded, followed by (b) one vertex from OPEN that is to be expanded for the first time.

We show that both the set of allowable candidate paths $P_{\text{candidate}}$ available to LazySP-Expand and the set of allowable candidate vertex sequences $S_{\text{candidate}}$ available to A* map surjectively to the same set of unexpanded frontier vertices V_{frontier} as illustrated in Figure 3.4. This is described by way of Theorems 3 and 4 below.

Proof of all theorems are available in Appendix A.

Theorem 3 *If LazySP-Expand and A* have evaluated the same set of edges, then for any candidate path $p_{\text{candidate}}$ chosen by LazySP yielding frontier vertex v_{frontier}, there exists an allowable A* sequence $s_{\text{candidate}}$ which also yields v_{frontier}.*

Theorem 4 *If LazySP-Expand and A* have evaluated the same set of edges, then for any candidate sequence $s_{\text{candidate}}$ chosen by A* yielding frontier vertex v_{frontier}, there exists an allowable LazySP path $p_{\text{candidate}}$ which also yields v_{frontier}.*

Algorithm 3 Lazy Weighted A* (without CLOSED list)

```
1:  function LazyWeightedA*($G, w, \hat{w}, h$)
2:      $g[v_{\text{start}}] \leftarrow 0$
3:      $Q_v \leftarrow \{v_{\text{start}}\}$                          ▷ Key: $g[v] + h(v)$
4:      $Q_e \leftarrow \emptyset$                 ▷ Key: $g[v] + \hat{w}(v, v') + h(v')$
5:      while $\min(Q_v.\text{TopKey}, Q_e.\text{TopKey}) < g[v_{\text{goal}}]$ do
6:          if $Q_v.\text{TopKey} \leq Q_e.\text{TopKey}$ then
7:              $v \leftarrow Q_v.\text{Pop}()$
8:              for $v' \in G.\text{GetSuccessors}(v)$ do
9:                  $Q_e.\text{Insert}((v, v'))$
10:         else
11:             $(v, v') \leftarrow Q_e.\text{Pop}()$
12:             if $g[v'] \leq g[v] + \hat{w}(v, v')$ then
13:                 continue
14:             $g_{\text{new}} \leftarrow g[v] + w(v, v')$                ▷ evaluate
15:             if $g_{\text{new}} < g[v']$ then
16:                 $g[v'] = g_{\text{new}}$
17:                 $Q_v.\text{Insert}(v')$
```

3.3.2 *Equivalence to Lazy Weighted A**

In a conventional vertex expansion algorithm, determining a successor's cost is a function of both the cost of the edge and the value of the heuristic. If either of these components is expensive to evaluate, an algorithm can defer its computation by maintaining the successor on the frontier with an approximate cost until it is expanded. The Fast Downward algorithm [50] is motivated by expensive heuristic evaluations in planning, whereas the Lazy Weighted A* (LWA*) algorithm [20] is motivated by expensive edge evaluations in robotics.

We show that the LazySP-Forward algorithm is edge-equivalent to a variant of the Lazy Weighted A* shortest-path algorithm. For a given candidate path, the Forward selector returns the first unevaluated edge.

Variant of Lazy Weighted A*. We reproduce a variant of LWA* without a CLOSED list in Algorithm 3. For the purposes of our analysis, the reproduction differs from the original presentation, and we detail those differences here. With the exception of the lack of CLOSED, the differences do not affect the behavior of the algorithm.

The most obvious difference is that we present the original OPEN list as separate vertex (Q_v) and edge (Q_e) priority queues, with sorting keys shown on lines 3 and 4. A vertex v in the original OPEN with $trueCost(v) = true$ corresponds to a vertex v in Q_v, whereas a vertex v' in the original OPEN with $trueCost(v') = false$ (and parent

v) corresponds to an edge (v, v') in Q_e. Use of the edge queue obviates the need for duplicate vertices on OPEN with different parents and the $conf(v)$ test for identifying such duplicates. This presentation also highlights the similarity between LWA* and the inner loop of the Batch Informed Trees (BIT*) algorithm [41].

The second difference is that the edge usefulness test (line 12 of the original algorithm) has been moved from before inserting into OPEN to after being popped from OPEN, but before being evaluated (line 12 of Algorithm 3). This change is partially in compensation for removing the CLOSED list. This adjustment does not affect the edges evaluated.

We make use of an invariant that is maintained during the progression of Lazy Weighted A*.

Proof of all invariants are available in Appendix A.

Invariant 3 *For all vertex pairs v and v', with v' a successor of v, if $g[v] + \max(w(v, v'), \hat{w}(v, v')) < g[v']$, then either vertex v is on Q_v or edge (v, v') is on Q_e.*

We will use $h(v) = h_{\text{lazy}}(v)$ from (3.4) and $\hat{w} = w_{\text{lazy}}$. Note that the use of these dynamic heuristics requires that the Q_v and Q_e be resorted after every edge is evaluated.

Equivalence. The equivalence follows similarly to that for A* above. Given the same set of edges evaluated, the set of allowable next evaluations is identical for each algorithm.

Proof of all theorems are available in Appendix A.

Theorem 5 *If LazySP-Forward and LWA* have evaluated the same set of edges, then for any allowable candidate path $p_{candidate}$ chosen by LazySP yielding first unevaluated edge e_{ab}, there exists an allowable LWA* sequence $s_{candidate}$ which also yields e_{ab}.*

Theorem 6 *If LazySP-Forward and LWA* have evaluated the same set of edges, then for any allowable sequence of vertices and edges $s_{candidate}$ considered by LWA* yielding evaluated edge e_{ab}, there exists an allowable LazySP candidate path $p_{candidate}$ which also yields e_{ab}.*

3.3.3 *Relation to Bidirectional Heuristic Search*

LazySP-Alternate chooses unevaluated edges from either the beginning or the end of the candidate path at each iteration. We conjecture that an alternating version of the Expand selector is edge-equivalent to the Bidirectional Heuristic Front-to-Front Algorithm [23] for appropriate lazy vertex pair heuristic, and that LazySP-Alternate is edge-equivalent to a bidirectional LWA*.

Algorithm 4 Maximum Edge Probability Selector *(for WeightSamp and Partition path distributions)*

```
function SelectMaxEdgeProb($G$, $p_{\text{candidate}}$, $\mathcal{D}_p$)
    $p(e) \leftarrow \Pr(\, e \in P \,)$ for $P \sim \mathcal{D}_p$
    $e_{\max} \leftarrow$ unevaluated $e \in p_{\text{candidate}}$ maximizing $p(e)$
    return $\{e_{\max}\}$
```

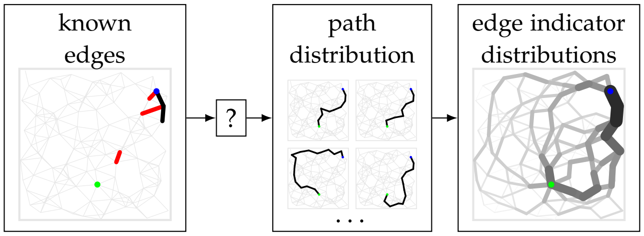


Figure 3.5: Illustration of maximum edge probability selectors. A distribution over paths (usually conditioned on the known edge evaluations) induces on each edge e a Bernoulli distribution with parameter $p(e)$ giving the probability that it belongs to the path. The selector chooses the edge with the largest such probability.

3.4 *Novel Edge Selectors*

Because we are conducting a search over paths, we are free to implement selectors which are not constrained to evaluate edges in any particular order (i.e. to maintain evaluated trees rooted at the start and goal vertices). In this section, we describe a novel class of edge selectors which is designed to reduce the total number of edges evaluated during the course of the LazySP algorithm. These selectors operate by maintaining a distribution over potential paths at each iteration of the algorithm (see Figure 3.5). This path distribution induces a Bernoulli distribution for each edge e which indicates its probability $p(e)$ to lie on the potential path; at each iteration, the selectors then choose the unevaluated edge that maximizes this edge indicator probability (Algorithm 4). The two selectors described in this section differ with respect to how they maintain this distribution over potential paths.

3.4.1 *Weight Function Sampling Selector*

The first selector, WeightSamp, is motivated by the intuition that it is preferable to evaluate edges that are most likely to lie on the true shortest path. Therefore, it computes its path distribution $\mathcal{D}_p$ by performing shortest path queries on sampled edge weight functions drawn from a distribution $\mathcal{D}_w$. This edge weight distribution is conditioned on the the known weights of all previously evaluated edges E_{eval}:

$$\mathcal{D}_p : \mathrm{SP}(w) \text{ for } w \sim \mathcal{D}_w(E_{\text{eval}}). \tag{3.5}$$

For example, the distribution $\mathcal{D}_w$ might consist of the edge weights induced by a model of the distribution of environment obstacles (Fig-

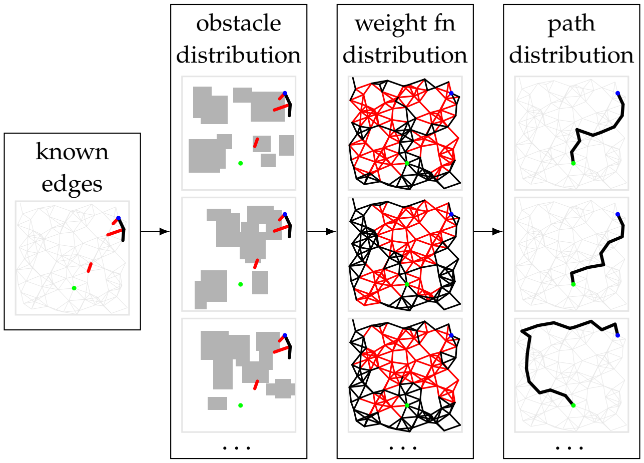


Figure 3.6: The WeightSamp selector uses the path distribution induced by solving the shortest path problem on a distribution over possible edge weight functions $\mathcal{D}_w$. In this example, samples from $\mathcal{D}_w$ are computed by drawing samples from $\mathcal{D}_O$, the distribution of obstacles that are consistent with the known edge evaluations.

ure 3.6). Since this obstacle distribution is conditioned on the results of known edge evaluations, we consider the subset of worlds which are consistent with the edges we have evaluated so far. However, depending on the fidelity of this model, solving the corresponding shortest path problem for a given sampled obstacle arrangement might require as much computation as solving the original problem, since it requires computing the resulting edge weights. In practice, we can approximate $\mathcal{D}_w$ by assuming that each edge is independently distributed.

3.4.2 *Partition Function Selector*

While the WeightSamp selector captures the intuition that it is preferable to focus edge evaluations in areas that are useful for many potential paths, the computational cost required to calculate it at each iteration may render it intractable. One candidate path distribution that is more efficient to compute follows an exponential form:

$$\mathcal{D}_p : f_P(p) \propto \exp(-\beta \operatorname{len}(p, w_{\text{lazy}})). \tag{3.6}$$

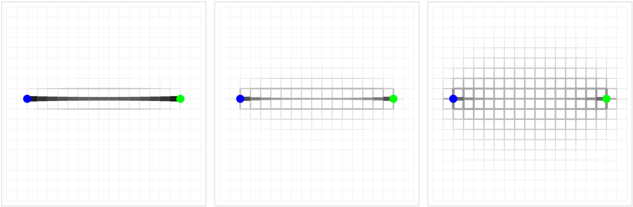
In other words, we consider all potential paths P between the start and goal vertices, with shorter paths assigned more exponentially probability than longer ones (with positive parameter β). We call this the Partition selector because this distribution is closely related to calculating partition functions from statistical mechanics. The corresponding partition function is:

$$Z(P) = \sum_{p \in P} \exp(-\beta \operatorname{len}(p, w_{\text{lazy}})). \tag{3.7}$$

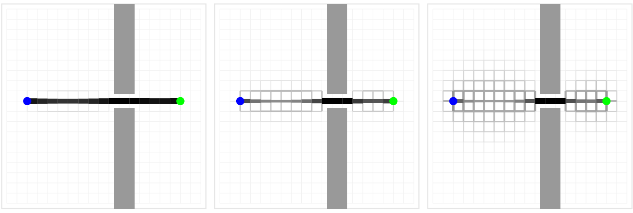
Note that the edge indicator probability required in Algorithm 4 can then be written:

$$p(e) = 1 - \frac{Z(P \setminus e)}{Z(P)}. \tag{3.8}$$

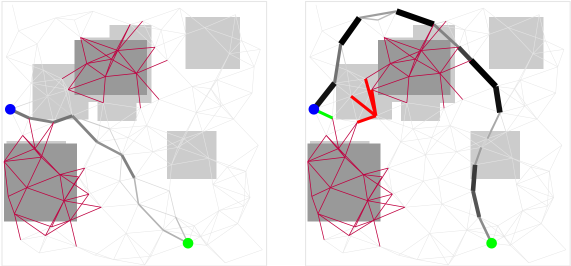
Here, $P \setminus e$ denotes paths in P that do not contain edge e.



(a) Initial $p(e)$ scores on a constant-weight grid with β: 50, 33, 28



(b) Initial $p(e)$ scores with ∞-weight obstacles with β: 50, 33, 28



(c) Initial $p(e)$ scores (d) Scores after five evaluations

Figure 3.7: Examples of the Partition selector's $p(e)$ edge score function. With no known obstacles, a high β assigns near-unity score to only edges on the shortest path; as β decreases and more paths are considered, edges immediately adjacent to the roots score highest. Since all paths must pass through the narrow passage, edges within score highly. For a problem with two a-priori known obstacles (dark gray), the score first prioritizes evaluations between the two. Upon finding these edges are blocked, the next edges that are prioritized lie along the top of the world.

It may appear advantageous to restrict P to only *simple* paths, since all optimal paths are simple. Unfortunately, an algorithm for computing (3.8) efficiently is not currently known in this case. However, in the case that P consists of all paths, there does exist an efficient incremental calculation of (3.7) via a recursive formulation.

We use the notation $Z_{xy} = Z(P_{xy})$, with P_{xy} the set of paths from x to y. Suppose the values Z_{xy} are known between all pairs of vertices x, y for a graph G. (For a graph with no edges, Z_{xy} is 1 if $x = y$ and 0 otherwise.) Consider a modified graph G' with one additional edge e_{ab} with weight w_{ab}. All additional paths use the new edge e_{ab} a non-zero number of times; the value Z'_{xy} can be shown to be

$$Z'_{xy} = Z_{xy} + \frac{Z_{xa} Z_{by}}{\exp(\beta w_{ab}) - Z_{ba}} \text{ if } \exp(\beta w_{ab}) > Z_{ba}. \tag{3.9}$$

This form is derived from simplifying the induced geometric series; note that if $\exp(\beta w_{ab}) \leq Z_{ba}$, the value Z'_{xy} is infinite. One can also derive the inverse: given values Z', calculate the values Z if an edge were removed. A derivation of this formulation is given in Appendix B.

This incremental formulation of (3.7) allows for the corresponding score $p(e)$ for edges to be updated efficiently during each iteration of LazySP as the w_{lazy} value for edges chosen for evaluation are updated. In fact, if the values Z are stored in a square matrix, the update for all pairs after an edge weight change consists of a single vector outer product.

3.5 *Experiments*

We compared the seven edge selectors on three classes of shortest path problems. The average number of edges evaluated by each, as well as timing results from our implementations, are shown in Figure 3.9. In each case, the estimate was chosen so that $w_{\text{est}} \leq w$, so that all runs produced optimal paths. The experimental results serve primarily to illustrate that the A* and LWA* algorithms (i.e. Expand and Forward) are not optimally edge-efficient, but they also expose differences in behavior and prompt future research directions. All experiments were conducted using an open-source implementation. Motion planning results were implemented using OMPL [119].

Random partially-connected graphs. We tested on a set of 1000 randomly-generated undirected graphs with $|V| = 100$, with each pair of vertices sharing an edge with probability 0.05. Edges have an independent 0.5 probability of having infinite weight, else the weight is uniformly distributed on $[1, 2]$; the estimated weight was unity for all edges. For the WeightSamp selector, we drew 1000 w samples. For the Partition selector, we used $\beta = 2$.

Roadmap graphs on the unit square. We considered roadmap graphs formed via the first 100 points of the $(2, 3)$-Halton sequence on the unit square with a connection radius of 0.15, with 30 pairs of start and goal vertices chosen randomly. The edge weight function was derived from 30 sampled obstacle fields consisting of 10 randomly placed boxes with dimensions uniform on $[0.1, 0.3]$, with each edge having infinite weight on collision, and weight equal to its Euclidean length otherwise. One of the resulting 900 example problems is shown in Figure 3.2. For the WeightSamp selector, we drew 1000 w samples with a naïve edge weight distribution in which each edge had an independent 0.1 collision probability. For the Partition selector, we used $\beta = 21$.

Roadmap graphs for robot arm motion planning. We considered roadmap graphs in the configuration space corresponding to 7-DOF right arm of the HERB home robot across three motion planning problems inspired by a table clearing scenario (see Figure 3.8). The problems consisted of first moving from the robot's home configuration to one of 7 feasible grasp configurations for a mug (pictured), second transferring the mug to one of 72 feasible configurations with the mug above the blue bin, and third returning to the home configuration. Each problem was solved independently. This common scenario spans various numbers of starts/goals and allows a comparison w.r.t. difficulty at different problem stages as discussed later.

For each problem, 50 random graphs were constructed by applying a random offset to the 7D Halton sequence with $N = 1000$, with additional vertices for each problem start and goal configuration. We used an edge connection radius of 3 rad, resulting $|E|$ ranging from 23404 to 28109. Each edge took infinite weight on collision, and weight equal to its Euclidean length otherwise. For the WeightSamp selector, we drew 1000 w samples with a naïve edge weight distribution in which each edge had an independent 0.1 probability of collision. For the Partition selector, we used $\beta = 3$.

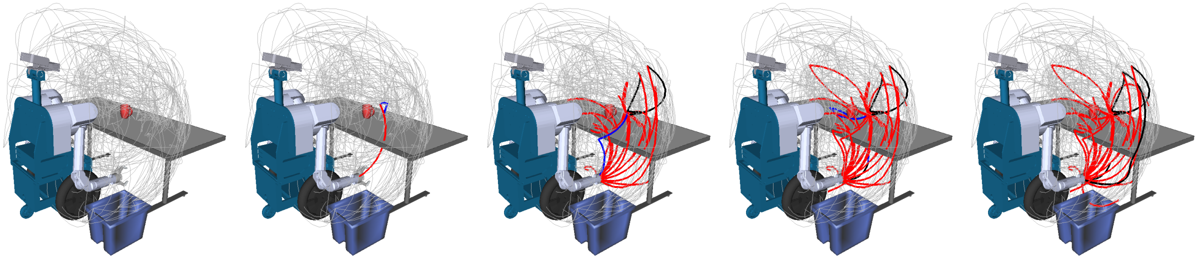


Figure 3.8: Visualization of the first of three articulated motion planning problems in which the HERB robot must move its right arm from the start configuration (pictured) to any of seven grasp configurations for a mug. Shown is the progression of the Alternate selector on one of the randomly generated roadmaps; approximately 2% of the 7D roadmap is shown in gray by projecting onto the space of end-effector positions.

3.6 *Discussion*

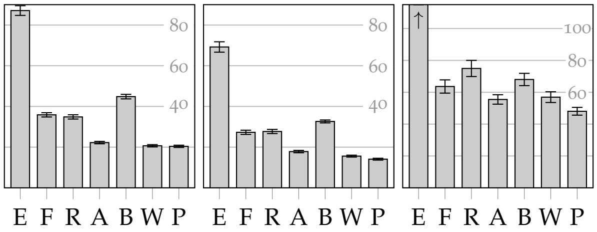
The first observation that is evident from the experimental results is that lazy evaluation – whether using Forward (LWA*) or one of the other selectors – grossly outperforms Expand (A*). The relative penalty that Expand incurs by evaluating all edges from each expanded vertex is a function of the graph's branching factor.

Since the Forward and Reverse selectors are simply mirrors of each other, they exhibit similar performance averaged across the PartConn and UnitSquare problem classes, which are symmetric. However, this need not the case for a particular instance. For example, the start of ArmPlan1 and the goal of ArmPlan3 consist of the arm's single home configuration in a relatively confined space. As shown in the table in Figure 3.9(a), it appears that the better selector on these problems attempts to solve the more constrained side of the problem first. While it may be difficult to determine a priori which part of the problem will be the most constrained, the simple Alternate selector's respectable performance suggests that it may be a reasonable compromise.

The per-path plots at the bottom of Figure 3.2 allow us to charac-

	E	F	R	A	B	W	P‡
PartConn	87.10	35.86	34.84	22.23	44.81	**20.66**	**20.39**
online†(ms)	***1.22***	*1.96*	*1.86*	***1.20***	*2.41*	*4807.19*	*3.32*
sel (ms)	*0.02*	*0.01*	*0.01*	*0.01*	*0.03*	*4805.64*	*2.07*
UnitSquare	69.21	27.29	27.69	17.82	32.62	15.58	**14.08**
online†(ms)	***0.91***	*1.47*	*1.49*	***0.94***	*1.71*	*3864.95*	*1.72*
sel (ms)	*0.01*	*0.01*	*0.01*	*0.01*	*0.02*	*3863.49*	*0.87*
ArmPlan(avg)	949.05	63.62	74.94	55.48	68.01	56.93	**48.07**
online (s)	*269.82*	***5.90***	*8.22*	***5.96***	*7.34*	*3402.21*	***5.80***
sel (s)	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*3392.76*	*1.54*
eval (s)	*269.78*	*5.87*	*8.20*	*5.94*	*7.31*	*9.39*	*4.21*
ArmPlan1	344.74	**49.72**	95.58	59.44	58.90	73.72	**50.66**
online (s)	*109.09*	***4.81***	*14.81*	*7.03*	*7.91*	*3375.35*	*7.25*
sel (s)	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*3358.82*	*1.61*
eval (s)	*109.07*	*4.78*	*14.77*	*7.01*	*7.88*	*16.47*	*5.59*
ArmPlan2	657.02	**62.24**	98.54	69.96	75.88	**66.24**	**62.16**
online (s)	*166.19*	***3.27***	*7.36*	*5.95*	*5.63*	*4758.04*	*5.99*
sel (s)	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*4750.16*	*2.03*
eval (s)	*166.17*	*3.26*	*7.34*	*5.93*	*5.61*	*7.82*	*3.91*
ArmPlan3	1845.38	78.90	**30.70**	37.04	69.26	**30.82**	**31.38**
online (s)	*534.16*	*9.61*	***2.50***	*4.91*	*8.47*	*2073.23*	*4.17*
sel (s)	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*2069.29*	*0.98*
eval (s)	*534.10*	*9.58*	*2.48*	*4.89*	*8.44*	*3.90*	*3.15*

(a) Average number of edges evaluated for each problem class and selector. The minimum selector, along with any selector within one unit of its standard error, is shown in bold. The ArmPlan class is split into its three constituent problems. Online timing results are also shown, including the components from the invoking the selector and evaluating edges. †PartConn and UnitSquare involve trivial edge evaluation time. ‡Timing for the Partition selector does not include pre-computation time. See Figure A.1 for details.



(b) PartConn (c) UnitSquare (d) ArmPlan

Figure 3.9: Experimental results for the three problem classes across each of the seven selectors, E:Expand, F:Forward, R:Reverse, A:Alternate, B:Bisection, W:WeightSamp, and P:Partition. In addition to the summary table (a), the plots (b-d) show summary statistics for each problem class. The means and standard errors in (b-c) are across the 1000 and 900 problem instances, respectively. The means and standard errors in (d) are for the average across the three constituent problems for each of the 50 sampled roadmaps. A more detailed table of results is available in Appendix A.

terize the selectors' behavior. For example, Alternate often evaluates several edges on each path before finding an obstacle. Its early evaluations also tend to be useful later, and it terminates after considering 10 paths on the illustrated problem. In contrast, Bisection exhibits a fail-fast strategy, quickly invalidating most paths after a single evaluation, but needing 16 such paths (with very little reuse) before it terminates. In general, the Bisection selector did not outperform any of the lazy selectors in terms of number of edges evaluated. However, it may be well suited to problem domains in which evaluations that fail tend be less costly.

The novel selectors based on path distributions tend to minimize edge evaluations on the problems we considered. While the WeightSamp selector performs similarly to Partition on the simpler problems, it performs less well in the ArmPlan domain. This may be because many more samples are needed to approximate the requisite path distribution.

The path distribution selectors are motivated by focusing evaluation effort in areas that are useful for many distinct candidate paths, as illustrated in Figure 3.7. Note that in the absence of a priori knowledge, the edges nearest to the start and goal tend to have the highest $p(e)$ score, since they are members of many potential paths. Because it tends to focus evaluations in a similar way, the Alternate selector may serve as a simple proxy for the more complex selectors.

We note that an optimal edge selector could be theoretically achieved by posing the edge selection problem as a POMDP, given a probabilistic model of the true costs. While likely intractable in complex domains, exploring this solution may yield useful approximations or insights.

4
Incremental Bidirectional Search

In Chapter 3, we introduced the Lazy Shortest Path (LazySP) algorithm, which addresses domains with expensive edge weight functions by interleaving the evaluation phase with a sequence of search queries using an existing pathfinding algorithm. Because this inner search is conducted many times, its efficiency is paramount.

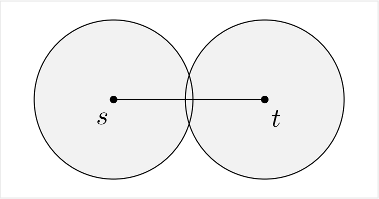
Most approaches for reducing the computational cost of pathfinding attempt to focus the search on a smaller subset of the graph. We consider three classes of such techniques (Figure 4.1):

1. *Bidirectional Search* – A bidirectional algorithm conducts two concurrent searches, one from the start vertex *s*, and the other from the destination vertex *t*. The depth to which each search must descend is typically a factor of two shallower than that of a unidirectional search. Such searches are therefore well-suited to graphs with larger branching factors. For example, for a roadmap graph, the number of vertices is polynomial in the depth and exponential in the dimensionality of the space.

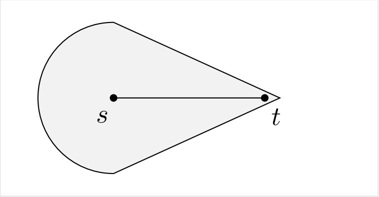
2. *Heuristic Search* – A heuristic-informed algorithm exploits a destination-directed heuristic function over vertices to bias exploration in the direction of the destination vertex. A strong and admissible such heuristic can drastically speed the search by reducing the number of vertices that must be expanded.

3. *Incremental Search* – An incremental algorithm is applicable to the dynamic pathfinding problem in which a sequence of search episodes are conducted with edge weight function changes (partially) between episodes. An algorithm is termed *incremental* because it incrementally updates only the portion of its data structures that are affected by the updated edge weight changes.

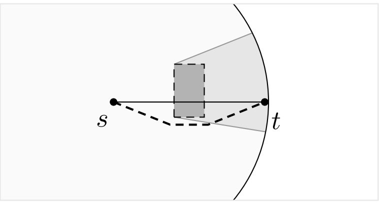
The principal contribution of this chapter is IBiD, an algorithm which combines these three techniques into a single algorithm (Ta-



(a) Bidirectional search. A path is found near the intersection of the two searches.



(b) Heuristic search. The search is biased towards the destination vertex.



(c) Incremental search. After a straight-line path is found for the first planning episode, a new obstacle appears; only the shaded region must be recomputed to find the new shortest path.

Figure 4.1: Illustrations of the three focusing techniques considered on a spatial pathfinding problem.

Note that vertex/edge insertions and deletions can be modeled as edge weight changes via infinite weights.

ble 4.1). While originally motivated for use with LazySP, IBiD is broadly applicable to incremental search problems.

Chapter outline. Finding shortest paths on graphs is a very extensively studied problem. This chapter begins with a comprehensive review of methods which solve shortest path problems by computing distance functions. After defining the problem in Section 4.1, we review distance functions and unidirectional methods in Section 4.2. Section 4.2.3 reviews bidirectional search methods, and Section 4.2.4 reviews incremental search. Section 4.3, introduces IBiD, an algorithm which combines bidirectional and incremental search. In Section 4.4, we review heuristic search methods, and discuss a heuristic-informed generalization of IBiD. The chapter concludes with experimental results and implementation notes.

	Unidirectional	Bidirectional
Complete *(Heuristic)*	Dijkstra [29] *A* [47]*	Bidirectional Dijkstra [85] *Bidirectional A* [54]*
Incremental *(Heuristic)*	DynamicSWSF-FP [98] *Lifelong Planning A* [67]*	IBiD *Heuristic IBiD*

Table 4.1: IBiD generalizes both the heuristic-informed bidirectional Dijkstra's search [45] and DynamicSWSF-FP [98]. There are a great many algorithms that we could place in each cell; we provide only a representative choice in each.

4.1 Problem Definition

The *shortest path problem* on graphs has been extensively studied over the past six decades. Consider a directed graph $G = (V, E)$ and accompanying edge weight function $w : E \rightarrow \mathbb{R}$. Note that we allow graphs with multiple edges connecting any pair of vertices, as well as graphs with edges to and from the same vertex.

The length of a path is equal to the sum of the weights of its constituent edges. We consider the *single-pair shortest path* (SPSP) problem, in which a path of minimal length is sought between distinct start and destination vertices $s, t \in V$. (We can also handle planning problems with multiple start/destination configurations as an SPSP problem by introducing virtual vertices and edges connecting to each candidate start/goal.)

The single-pair problem is also called the *two-terminal* or *point-to-point* problem.

Our review is applicable to problems where w is everywhere finite. The algorithms that we consider do not distinguish between nonexistent and infinite-weight edges, and so will return that no path exists in the case where shortest paths contain infinite-weight edges.

An example problem. We will carry forward an illustrative example problem from the public dataset of the 9th DIMACS Implementation

Challenge [27] (Northeast USA) comprising an approximate road network, using transit time as the edge weight function (Figure 4.2). In this way, a shortest path between a pair of start and destination locations minimizes the total transit time between them. More details about the graph are given in the experimental section in Section 4.5. The graph is sufficiently large that minimizing search computation is important, and the available transit time heuristic is broadly useful but not perfectly strong. The road routing domain is of great interest, and it is commonly used for benchmarking and easy to visualize.

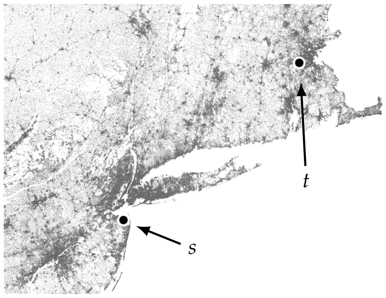


Figure 4.2: A graph of the Northeast USA from the 9th DIMACS Implementation Challenge comprises 1,524,453 vertices and 3,868,020 directed edges. A shortest path problem from a start s in New Jersey to a destination t outside Boston will be used as an example.

Problem Settings. The single-pair problem has been extensively studied. There are techniques that are particular to memory-constrained settings [58] or to settings where pre-computation is available [44]. While we do not focus on such settings, the algorithms we propose may be complementary to these techniques.

4.2 *Review of Pathfinding with Distance Functions*

This section contains a unified presentation of unidirectional, bidirectional, and incremental search strategies by examining the properties of the distance functions that they maintain. These properties and invariants can be established for arbitrary distance function approximations. Examination of these properties then informs the development of algorithms which calculate them, which we defer to Section 4.3.

While much of this section summaries prior work, the presentation of the bidirectional termination condition (Theorem 10 in Section 4.2.3) in particular is formulated to enable the novel theorems presented in Section 4.3.

4.2.1 *Shortest Paths via the Start Distance Function*

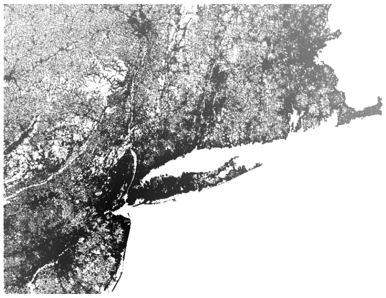


Figure 4.3: The distance function from the start vertex.

Once the distance function d^* is computed, a shortest path to any destination t can be generated trivially by walking backwards to s guided by d^*.

The pioneering pathfinding algorithms of the late 1950s address a generalization of the SPSP problem called the *single-source* problem, where shortest paths are calculated from the start vertex s to all vertices on the graph. They proceed by calculating the *start distance function* $d^* : V \to \mathbb{R}$, which gives the length of the shortest path from s to each vertex v. In other words:

$$d^*(v) = \min_{p \in P_{sv}} \text{len}(p, w), \tag{4.1}$$

where P_{sv} is the set of all paths from s to v, and len is given by (3.1) as the sum of the path's constituent weights with respect to the edge weight function w. Where no paths to v exist, we take $d^*(v) = \infty$. Note that d^* is only well-defined on graphs with no negative-length cycles reachable from s.

Importantly, d^* can also be characterized locally by

$$d^*(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u \in \text{Pred}(v)} d^*(u) + w(e_{uv}) & \text{otherwise,} \end{cases} \tag{4.2}$$

where $\text{Pred}(v)$ yields the predecessor vertices of v, and a vertex $v \neq s$ with no predecessors takes $d^*(v) = \infty$. The distance function is akin to the *value function* in more general decision problems addressed by dynamic programming. The equations (4.2) are the *Bellman equations* [6], which rely on the principle of optimality. This characterization also follows implicitly from early results for the all-pairs problem [107, 5]. Note that while (4.2) is a necessary condition of d^*, it is not sufficient in general. In particular, if w has cycles of length zero, then even though d^* is well-defined, (4.2) admits an infinite number of incorrect solutions for d.

Note that while the distance function d^* necessarily satisfies the equations (4.2), they are not generally a sufficient condition; if a reachable cycle of zero length exists, (4.2) will not have a unique solution.

Reconstructing a Shortest Path from the Distance Function. Calculating the start distance function is only useful for solving the SPSP problem if it can be used to efficiently determine a shortest path. We can show that such a path can be reconstructed by starting at the destination t and progressively prepending the predecessor edge e_{uv} (and vertex u) which locally minimizes $d^*(u) + w(e_{uv})$. Any path constructed in this way is guaranteed to be a shortest path, and this process is guaranteed to terminate if the graph contains no zero-length cycles.

4.2.2 *Approximating d^* via Tensioned Estimates*

How can we compute d^* efficiently over the graph? Consider an approximation function d which satisfies the following four properties:

$$\begin{aligned} d^*(v) &\leq d(v) & \forall v \quad & \text{(4.3a)} \\ d(v) &= 0 & v = s \quad & \text{(4.3b)} \\ \min_{u \in \text{Pred}(v)} d(u) + w(e_{uv}) &\leq d(v) & v \neq s \quad & \text{(4.3c)} \\ d(u) + w(e_{uv}) &\geq d(v) & \forall e_{uv} \quad & \text{(4.3d)} \end{aligned}$$

Conditions (4.3b – 4.3d) follow directly from the local characterization (4.2); in particular, the case where $v \neq s$ has been split into the two equivalent inequalities (4.3c) and (4.3d). In contrast, for now we take the global inequality (4.3a) on faith; we will later revisit the implications of relying on this assumption.

We can show that any estimate d satisfying these properties is the unique distance function d^* via Theorem 7.

Proofs for all theorems in this chapter are located in Appendix C.

Theorem 7 *If $d : V \to \mathbb{R}$ satisfies (4.3), then $d = d^*$.*

The principal method for arriving at an approximation which satisfies (4.3) is via *tensioned estimates*. Consider an arbitrary approximation d which satisfies only (4.3a – 4.3c), and consider the following edge labeling:

$$\text{edge } e_{uv} \text{ is } \textit{tensioned} \text{ iff } d(u) + w(e_{uv}) < d(v). \tag{4.4}$$

Tensioned edges are therefore those that violate (4.3d). A restatement of Theorem 7 is that an approximation d satisfying (4.3a – 4.3c) with no tensioned edges is everywhere correct.

Edge Relaxation. How can we arrive at an approximation satisfying Theorem 7? The principal technique treats the properties (4.3a – 4.3c) as invariants. An initial approximation d is chosen for which the invariants trivially hold, such as $d(v) = \infty \ \forall v \neq s$, which will generally have many edges in tension. The tensioned approximation d is then iteratively improved via *edge relaxation* as described by Ford [37], wherein a tensioned edge e_{uv} is selected and relaxed by setting $d(v) \leftarrow d(u) + w(e_{uv})$. It can be shown that applying this process arbitrarily maintains invariants (4.3a – 4.3c).

It can be further shown that for a finite graph, the number of edge relaxations needed is also finite. The well-known Bellman-Ford method [107, 6, 86] cycles through all edges repeatedly, relaxing all tensioned edges found (at most $|V| - 1$ repetitions are sufficient for convergence). Note that this does not place any requirements on w (other than that d^* must exist, so there must not be any negative-length cycles reachable from s).

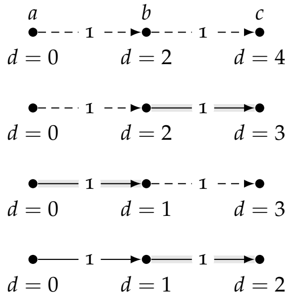


Figure 4.4: Ordering problems. Consider the vertices $a \to b \to c$, with edges e_{ab} and e_{bc} both in tension; if e_{bc} is relaxed before e_{ab}, then e_{bc} will need to be relaxed a second time.

Approximation Soundness. The need for multiple cycles of Bellman-Ford stems from the fact that each edge may need to be relaxed several times. This occurs because relaxing an edge changes the d-value of the destination vertex, which may newly tension downstream edges (see Figure 4.4).

We can exploit our intuition to order relaxations from start to destination in the special case where $w \geq 0$ (note that this requirement is stronger than requiring no reachable negative-length cycles). We can then show that our approximation d is *sound* for a subset of vertices as described by Theorem 8.

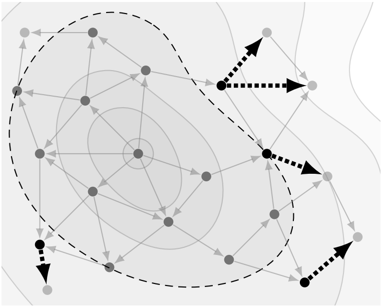


Figure 4.5: Tensioned edge trust region for $w \geq 0$. Contours are of the current estimate d. Currently tensioned edges are bold and dotted.

Theorem 8 *Consider $d : V \to \mathbb{R}$ satisfying (4.3a – 4.3b), and let D be the smallest value $d(u)$ among all tensioned edges e_{uv} (or ∞ if no such edges exist). If $w \geq 0$, any vertex x with $d(x) \leq D$ has $d(x) = d^*(x)$.*

As a result, a given value D creates a region of vertices with values $d(x) \leq D$ that are known to be accurate. This confers two distinct advantages when designing an algorithm: an efficient relaxation ordering, and an early termination condition for single-pair problems.

Efficient Relaxation Ordering. Therefore, all tensioned edges e_{uv} with $d(u) = D$ (of which there must be at least one if any edges are tensioned) can be relaxed immediately, and will never be retensioned. This is exactly the order imposed by the OPEN list in Dijkstra's algorithm [29].

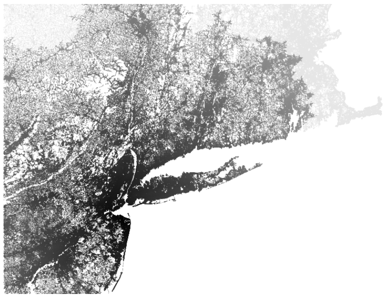


Figure 4.6: Dijkstra's algorithm computes the start distance function d^* to solve the example shortest path problem. Darker vertices have smaller d-values. The algorithm stops upon reaching the destination vertex t after expanding 1,290,820 vertices.

Early Termination. The soundness result shows that once the destination vertex t satisfies $d(t) \leq D$, it has the correct start distance. Since we are only interested in reconstructing a shortest path to t, we are interested in terminating computation of the distance function as early as possible.

However, while Theorem 8 demonstrates that $d(t) = d^*(t)$, it is not by itself insufficient to demonstrate that such a shortest path to t can be reconstructed. An illustration of such a problem case is given in Figure 4.7. This requires the addition of Theorem 9 below. Notably, the proof for Theorem 9 relies on (4.3c).

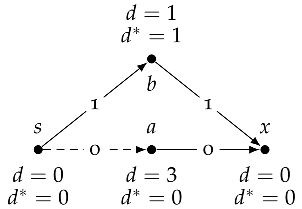


Figure 4.7: Problem case for pathfinding with distance functions in cases where invariant (4.3c) does not hold. Here, d satisfied a and c, with edge e_{sa} tensioned, and $d' = 0$. While the approximation d is *sound* at x via Theorem 8 (i.e. $d(x)$ is correct), the path reconstructed from t is not a shortest path.

Theorem 9 *Consider $d : V \to \mathbb{R}$ satisfying (4.3a – 4.3c), and let D be the smallest value $d(u)$ among all tensioned edges e_{uv} (or ∞ if no such edges exist). If $w \geq 0$, for any vertex x with $d(x) \leq D$, a path reconstructed backwards from x by iteratively selecting a predecessor minimizing $d(u) + w(e_{uv})$ until s is reached is a shortest path from s to x of length $d^*(x)$.*

Armed with Theorems 8 and 9, we can terminate edge relaxation early and reconstruct a shortest path from s to t. This algorithm is listed as "Dijk" (Dijkstra's algorithm) in the results of Figure 4.16.

4.2.3 *Bidirectional Search*

A prominent technique for minimizing pathfinding computation for single-pair problems is bidirectional search (also called "doubletree" search [30]). In a bidirectional algorithm, the distance d_t to the destination is calculated in a growing region around the destination vertex t concurrently with the conventional start distance d_s around s (Figure 4.8). The destination distance function d_t, yielding the distance of a shortest path from each vertex u to t, obeys a complementary definition and local characterization as d_s, with vertex predecessors replaced with successors. Approaches such as edge relaxation (Section 4.2.2) can therefore be used to calculate d_t using a region around t within which shortest paths can be reconstructed.

Loosely speaking, the search can terminate with a shortest path once the two regions intersect. Each search need only descend to a depth a factor of two shallower than a unidirectional search. Therefore, problems where the graph density grows quickly with the search depth are particularly well-suited to bidirectional algorithms. For example, for a roadmap graph embedded in an ambient Euclidean space, the number of expanded vertices in a ball is polynomial in the depth and exponential in the dimensionality of the space. It has also been empirically established that bidirectional approaches are beneficial for instances in which many vertices around both the start and destination vertices are costly (e.g. due to obstacles). This may be because the number of vertices that need to be expanded in these regions do not grow quickly with the search depth.

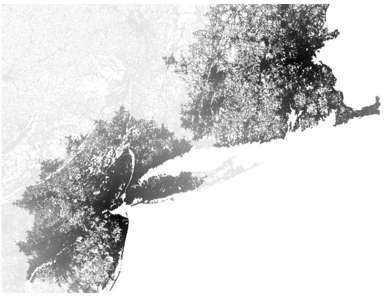


Figure 4.8: The bidirectional Dijkatra's algorithm computes d_s around the start vertex and d_t around the destination vertex. Darker vertices have smaller d-values in their respective regions. The algorithm terminates after expanding a total of 1,178,200 vertices using distance to balance expansions.

The first bidirectional algorithm was proposed by Dantzig [21], and the first precisely described algorithm was presented by Nicholson [89], and the similar Bi-Directional Shortest Path Algorithm (BSPA) was analyzed by Pohl [96]. Implementation of a sound and efficient algorithm turns on when and how to terminate with a shortest path.

A Correct Termination Condition. What happens upon an encounter between the forward and reverse searches? Consider running each search until the first vertex is found that satisfies Theorem 8 in both directions (that is, the first vertex x for which $d_s(x) \leq D_s$ and $d_t(x) \leq D_t$). While Theorem 8 correctly demonstrates that the values $d_s(x)$ and $d_t(x)$ are correct (and Theorem 9 similarly demonstrates that a shortest path can be reconstructed from s to x and also from x to t), this is not sufficient to demonstrate that the shortest path actually passes through x. See Figure 4.9 for a counter-example that illustrates this point.

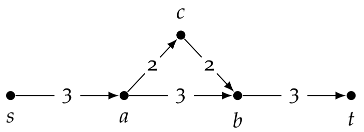


Figure 4.9: Simple illustration of a problem case for terminating a bidirectional search. With a balanced distance criterion, c will be the first vertex expanded in both directions, but it does not lie on the shortest path.

Importantly, is necessary to consider the *edges* connecting the two distance function approximations. A correct termination condition is surprisingly subtle, with several correct variations proposed [89, 32, 94, 45]. What is necessary is a bidirectional equivalent to the completeness-based termination condition from Theorem 9.

There were early incorrect attempts at a sound termination condition [8].

Suppose d_s and d_t are approximations to d_s^* and d_t^*, respectively, with each satisfying (4.3). Let D_s be the minimum u-value among all tensioned edges in d_s (and likewise for d_t). Then we can establish the following proof of a correct termination condition:

This treatment of the bidirectional termination condition is presented differently than in most related work because it will help us formulate an incremental version in Section 4.3.

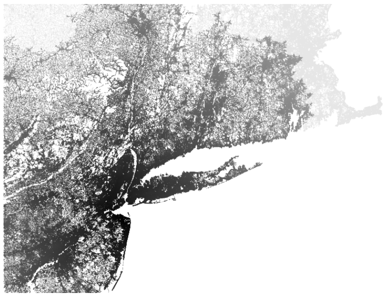
Theorem 10 *Define E_{conn} as the set of all edges e_{uv} such that $d_s(u) \leq D_s$ and $d_t(v) \leq D_t$, and define ℓ_e s.t. $\ell_e(e_{uv}) = d_s(u) + w(e_{uv}) + d_t(v)$. If $w \geq 0$, $s \neq t$, E_{conn} is non-empty, and e_{uv}^* minimizes ℓ_e among E_{conn} with $\ell_e(e_{uv}^*) \leq D_s + D_t$, then $\ell_e(e_{uv}^*)$ is the length of the shortest path, and e_{uv}^* lies on one such path.*

Designing an Efficient Bidirectional Algorithm. In the case where the approximations d_s and d_t are improved via edge relaxation, since $w \geq 0$, by Theorem 8 once an edge e_{uv} becomes included in E_{conn}, its length value $\ell_e(e_{uv})$ will not change. Therefore, it is sufficient for a relaxation algorithm to consider only edges newly added to E_{conn} at each iteration, and keep track of the best edge e^*_{uv} with its value $\ell_e(e^*_{uv})$ found so far. Note also that Theorem 9 allows a shortest path to be constructed from e^*_{uv} buy walking backwards from u to s, and forwards from v to t.

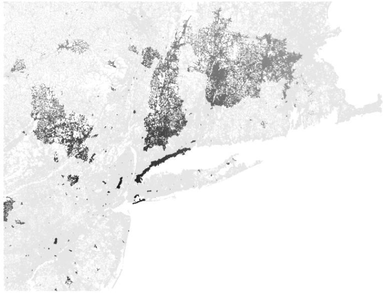
This algorithm is listed as "BiDijk" (bidirectional Dijkstra's algorithm) in the results of Figure 4.16.

4.2.4 *Incremental Search for Dynamic Problems*

The shortest path on a graph is of course intimately tied to the edge weight function w. If the weight function changes from $w^{(1)}$ to $w^{(2)}$, a shortest path $p^{(1)}$ w.r.t. $w^{(1)}$ will generally no longer be a shortest path w.r.t. $w^{(2)}$, even if changes are small and localized. For example, if the weight of edge on $p^{(1)}$ increases, or if the weight of some edge not on p decreases, then some other path $p^{(2)}$ may become shorter than $p^{(1)}$. The *dynamic shortest-path problem* considers finding a shortest path for each of a sequence of input edge weight functions. Figure 4.10 shows an incremental algorithm finding a shortest path quickly for a subsequent planning episode.



(a) Initial episode



(b) Subsequent episode

Figure 4.10: Initial episode: 1,287,897 expansions. Subsequent episode: 391,122 expansions.

We concern ourselves with the *fully dynamic* case, in which arbitrary edge weight changes are allowed. As mentioned in Section 4.1, for our pathfinding problems, we treat edges with infinite weight equivalently with non-existent edges. Therefore, deleting or inserting an edge can be represented by setting its edge weight to or from infinity, respectively. We include here a brief review of the approach underlying common algorithms for the dynamic pathfinding problem.

The dynamic problem has been studied in the literature. Early results demonstrated properties of optimal spanning trees under certain types of update operations [111]. Early algorithms considered restricted problem such as the insert-only problem with constant edge weights [82], or restricted settings such as planar graphs [65]. More general algorithms followed [98, 38, 39]; we describe and generalize the DynamicSWSF-FP algorithm of Ramalingam and Reps below. A review of more general dynamic problems on graphs is available in [33], including for the more general all-pairs problem [26].

We defer for now discussing the problem of integrating heuristic functions with incremental search; see Section 4.4.2 for that discussion.

Problem Definition. Consider a directed graph $G = (V, E)$ as described in Section 4.1. Consider a sequence of pathfinding episodes with different edge weight functions $w^{(1)}, w^{(2)}, \ldots$ with $w^{(i)} : E \rightarrow \mathbb{R}$. The dynamic single-pair shortest-path (SPSP) problem entails finding shortest paths $p^{(1)}, p^{(2)}, \ldots$ between fixed start and destination vertices $s, t \in V$ for each episode.

Approaches to the Dynamic Problem. The simplest class of solutions to the dynamic SPSP problem entails running a conventional SPSP algorithm to compute a solution path for episode in turn. Consider, for example, applying the edge relaxation approach from Section 4.2.2 to compute the start distance function $d_s^{(i)}$ from scratch for each episode's edge weight function $w^{(i)}$.

Investigating this approach more closely reveals an opportunity for a more efficient algorithm. If the changes in the weight function between each episode affect only a subset of the edges, then it is often the case that the value of the distance function computed does not change for a large fraction of the vertices in the graph. In the face of a change in weight function $w^{(i)} \rightarrow w^{(i+1)}$, it is the objective of an *incremental* approach to adapt a previously computed estimate $d_s^{(i)}$ to a new one $d_s^{(i+1)}$ with a minimal amount of additional computation.

Inapplicability of the Tensioned Estimates Approach. The tensioned estimates approach of Section 4.2.2 made use of two clever devices to allow its approximation d to be iteratively improved to the true distance function d^*. First, it relied on a global invariant (4.3a) to ensure that the approximation was never under-consistent. Second, it relied on a decomposition of the local characterization (4.2) into two inequalities (4.3c) and (4.3d), the former treated as a second invariant, and the latter represented not as a constraint but as a labeling of tensioned edges to be iteratively relaxed.

Unfortunately, the incremental setting precludes this approach. Consider a new episode in which the estimate $d^{(i+1)}$ is initialized with the preceding estimate $d^{(i)}$. Since the weight function $w^{(i+1)}$ has changed, there is no guarantee that either of the invariants (4.3a) or (4.3c) still hold. In particular, if edge weights increase, it is common for one or both invariants to be violated, and we can no longer rely on Theorems 8 or 9 to generate sound shortest paths.

A New Approach. The key idea underlying the incremental approach, and the DynamicSWSF-FP [98] algorithm, is to restate the local characterization (4.2) in a different way. In particular:

$$r(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u \in \text{Pred}(v)} d(u) + w(e_{uv}) & \text{otherwise} \end{cases} \tag{4.5a}$$

$$d(v) = r(v) \tag{4.5b}$$

(Here, $r(v)$ represents the "right-hand side" of the local characterization). It is easy to see how (4.5) taken together directly implies (4.2). The motivation behind this decomposition is that the former (4.5a) can be held as an invariant, while the latter (4.5b) can be established iteratively. Prior work adopts the following labeling for each vertex v: if $d(v) = r(v)$, the vertex is *consistent*, whereas inconsistent vertices are either *under-consistent* if $d(v) < r(v)$ or *over-consistent* if $d(v) > r(v)$.

This meaning of consistency for vertices in incremental search is distinct from consistency of heuristic or potential functions.

Importantly, the inapplicability of the global invariant (4.3a) between episodes has implications on the class of weight functions that can be solved by the incremental approach. In particular, when applied to weight functions with cycles of zero length (for which d^* is well-defined), there is no guarantee that a distance function d which satisfies (4.5b) matches d^*. We therefore restrict the class of weight functions considered to those with only positive length cycles.

Indeed, later in Theorem 11 we will restrict w further so that $w > 0$ to garner the benefits of approximation soundness and early termination. However, we conjecture that the incremental algorithm is correct even without $w > 0$ (as long as no non-positive cycles exist) as long as the algorithm is not terminated early.

Approximation Soundness. In Section 4.2.2, we showed that in cases with $w \geq 0$, the minimal value k defines a trust region that can be used both to order edge relaxations as well as inform a termination condition after which a correct path can be reconstructed. We can develop a similar argument about the soundness of an approximation in the dynamic case. In particular, given an arbitrary estimate d (and the corresponding function r as defined by (4.5a), this argument makes use of a third function k defined as:

$$k(v) = \min\left[d(v), r(v)\right]. \tag{4.6}$$

We can then establish the following central soundness result for incremental search:

Theorem 11 *Consider $d : V \rightarrow \mathbb{R}$, with r satisfying (4.5a), and let K be the smallest value $k(v)$ among all inconsistent vertices (or ∞ if no such vertices exist). If $w > 0$, any consistent vertex x with $d(x) \leq K$ has $d(x) = d^*(x)$. Further, walking backwards from x choosing predecessors that minimize $d(u) + w(e_{uv})$ terminates at s with a shortest path of length $d^*(x)$.*

This theorem enables the same two efficiency advantages as in the relaxation approach, in the case where $w > 0$. It provides that a vertex rendered consistent during an iteration (by setting $d(x) \leftarrow r(x)$) with $d(x) \leq K$ has the correct value and will never be revisited. It also enables an algorithm to terminate early once the destination vertex t becomes consistent, as exploited by the Lifelong Planning A* algorithm [67].

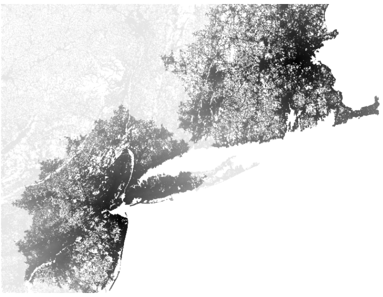
4.2.5 *Algorithm Design*

In the same way that the edge relaxation decomposition (4.3) allowed way to progressively update an estimate d via a priority queue, the decomposition (4.5) affords the same opportunity. The DynamicSWSF-FP algorithm [98] maintains a queue Q of inconsistent vertices, and processes them in the order given by the priority key (4.6) used in the soundness result in Theorem 11.

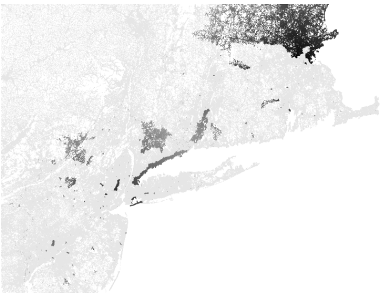
One key aspect of the algorithm is the way underconsistent vertices are handled. Upon encountering an underconsistent vertex ($d(v) < r(v)$), the algorithm applies $d(v) \leftarrow \infty$ (if the vertex remains inconsistent, it would be rendered consistent in a future iteration if necessary). This ensures that each vertex is processed at most twice during each planning episode. This algorithm is listed as "Dyn-SWSF" (Dynamic SWSF-FP) in the results of Figure 4.16.

4.3 *Incremental Bidirectional Search*

The principal motivation behind the Incremental Bidirectional (IBiD) search algorithm is to leverage the early termination efficiency of a bidirectional search with the efficiency of an incremental search for dynamic single-pair shortest path problems. An example of IBiD is shown in Figure 4.11.



(a) Initial search



(b) Replan search

Figure 4.11: Initial search: 1,181,616 expansions. Replan: 262,422 expansions.

Because it is derived from DynamicSWSF-FP, the IBiD algorithm is applicable to graphs with $w > 0$. (If zero weights are present, the algorithm can be adapted by transforming w to use lexicographically sorted tuples as developed by LPA* [67], described in Section 4.4.2.) Note also that as an extension of Bidirectional Dijkstra's algorithm [45], the algorithm presented requires $s \neq t$. If the start and destination vertices coincide, an empty path can be returned in a preprocessing step.

4.3.1 *Bidirectional Termination with Incremental Distance Functions*

Consider a graph endowed with two distance functions, a start distance approximation d_s, and a destination distance approximation d_t. Since this is an incremental setting, we can make use of the formulation of Theorem 11 from Section 4.2.4 to establish the correctness of these approximations for certain vertices.

It is essential that the bidirectional termination condition from Theorem 10 be adapted to handle these incrementally maintained distance functions. The central theorem that enables the IBiD algorithm is:

Theorem 12 *Consider a graph with* $w > 0$ *and with* $s \neq t$*. Consider a*

Algorithm 5 As a bidirectional algorithm, IBiD conducts two independent DynamicSWSF-FP searches, one computing distance from the start vertex s, and the other computing distance to the destination vertex t.

```
procedure InitializeStart()
    for all $v \in V$ do
        $d_s(v) \leftarrow \infty$;  $r_s(v) \leftarrow \infty$
    $r_s(s) \leftarrow 0$
    $Q_s \leftarrow \{s\}$        ▷ key for $v$ : $\min\big(r_s(v), d_s(v)\big)$
    ProcessStartQueue()
procedure UpdateStartDistance($v$)
    if $v \neq s$ then
        $r_s(v) \leftarrow \min_{u \in \mathrm{Pred}(v)} \big(d_s(u) + w(u,v)\big)$
    Ensure $v \in Q_s$ iff $d_s(v) \neq r_s(v)$
procedure ProcessStartQueue()
    $u \leftarrow Q_s$.Pop()
    if $r_s(u) < d_s(u)$ then        ▷ over-consistent
        $d_s(u) \leftarrow r_s(u)$
        for all $v \in \mathrm{Succ}(u)$ do
            UpdateStartDistance($v$)
    else        ▷ under-consistent
        $d_s(u) \leftarrow \infty$
        for all $v \in \mathrm{Succ}(u) \cup u$ do
            UpdateStartDistance($v$)
procedure InitializeDest()
    for all $v \in V$ do
        $d_t(v) \leftarrow \infty$;  $r_t(v) \leftarrow \infty$
    $r_t(t) \leftarrow 0$
    $Q_t \leftarrow \{t\}$        ▷ key for $v$ : $\min\big(r_t(v), d_t(v)\big)$
    ProcessDestQueue()
procedure UpdateDestDistance($u$)
    if $u \neq t$ then
        $r_t(u) \leftarrow \min_{v \in \mathrm{Succ}(u)} \big(w(u,v) + d_t(v)\big)$
    Ensure $u \in Q_t$ iff $d_t(u) \neq r_t(u)$
procedure ProcessDestQueue()
    $v \leftarrow Q_t$.Pop()
    if $r_t(v) < d_t(v)$ then        ▷ over-consistent
        $d_t(v) \leftarrow r_t(v)$
        for all $u \in \mathrm{Pred}(v)$ do
            UpdateDestDistance($u$)
    else        ▷ under-consistent
        $d_t(v) \leftarrow \infty$
        for all $u \in \mathrm{Pred}(v) \cup v$ do
            UpdateDestDistance($u$)
```

start distance approximation $d_s : V \to \mathbb{R}$, *with* r_s *satisfying (4.5a), and let* K_s *be the smallest value* $k_s(v)$ *among all inconsistent vertices, or* ∞ *if no such vertices exist; likewise, consider the same for a destination distance approximation* d_t *and its accompanying* r_t *and* K_t*.*

Define E_{conn} *as the set of all edges* e_{uv} *such that u is s-consistent with* $d_s(u) \leq K_s$ *and v is t-consistent with* $d_t(v) \leq K_t$*, and define* ℓ_e *s.t.* $\ell_e(e_{uv}) = d_s(u) + w(e_{uv}) + d_t(v)$*. If* E_{conn} *is non-empty, and* e^*_{uv} *minimizes* ℓ_e *among* E_{conn} *with* $\ell_e(e^*_{uv}) \leq K_s + K_t$*, then* $\ell_e(e^*_{uv})$ *is the length of the shortest path, and* e^*_{uv} *lies on one such path.*

Note that the first half of the conditions from Theorem 12 is drawn from the soundness conditions for incremental search from Theorem 11, while the second half resembles the bidirectional termination conditions from Theorem 10.

4.3.2 *Algorithm Design*

Simultaneous Incremental Searches. The IBiD algorithm runs two independent conventional DynamicSWSF-FP [98] searches simultaneously, one from the start vertex s and one from the destination vertex

Algorithm 6 IBiD Outline

```
procedure Main()
    InitializeStart(); InitializeDest()
    $Q_c \leftarrow \emptyset$                    ▷ key for $(u,v) : d_s(u) + w(u,v) + d_t(v)$
    loop
        while not TerminationCondition() do
            if $Q_s$.TopKey $<$ $Q_t$.TopKey then    ▷ prioritize arbitrarily
                ProcessStartQueue($u$)
            else
                ProcessDestQueue($u$)
            Ensure $(u,v) \in Q_c$ iff $u \neq Q_s$, $v \neq Q_t$, key $\neq \infty$
        $(u_c, v_c) \leftarrow Q_c$.Top
        $\pi \leftarrow$ (walk $d_s$ from $u_c$ to $s$) $\cup$ (walk $d_t$ from $v_c$ to $t$)
        wait for edges $(u,v) \in E_{\text{delta}}$ with changed weights $w(u,v)$
        NotifyWeightChanges($E_{\text{delta}}$)
procedure NotifyWeightChanges($E_{\text{delta}}$)
    for all $(u,v) \in E_{\text{delta}}$ do
        UpdateStartDistance($v$)
        UpdateDestDistance($u$)
    Ensure $(u,v) \in Q_c$ iff $u \neq Q_s$, $v \neq Q_t$, key $\neq \infty$
```

t, with each maintaining a separate priority queue of inconsistent vertices Q_s and Q_t respectively (Algorithm 5).

As with a conventional bidirectional search, the two sides can be alternated arbitrarily, except that the each queue must be processed once upon initialization so that $d_s(s) = 0$ and $d_t(t) = 0$.

Remembering Connections. Theorem 12 has ramifications when designing an algorithm. In the case of bidirectional search described in Section 4.2.3, the algorithm could take a shortcut since D-values always decreased, so it was not necessary to remember older edges when better ones were found. In the incremental case, where the start and destination distance functions d_s and d_t may both increase and decrease between episodes, it is no longer sufficient to remember only the best connection found so far.

Instead, we track all potential connection edges in a connection queue Q_c. Members of this queue are edges on the graph e_{uv} for which vertex u is s-consistent, and vertex v is t-consistent. This queue is sorted by the key $d_s(u) + w(u,v) + d_t(v)$. As an optimization, the queue does not include edges for which the key is infinite, since that corresponds to a non-existent path. The resulting algorithm outline is shown in Algorithm 6.

Termination Condition. The termination condition from Theorem 12 is captured in Algorithm 7. The first conditional handles cases where no finite path exists. This algorithm is listed as "IBiD" in the results of Figure 4.16.

Algorithm 7 IBiD Termination Condition

```
function TerminationCondition()
    if [$Q_s$.Empty and $d_s(t) = \infty$] or [$Q_t$.Empty and $d_t(s) = \infty$] then
        return True                                  ▷ no solution path
    $(u_c, v_c) \leftarrow Q_c$.TopKey               ▷ return False if $Q_c$ empty
    if $Q_s$.TopKey + $Q_t$.TopKey $< d_s(u_c) + w(u_c, v_c) + d_t(v_c)$ then
        return False
    if $Q_s$.TopKey $< d_s(u_c)$ or $Q_t$.TopKey $< d_t(v_c)$ then
        return False
    return True
```

4.4 *Heuristic Search*

When examining how to assemble a heuristic-informed algorithm that integrates bidirectional with incremental methods, it is instructive to examine how these efforts have been addressed in the past.

Review of Heuristic Methods. Heuristic methods such as the Graph Traverser [31] were originally applied to pathfinding problems in order to find non-optimal solutions more economically. These unidirectional methods proceed similarly to Dijkstra's algorithm, but instead of prioritizing Open vertices based on their start distance d_s, they use a destination-directed heuristic function h_t. Hart, Nilsson, and Raphael [47] established that these approaches can be combined ($d_s + h_t$) to yield an admissible algorithm (A*) for the shortest-path problem, as long as h_t meets certain conditions. See Figure 4.12. We show the results of unidirectional heuristic search as "A*" in the results of Figure 4.16.

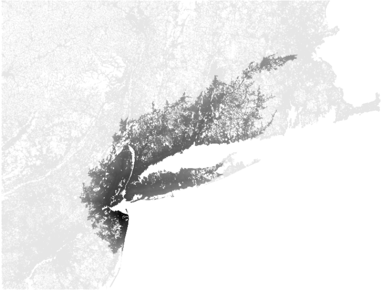


Figure 4.12: A* search. 532,880 expansions.

4.4.1 *Heuristic Methods in Bidirectional Search*

Attempts to provide a bidirectional algorithm which incorporates heuristic estimates generally take one of three approaches, which we survey here.

First, "front-to-back" methods such as Pohl's Bidirectional Heuristic Path Algorithm (BHPA) [96] conduct two conventional heuristic-informed searches (i.e. the start search directed to the destination vertex, and vice versa). This approach suffers from the problem that the two sets of CLOSED vertices must overlap substantially before the search can be terminated, and therefore the benefit of the bidirectional search is not fully realized.

Second, "front-to-front" methods exploit a pairwise heuristic function $h(u, v)$ evaluated over every pair of vertices on each search's OPEN list, and the vertices representing the shortest potential path are expanded. Early approaches with inflated heuristics [30] did not yield promising results. The later Bidirectional Heuristic Front-to-Front Algorithm (BHFFA) [22] does show an improved number of vertex expansions. While this approach does not suffer from the region overlap problem inherent in front-to-back methods, it instead incurs the very high cost of computing full pairwise heuristics, and updating all vertices on OPEN after each vertex expansion.

An earlier version of BHFFA [23] was shown to be incorrect due in large part to the difficulty establishing a correct bidirectional termination condition.

The third approach derives from an alternative interpretation of heuristic search algorithms first proposed by Ikeda et. al. [54]. Under this interpretation, a heuristic-informed A* search is equivalent to an uninformed Dijkstra's search under a particular transformation of the edge weights using the heuristic function as a vertex potential function. Therefore, because front-to-back methods use different heuristics for the forward and reverse searches, they are effectively searching graphs with different weight functions. The proposed solution is to arrive at a single consistent potential function, which allows the transformed pathfinding problem to be solved using the uninformed bidirectional algorithm described in Section 4.2.3.

Potential-Adjusted Weight Functions. Consider an arbitrary vertex potential function $b : V \rightarrow \mathbb{R}$, and consider the following transformation of a weight function w into transformed weight function w_b:

$$w_b(e_{uv}) = w(e_{uv}) - [b(u) - b(v)] \,. \tag{4.7}$$

The effect of this transformation on the lengths of paths over G bears investigation. Consider a path p_{xy} from x to y which has length ℓ w.r.t. w. What is its weight ℓ_b w.r.t. w_b? A simple summation reveals that

$$\ell_b(p_{xy}) = \ell(p_{xy}) - [b(x) - b(y)] \,. \tag{4.8}$$

That is, the difference between the initial and transformed lengths of the path from x to y is independent of the path itself (its constituent vertices and edges), and only a function of the endpoint vertices. In particular, this implies that a shortest path from x to y w.r.t. w_b remains a shortest path w.r.t. w. Therefore, any admissible shortest path algorithm can be used to solve w.r.t. w_b. Note that the transformation does not change the length of any cycles in G. Note also that as a consequence of (4.8), the original and transformed start distance functions are related by

$$d_b^*(v) = d^*(v) - [b(s) - b(v)] \,. \tag{4.9}$$

Benefit of Potential Adjustment. If the ordering of solution paths does not change under the potential function transformation, what benefit would be obtained by conducting the search on the transformed weight function? Indeed, computing the full distance function d_s over w_b, e.g. by edge relaxation, appears just as expensive.

The key insight is that under an appropriately chosen potential function, many shortest path algorithms are able to terminate more quickly. To see why, consider the potential function b coinciding with the destination heuristic function h_t used in A*. By (4.7), the weight w_b of a directed edge e_{uv} which makes progress towards the destination (so that $b(v) < b(u)$) will have its transformed weight be reduced, whereas an edge in the incorrect direction will have its transformed weight increase. The transformed quantity w_b from (4.7) can be considered the *excess weight* of each edge, relative to the expected weight given the progress measured by the heuristic function (this quantity has also been called the "waste" of the edge [94]). A pathfinding algorithm therefore searches only in the excess weight space, and is therefore biased to explore paths which make progress towards the destination vertex. In effect, while the transformation does not affect the relative ordering of paths, it does shift the weight between edges on paths to accentuate poor choices to allow for early termination.

Requirements on the Potential Function. The argument that ordering of paths (and therefore shortest path(s)) is invariant to the transformation is true for any arbitrary potential function. However, in order to be useful, we must be able to solve the transformed problem efficiently (i.e. with early termination). As we saw for edge relaxation in Section 4.2.2, our soundness result from Theorem 9 requires that the edge weight function be nonnegative. This induces the requirement that the vertex potential b to be *consistent* (also known as *monotone* or *dual feasible*), i.e.

$$b(u) + w(e_{uv}) \geq b(v) \quad \forall e_{uv} \in E. \tag{4.10}$$

If (4.10) holds, then the efficient ordering and early termination for edge relaxation (i.e. Dijkstra's algorithm) can be applied to the transformed problem w_b, and the returned path is also a shortest path w.r.t. w.

Potential Functions for Bidirectional Search. Treating heuristic search as such a potential function transformation allows Ikeda et. al. [54] to apply the same idea to bidirectional search. In particular, if one commits to a single potential function b, then the conventional bidirectional algorithm (Section 4.2.3) can be applied directly. The question is, what potential function should we use?

A bidirectional search customarily has available two heuristic functions, one $h_t(v)$ that approximates the length of paths from v to the destination vertex t, and one $h_s(v)$ that approximates the length of paths from the start vertex to v. The most commonly used potential function [54, 45] is simply the average of the two:

$$b(v) = \frac{h_t(v) - h_s(v)}{2}. \tag{4.11}$$

Note that the start heuristic value is negated; when the start heuristic is used during the reverse search, it is applied to predecessor vertices. It can be shown that if h_s and h_t are consistent (i.e. satisfy (4.10)), b is also consistent.

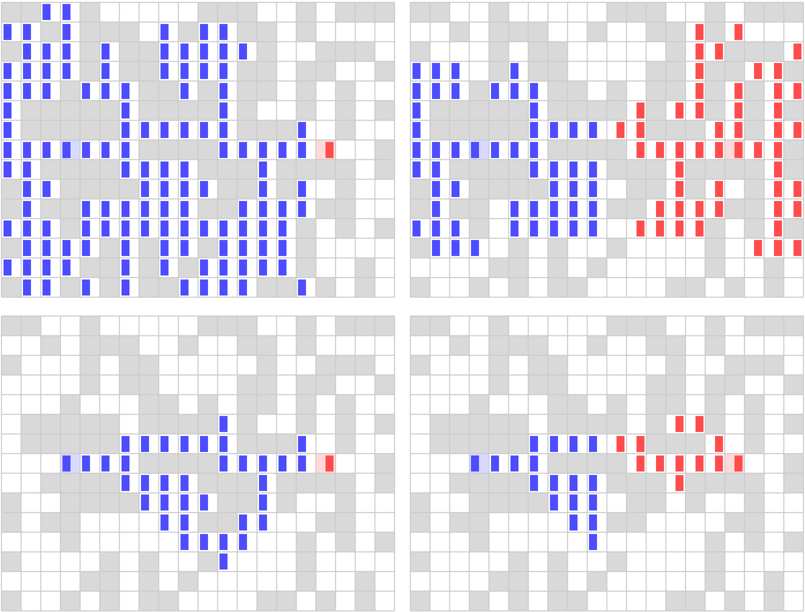


Figure 4.13: Illustration of behavior of bidirectional heuristic search on a shortest path problem reproduced from [67]. Vertices that are start-expanded are shown in blue, while those that are destination-expanded are shown in red. At left, the algorithm performs only start-side expansions, which approximates the behavior of a unidirectional algorithm. At right, the algorithm performs distance-balanced expansions between the two searches. The top row shows the behavior of the algorithm with no heuristic potential function. Start and destination heuristic functions are available, and their effect are shown at bottom. The unidirectional search uses the destination heuristic h_t as its potential function, while the bidirectional search uses the averaged potential function (4.11).

Figure 4.13 shows the behavior of the bidirectional heuristic search algorithm on a path planning problem in an eight-connected grid world. This example problem is reproduced from [67]. The figure shows the effect of the combination of bidirectional and heuristic search.

Figure 4.14 shows the result of conducting a bidirectional search using this averaged potential function for the example road network problem. This algorithm is listed as “H-BiDijk” (heuristic bidirectional Dijkstra’s) in the results of Figure 4.16.

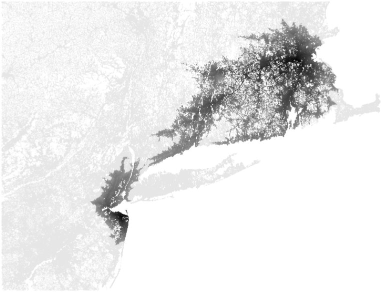


Figure 4.14: Bidirectional A* search. 515,588 expansions.

4.4.2 Heuristic Methods in Incremental Search

Efforts to integrate heuristics into incremental search algorithms have taken a similar potential function approach. Koenig et. al. developed Lifelong Planning A* [67], which implicitly applies this transformation and then applies the DynamicSWSF-FP incremental algorithm [98] from Section 4.2.4 to solve the dynamic problem. This approach has resulted in a number of commonly used algorithms such as D* Lite [66], Field D* [36], and Anytime D* [80].

However, as discussed in Section 4.2.4, the underlying incremental algorithm has a stricter requirements: while Dijkstra's algorithm requires $w \geq 0$, Theorem 11 governing the incremental algorithm requires $w > 0$. As a result, even in the case where the potential function is consistent via (4.10), the transformed weight function w_b will not generally satisfy $w_b > 0$. This occurs along all edges for which the weight is identical to the potential function difference – exactly the expected scenario when the potential function is accurate.

We consider only optimal algorithms (with admissible heuristics); for an extension to inflated search, see [1].

Avoiding the Zero Weight with Sorted Tuples. LPA* solves the zero weight problem by effectively conducting the incremental search over a different set of edge weights – instead of the reals $w : E \rightarrow \mathbb{R}^{>0}$, each transformed edge weight is instead a member of a product space

$$w_b \; : \; E \; \rightarrow \; \mathbb{R}^{\geq 0} \times \mathbb{R}^{>0}. \tag{4.12}$$

with the transformation defined as

$$w_b(e_{uv}) = \big[w(e_{uv}) - [b(u) - b(v)];\; w(e_{uv}) \big]. \tag{4.13}$$

Note that the first element of the tuple coincides with the non-incremental potential function transformation from (4.7). Elements of the product space are sorted lexicographically; that is, their comparison only considers the first component, with the exception that when the first component is equal, the second component is then considered. Summing two elements is performed component-wise. Note that since the we already assert that $w > 0$ for incremental domains, the second component of w_b is everywhere positive, and therefore $w_b > (0, 0)$.

Note that in order to maintain $w_b > 0$, the second element need only be positive, and could even be constant (e.g. unity). We keep the value $w(e_{uv})$ for consistency with LPA*.

Equivalence to Lifelong Planning A.* Our treatment of the transformed edge weight function w_b and the key function k differs from the original presentation [67]. Here, we will briefly show that they are equivalent.

Consider two algorithms conducting a search over a graph: (a) LPA* and (b) DynamicSWSF-FP with its incremental key (4.6) and using the transformed edge weight function (4.13). LPA* maintains the two values $g'(v)$ and $rhs'(v)$ for each vertex, as well as the key $k'(v)$ for sorting its priority queue (we use the prime notation for LPA* values):

$$g'(v) \quad \text{and} \quad rhs'(v) = \min_{u \in \text{Pred}(v)} g'(u) + w(e_{uv}), \tag{4.14}$$

$$k'(v) = \left[\min\left(g'(v), rhs'(v)\right) + b(v);\ \min\left(g'(v), rhs'(v)\right)\right]. \tag{4.15}$$

In contrast, our algorithm (b) builds its distance function $d(v)$ over the transformed tuple weight function w_b from (4.13). We show the equivalence of these two approaches via the following invariant:

$$d(v) = \left[g'(v) + b(v) - b(s);\ g'(v)\right]. \tag{4.16}$$

Note that this is trivially true at the start of each algorithm, when $g'(v) = \infty$ and $d(v) = [\infty; \infty]$ for all vertices.

We next derive the complementary equivalence between $r(v)$ and $rhs'(v)$ for $v \neq s$ ($r(s) = rhs'(s) = 0$ trivially).

$$\begin{aligned} r(v) &= \min_{u \in \text{Pred}(v)} d(u) + w_b(e_{uv}) && (4.17a) \\ &= \min_{u \in \text{Pred}(v)} \left[g'(u) + w(e_{uv}) + b(v) - b(s);\ g'(u) + w(e_{uv})\right] && (4.17b) \\ &= \left[rhs'(v) + b(v) - b(s);\ rhs'(v)\right] && (4.17c) \end{aligned}$$

As a result, we can write the key k used by our DynamicSWSF-FP algorithm over the transformed edge weights as follows:

$$\begin{aligned} k(v) &= \min(d(v), r(v)) && (4.18a) \\ &= \min\left(\left[g'(v) + b(v) - b(s);\ g'(v)\right],\right. && (4.18b) \\ &\qquad \left.\left[rhs'(v) + b(v) - b(s);\ rhs'(v)\right]\right) && (4.18c) \\ &= [-b(s); 0] + \left[\min\left(g'(v), rhs'(v)\right) + b(v);\ \min\left(g'(v), rhs'(v)\right)\right] && (4.18d) \\ &= [-b(s); 0] + k'(v). && (4.18e) \end{aligned}$$

This shows that the tuple keys used to sort the priority queues for the two algorithms are identical, up to the constant $[-b(s); 0]$. Therefore, both algorithms will select the same inconsistent vertex at each iteration, and it follows from (4.16) and (4.17c) will make the same determination w.r.t. over vs. under consistency and set $g'(v)$ and $d(v)$ respectively in order to maintain the invariant (4.16).

This algorithm is listed as "LPA*" in the results of Figure 4.16.

4.4.3 *Heuristic IBiD*

The potential function transformation approach is applicable to Dijkstra's algorithm (yielding A*), Bidirectional Dijkstra's algorithm (yielding H-BiDijk), and DynamicSWSF-FP (yielding LPA*). Therefore, it is natural to apply the same approach to IBiD, which is both bidirectional and incremental.

As with the core IBiD algorithm presented in Section 4.3, the heuristic variant requires $w > 0$ and $s \neq t$. Given consistent heuristics h_s and h_t, form the averaged potential function b as described in Section 4.11. This induces a transformed edge weight function w_b according to (4.13). The Heuristic IBiD algorithm consists of running IBiD (Algorithm 6) over these transformed edge weights.

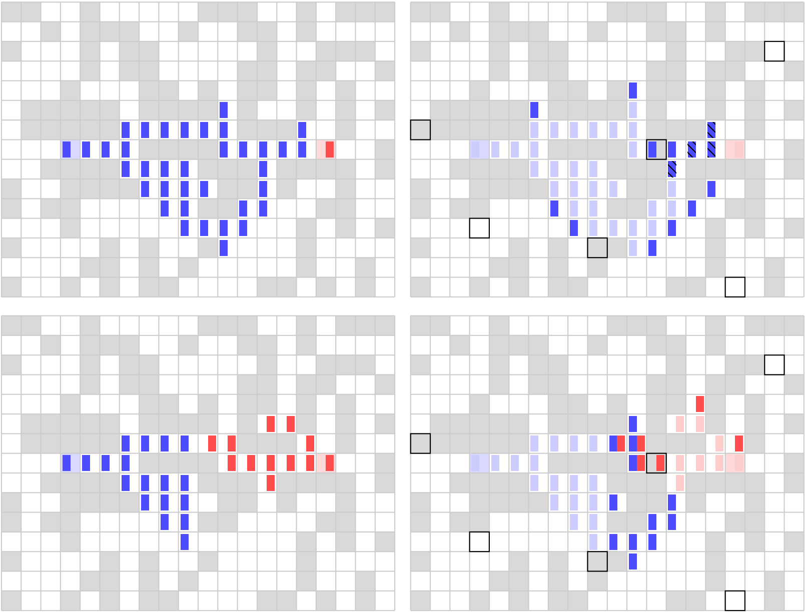


Figure 4.15: Comparison of Heuristic IBiD parameters (expansion balancers and potential functions) on a dynamic grid world pathfinding problem reproduced from [67]. Heuristic IBiD with only start-side expansions and a destination-side heuristic (top) proceeds identically to Lifelong Planning A*, performing 37 expansions on the original world (left) followed by 18 expansions over 14 vertices on the chanced world (right). Heuristic IBiD with distance-balanced expansions and an average potential (bottom) performs 30 expansions on the original world followed by 18 expansions over 15 vertices on the changed world.

The behavior of Heuristic IBiD on a dynamic problem is shown in Figure 4.15. This algorithm is listed as "H-IBiD" (Heuristic IBiD) in the results of Figure 4.16.

4.5 *Experimental Results*

We performed an experimental comparison of the various pathfinding algorithms described in this chapter on a set of incremental problem instances drawn from an approximate road network of the Northeast USA from the public dataset from the 9th DIMACS Implementation Challenge [27]. The directed graph consists of comprises 1,524,453 vertices and 3,868,020 edges. Each edge is annotated with a transit time. The shortest path problem between a pair of start and destination locations therefore minimizes the total transit time between them.

We compared a total of eight algorithms on this problem, consisting of each combination of { unidirectional, bidirectional }, {

complete, incremental }, and { uninformed, heuristic informed }. A complete table of algorithms is given in Table 4.2.

Bidir	Heur	Inc	Name	Full name
			Dijk	Dijkstra's algorithm
X			BiDijk	Bidirectional Dijkstra's algorithm
	X		A*	A* search
X	X		H-BiDijk	Heuristic bidirectional Dijkstra's
		X	DynSWSF	DynamicSWSF-FP
X		X	IBiD	Incremental Bidirectional Dijkstra's
	X	X	LPA*	Lifelong Planning A*
X	X	X	H-BiDijk	Heuristic IBiD

Table 4.2: Table of algorithms compared in the Northeast USA dynamic pathfinding problem. Each algorithm is marked as bidirectional ("Bidir"), heuristic-informed ("Heur"), and/or incremental ("Inc").

Heuristics. For algorithms that use heuristics (A*, H-BiDijk, LPA*, and H-IBiD), we compute them using 2D Euclidean distance. We therefore project the input data given in geographic coordinates onto the 2D plane using the latitude/longitude scale at the midpoint latitude (41.25°N). This results in a projection error of less than 0.3%. Because edge weights are transit times, we use the maximum transit speed over the entire graph as the conversion factor (30.11 m/s), so that the resulting heuristic is consistent.

For the unidirectional algorithms (A* and LPA*), the destination heuristic function h_t was used. For the bidirectional algorithms (H-BiDijk and H-IBiD), the averaged potential function b from (4.11) was used.

Dynamic Edge Weights. For the dynamic setting, we created instances on the graph derived from a traffic analogy. Each edge may either be *unblocked*, taking its original transit time as its edge weight, or *blocked*, taking an infinite edge weight. Between each planning episode, each edge transitions independently between blocked and unblocked with some probability. We ran experiments across four problem classes with different probability parameters, ranging from P1 (infrequent changes) to P4 (frequent changes), with transition probabilities given in Table 4.3. The expected steady-state proportion of blocked edges on the graph follows from

$$P_{\text{blocked}} = \frac{P_{\text{block}}}{P_{\text{block}} + P_{\text{unblock}}}, \tag{4.19}$$

and the transition probabilities were chosen so that this expected proportion is constant across the problem classes at 0.002.

Each problem instance consisted of a start/destination vertex pair chosen uniformly at random from the vertex set, and a ran-

Problem	P_{block}	P_{unblock}	P_{blocked}
P1	0.0001	0.0499	0.002
P2	0.0002	0.0998	0.002
P3	0.0005	0.2495	0.002
P4	0.0010	0.4990	0.002

Table 4.3: Traffic transition parameters for each edge of the Northeast USA graph for the four problem classes P1 – P4.

domly chosen initial distribution of blocked edges (each edge initially blocked with the steady-state 0.002 probability). We compared all eight algorithms against the same set of 1600 problem instances formed by all combinations of the four different traffic transition problem classes, 20 different start/destination pairs and 20 different traffic initial distributions.

Each problem instance consists of ten planning episodes, with traffic transition probability governed by the parameters in Table 4.3 between each episode. The four non-incremental algorithms were restarted from scratch for each episode. The four incremental algorithms were notified of the changed edge weights before each episode.

Discussion of Results. The results of this experiment are shown in Figure 4.16. The results are primarily partitioned by algorithm. For each algorithm, the results from the 1600 instances are broken down as follows. Since the four different problem classes affect only the edge weight transition probabilities for non-initial (replanning) episodes, summary statistics are shown separately for only the 400 unique initial episodes, which are common across all problem classes. The statistics for the replanning episodes are then averaged across the remaining nine episodes, and are broken down by problem class.

Effect of Incremental Search. The first observation from the data is that each of the first four non-incremental algorithms show identical performance between the initial and replanning episodes. This is expected, because the joint edge weight distribution for each episode treated individually is identical (edges blocked with independent probability 0.002), and the non-incremental algorithms address each episode from scratch.

The performance of the four incremental algorithms on the initial episodes is identical to that of their non-incremental counterparts. In contrast, the results largely show significantly improved performance on the replanning episodes relative to the initial episodes. This demonstrates the effectiveness of incremental algorithms. As expected, as the frequency of edge weight changes decreases (most infrequent for P1) so that there are fewer changed edge weights be-

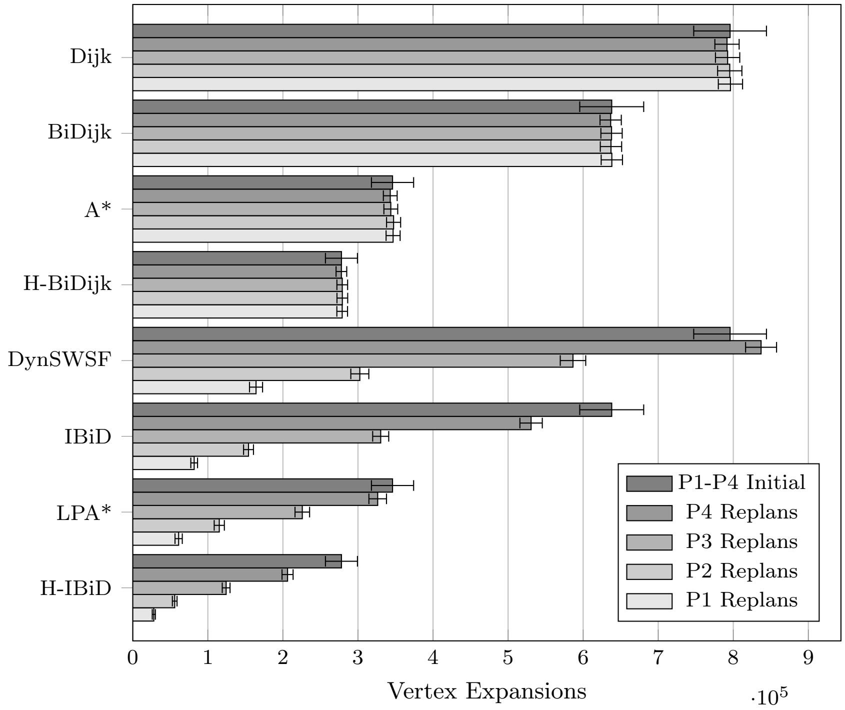


Figure 4.16: Expected number of vertex expansions required to solve a single-pair shortest path query on a Northeast USA road network. Four problems P1-P4 were considered with different numbers of expected edges with traffic changes between episodes (P1: few changes, P4 many changes). The dataset consists of 400 instances of 10 episodes each deriving from 20 randomly selected vertex pairs and 20 random traffic seeds. The "Initial" series captures the first episode for each problem instance (with identical behavior between problems P1-P4 because no traffic changes affect the initial episode). The "Replans" series capture an average over the remaining nine episodes. Note that (a) only the latter four incremental planners see savings during replanning, and (b) the initial episode is identical between each incremental planner and its corresponding complete cousin.

tween each episode, the benefits of incremental search are magnified. On the other hand, the results for DynamicSWSF-FP on the P4 problem class (with many changed edge weights) actually shows an increased number of vertex expansions – this can happen because incremental algorithms can visit a vertex more than once (at most twice with $w > 0$).

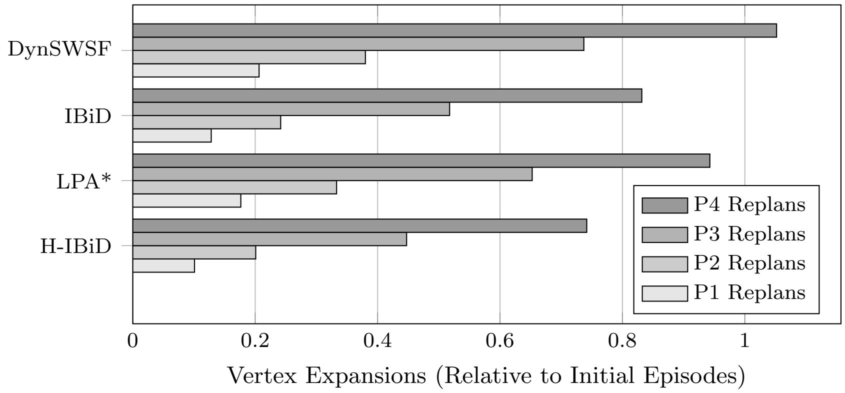


Figure 4.17: Relative performance improvement for replanning episodes relative to initial episodes for each incremental algorithm on the dynamic road network problem.

Comparison of Bidirectional and Heuristic Methods. For this problem domain, it appears that the benefit of adding a destination heuristic

(A*) is more significant than the benefit of conducting a bidirectional search (BiDijk). However, in both the complete and incremental settings, combining both techniques outperforms either one individually.

Benefit of Incremental Search. Figure 4.17 shows the same results for the replanning episodes of each incremental algorithm normalized to its initial episode. In other words, it shows the benefit of incremental search for each algorithm and problem class. In particular, this shows that not only does the bidirectional and heuristic-informed algorithm (H-BiDijk) outperform the other non-incremental planners as shown in Figure 4.16, but also that integrating incremental search (H-IBiD) yields the largest additional relative speedup. It appears that for this problem domain, the combination of these three techniques is particularly effective.

4.6 Available Implementations

Many existing implementations of bidirectional, incremental [2], and heuristic [79] pathfinding algorithms exist. We provide an implementation of the algorithms in this chapter for the Boost Graph Library [108] at the following address:

`https://github.com/personalrobotics/lemur/tree/master/pr_bgl`

5

Maximizing Utility in Motion Planning

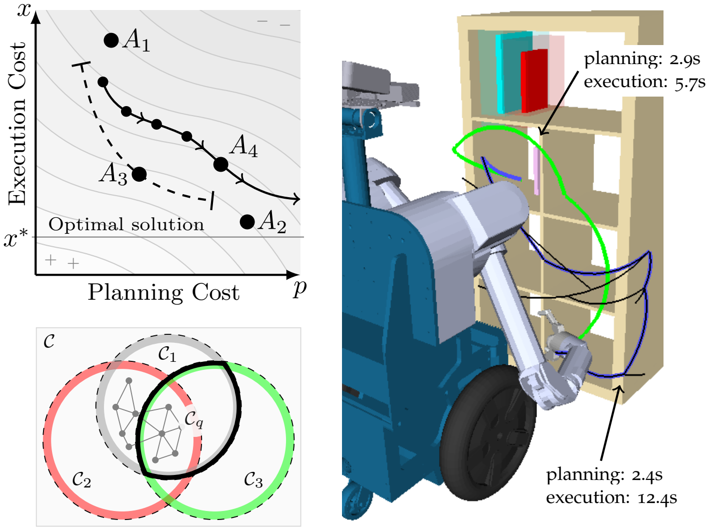
Up to this point, we have considered general optimization problems on graphs: given an edge weight function $w : E \rightarrow \mathbb{R}$ which is expensive to evaluate, how can arrive at a minimizing solution path as quickly as possible? We saw in Chapters 3 that the LazySP algorithm can take advantage of different edge selectors in order to minimize the number of such evaluations, and Chapter 4 discussed the IBiD incremental search algorithm for efficiently solving the intervening dynamic pathfinding problem.

In this chapter, we will ask a more fundamental question – what objective should we be optimizing in the first place? Chapter 2 laid out the functionals x_{valid} and x_{int} capturing the feasible and optimal motion planning problems, respectively. But as we report in the introduction, robots must allocate limited resources between planning motions and subsequently executing them. Existing approaches for motion planning in such high-dimensional spaces, including sampling-based, asymptotically optimal and anytime planners, risk either quickly returning a solution that is expensive to execute, or spending too much planning effort improving an existing solution.

The primary contribution of this chapter is to imbue lazily-evaluated sampling-based approaches with a *utility function* which incorporates both planning and execution cost in its objective. Reasoning over utility provides the planner with a means to trade off between these costs using a single input parameter, as well as a natural termination condition. Furthermore, treating planning cost explicitly provides a natural way to accommodate task-specific planning heuristics. The planner performs well on a set of benchmark tasks, and is particularly amenable to caching and parallelization.

5.1 Motivation and Related Work

Autonomous robots in the real world must carefully allocate limited resources (e.g. time or energy) between planning motions and execut-



planning: 2.9s
execution: 5.7s

planning: 2.4s
execution: 12.4s

Figure 5.1: A robot reaching to grab a book must allocate limited resources between planning and executing its motion. Our planner reasons over a user-defined utility function (top left) to explicitly trade-off between these two significant sources of cost. As its roadmap search (right) discovers valid edges (/), it mediates between high-cost edges that are fast to find (/) and low-cost edges that require significant planning (/) directly via their prospective utilities. Because the planner accepts arbitrary estimators for planning cost, it can naturally exploit both caching and geometric structure in $\mathcal{C}$ (bottom left).

ing them. A robot that favors execution risks prematurely committing to an uneconomical path, whereas one that favors planning risks unduly delaying execution due to incessant small refinements. This tradeoff is especially relevant for working with or around humans, who are simultaneously unaccommodating of long pauses and of wild and unpredictable motion.

Most planning algorithms pick a side and stick to it.

Randomized algorithms such as the PRM [63] and RRT [74], strive to find feasible paths while minimizing planning cost, falling in the A_1 category of the planning vs. execution plot (Figure 5.1, we will formalize notation in Section 5.2). Techniques such as caching, parallelization [53], and lazy evaluation [10] are often used to further reduce online planning cost.

Optimal algorithms such as FMT* [57] exclusively optimize for execution cost (A_2 in Figure 5.1), and often use techniques such as informed heuristics to improve planning efficiency as a secondary objective. Graph search approaches [47] can guarantee solution optimality with respect to a chosen discretization, and often incorporate a parameter (A_3) such as inflation [95] or an execution cost bound [118] that reduces the algorithm's runtime at the expense of optimality. However, it can be difficult to set these parameters for a particular domain.

Anytime algorithms A_4 for continuous [61, 41, 48] or discrete [81] problems come the closest, providing a sequence of solutions with decreasing execution cost. However, they are best suited for *uncertain* planning budgets, and as a consequence of this uncertainty incur planning cost producing solutions that never get used. It can also be difficult to decide when to terminate them. We wish to do better.

Expressing the Planning vs. Execution Tradeoff. Our goal is to devise an algorithm that *explicitly* reasons over both planning and execution cost. Our algorithm should behave like an optimal algorithm if execution cost is critical, and like a randomized algorithm if planning cost is critical. Crucially, it should behave like a mix between the two if both are important to the user.

To enable this, we borrow a concept from the AI community [101], allowing the user to specify a *utility function* describing their planning vs. execution tradeoff (illustrated via isocontours in Figure 5.1). We describe some examples and theoretical properties of this function in Section 5.2, but it can be arbitrarily nuanced, or extremely simple, e.g. a linear function that sums planning and execution cost. We argue that utility is an essential metric by which to evaluate motion planners.

Considering this utility function, we make the following observations: (1) we can update estimates of utility online during planning; (2) we can estimate execution cost of a path (e.g. path length or energy) quite accurately.

However, our key insight is that we can *estimate the planning cost* in domains where this cost is dominated by the effort of verifying the validity of edges via collision-checking. This is true for most real-world robot motion planning problems where 80-95% of the planning time is dominated by collision-checking.

Lazily Evaluated Marginal Utility Roadmas. When combined, these insights enable a planning approach which reason explicitly over utility using online estimates. This leads naturally to the Lazily Evaluated Marginal Utility Roadmaps (LEMUR) planner (Section 5.3). LEMUR takes as input a parameterized utility function U_λ and attempts to find and validate a solution path that maximizes that utility.

LEMUR has several advantages over anytime planners. First, the user's utility function provides a natural termination condition – the planner returns a solution when no alternatives provide a prospective improvement in utility. Second, the planner does not waste effort generating intermediate solutions.

The performance of LEMUR is also easier to tune to a particular problem domain. In particular, the parameter it uses to mediate between costs is the user's utility function itself. It is therefore unnecessary to tune inflation factors or trajectory optimization budgets to achieve good performance.

We compare LEMUR to several sampling-based and anytime planners across a set of motion planning problems in Section 5.4.

LEMUR's ability to accept an arbitrary planning effort model enables it to be customized to different domains. In Chapter 6, we discuss such a model specific to problems over a family of C-space subsets. A prime example of this is multi-step manipulation tasks in which the C-space changes with each object grasp and placement. This allows computation to be efficiently cached and reused.

The combination of marginal utility, lazy evaluation, and roadmap methods work in concert and are adaptable to new domains. Section 5.5 concludes the chapter with a discussion of potential extensions and applications of our algorithm.

5.2 *Utility in Motion Planning*

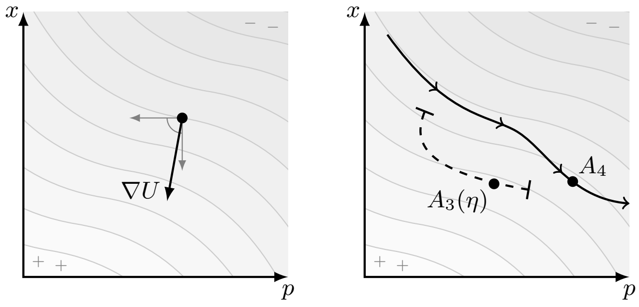
We consider the optimal motion planning problem as described in Chapter 2. We consider a cost function $x : \Xi \rightarrow \mathbb{R}^+$ which measures path quality – in our case, the cost of executing a solution path. Examples of x include the path's length or the time or energy required to execute it. x is commonly treated as part of the problem specification; each instance admits a path(s) with optimal execution cost x^*.

A sound motion planner A accepts a problem instance and yields a valid solution path ξ with execution cost $x[\xi]$. We are also interested in measuring the planning cost incurred by the algorithm itself via a real-valued performance metric p. Examples of p include the number of iterations or collision checks performed, or the amount of time or energy consumed by the planner. Figure 5.2 provides an illustration of these two costs on the plane.

In general, a planner endeavoring to reduce the execution cost of its solution path must incur additional planning cost to do so. While a wealth of planning approaches are available in the literature, few attempt to capture this underlying tradeoff directly. We propose to do so via a *utility function*. Utility functions were first exploited by the Best-first Utility-Guided Search (BUGSY) algorithm [101, 14] to order vertex expansions during a conventional graph search. We propose to apply a similar insight to motion planners which use lazy evaluation.

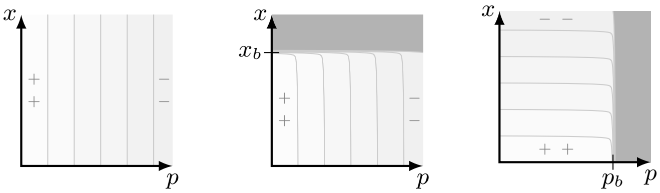
5.2.1 *Utility Functions*

A utility function U is a scalar-valued function over both p and x which provides the motion planner with the *utility* of returning a solution path with execution cost x calculated after incurring planning cost p (Figure 5.2). Utility functions are applicable to a wide range of planning regimes and often emerge readily from the problem domain. A utility function also naturally reconciles cost functions p and x that are in different units (e.g. collision checks and path length). We follow the convention that utility is to be maximized (whereas cost is to be minimized).



(a) A utility function over p, x

(b) Maximizing utility with parameterized or anytime planners



(c) First feasible

(d) Bounded cost

(e) Planning budget

Figure 5.2: Contours of a utility function $U(p, x)$. Also some examples of simple utility functions relevant to related work.

While U may be any function, there are certain properties that we can expect from any reasonable choice. In particular,

$$\nabla U \leq \mathbf{0}. \tag{5.1}$$

This follows from the following arguments (Figure 5.2(a)). If $\partial_x U$ were positive, it would be beneficial for a planner to return a path with larger execution cost. Similarly, if $\partial_p U$ were positive, it would be beneficial for a planner to artificially delay returning a given solution path.

Several common planning regimes can be expressed exactly as utility functions. For example, requesting a planner to return a solution as quickly as possible irrespective of its execution cost corresponds to the first-feasible utility in Figure 5.2(c). Bounded-cost planners such as Potential Search (PTS) [118] address problems in which a solution below a prescribed cost x_b is desired as quickly as possible (Figure 5.2(d)). The converse formulation requests the lowest-cost path within a fixed planning budget p_b (Figure 5.2(e)).

It is instructive to consider how existing types of planners might be applied if utility is to be maximized. Many planners (A_3 in Figure 5.2(b)) take parameters which profoundly affect both the quality of their solutions and the planning cost they incur. The range parameter of the RRT [74] and the inflation factor in Weighted A* [95] are

Algorithm 8 Lazily Evaluated Utility-Guided Search Outline

```
for iteration $i \in 1, 2, \ldots$ do
    $\bar{p}_i \leftarrow$ planning cost incurred so far
    $\Xi_i \leftarrow$ set of paths to consider at iteration $i$
    $\grave{p}_i : \Xi_i \rightarrow \mathbb{R}^+$    ▷ additional planning cost estimator
    $\hat{x}_i : \Xi_i \rightarrow \mathbb{R}^+$    ▷ execution cost estimator
    $\xi_i = \arg\max_{\xi \in \Xi_i} U(\ \bar{p}_i + \grave{p}_i(\xi),\ \hat{x}_i(\xi)\ )$
    return $\xi_i$ if $\grave{p}_i(\xi_i) = 0$
    evaluate $\xi_i$    ▷ incurs requisite planning cost
```

common examples of this. However, it is difficult to choose the values for these parameters; they must often be learned empirically in a way that is specific both to the planner and to the problem instance.

Anytime planners such as Anytime Repairing A* [81] and RRT* [60] return a low-quality solution quickly, and then continually return improved solutions as more planning is performed (A_4 in Figure 5.2(b)). Applying an anytime planner to a utility-maximization problem suffers from two drawbacks. First, an outside process must enforce an appropriate termination condition. While this may be straightforward for simple utilities (e.g. from Figures 5.2(c)-5.2(e)), a nontrivial utility function requires a complex termination model which must be learned in a problem-specific way. Second, scarce planning resources are typically allocated on low-quality intermediate paths which will go unused.

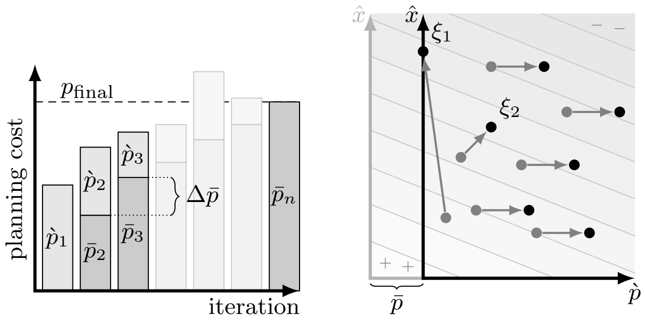
5.2.2 *Outline of Lazily Evaluated Utility-Guided Search*

We endeavor to marry utility functions with lazy motion planning. Lazy path evaluation is well-suited to domains with expensive validity checking, and is a common technique [10, 20]. Here, we provide the general outline of a class of algorithms which takes as input a utility function U and selects paths to evaluate based on estimates of their utility.

The planner maintains estimates $\hat{p}$ and $\hat{x}$ of the planning and execution costs, respectively, for each of a set of prospective trajectories Ξ. While a prospective path's execution cost may be easy for a planner to estimate via commonly used heuristics, incorporating estimates of the planning cost to be incurred by the algorithm itself is more difficult. While the planner is running, we can decompose its planning estimate $\hat{p}$ for a prospective path ξ into two components:

$$\hat{p}(\xi) = \bar{p} + \grave{p}[\xi] \tag{5.2}$$

that is, the measured planning cost elapsed $\bar{p}$ and the estimated remaining planning cost $\grave{p}$.



(a) Planning cost estimates

(b) Evolution of path utilities

Figure 5.3: A planner reasoning with a linear utility function chooses to first evaluate trajectory ξ_1. After incurring planning cost $\bar{p}$ (less than it had expected), it finds that it is more expensive than expected; some of the planning work has also adjusted the estimates for nearby trajectory ξ_2. However, the estimated remaining planning costs $\grave{p}$ for all other unaffected trajectories have remained constant. Therefore, the relative utilities have also not changed. As such, a planner need not re-order any trajectories whose estimated planning cost to go has not changed.

Consider the planner outline in Algorithm 8. At each iteration i, the planner has incurred $\bar{p}_i$ planning cost so far. It considers a set of prospective paths Ξ_i (left unspecified). It also has available estimators for each path's execution cost $\hat{x}_i$ and its remaining planning cost $\grave{p}_i$. Note that these estimators can change between iterations – for example, a path's remaining planning estimate becomes 0 once it is fully evaluated, and its execution effort becomes ∞ if it is found to be infeasible. Estimates for other paths which share segments may also be updated (Figure 5.3).

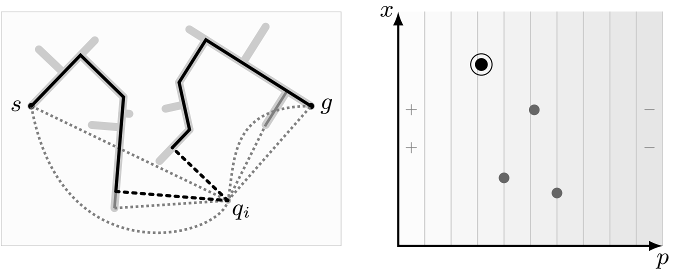
Using these estimators, the planner selects among Ξ_i the path ξ_i which maximizes the given utility function U. If no planning cost remains, it is returned; otherwise, it is evaluated, incurring the requisite planning cost. Note that this algorithm terminates naturally, and therefore avoids incurring planning cost seeking out low-quality intermediate solutions.

The outline in Algorithm 8 is able to broadly capture the behavior of a number of well-known planning algorithms. For example, consider the symmetric bidirectional variant of the common RRT-Connect algorithm [69]. At each iteration i, the algorithm samples a configuration q_i uniformly from $\mathcal{C}$. The candidate path that it then selects for evaluation (Figure 5.4) is identical to the path which minimizes the first-feasible utility function (Figure 5.2(c)) among Ξ_i, the set of all prospective paths constrained to pass through q_i. It is therefore unsurprising that RRT-Connect typically completes with remarkably little planning cost (albeit often with poor execution cost).

5.2.3 *Linear Combinations and Elapsed Planning Cost*

One particularly common class of utility functions consists of the a linear combinations of p and x:

$$U_l(p, x) = -w_p\, p - w_x\, x \tag{5.3}$$



(a) RRT snapshot.

(b) Utility

Figure 5.4: At each iteration i, the simplified bidirectional RRT-Connect [69] planner always selects among all paths constrained to pass through the sampled configuration q_i that which minimizes the necessary planning cost only.

for non-negative w_p, w_x. Despite their simplicity, such utilities are able to capture regimes which span the gamut between minimizing planning and execution costs. In particular, if w_p and w_x are chosen to bring their metrics into the same total task units with equal weight, U_l commands the planner to directly minimize the sum of the two. This is particularly appropriate in regimes where the robot is to immediately execute the planned path, and total task cost is to be minimized.

In addition to their applicability, a utility function that is a linear combination also exhibits a desirable cost-discounting property which a planner can exploit.

Theorem 13 *If the utility function can be written as $U(p, x) = -w_p\, p - f(x)$, then the path ξ_i selected by Algorithm 8 (line 6) is independent of the elapsed planning cost $\bar{p}$.*

Proof of Theorem 13 We have:

$$\begin{aligned} \xi_i &= \operatorname*{arg\,max}_{\xi \in \Xi_i} \Big[U\big(\, \bar{p}_i + \grave{p}_i[\xi],\ \hat{x}_i[\xi]\big) \Big] && (5.4) \\ &= \operatorname*{arg\,max}_{\xi \in \Xi_i} \Big[U\big(\, \grave{p}_i[\xi],\ \hat{x}_i[\xi]\big) - w_p\, \bar{p}_i \Big] && (5.5) \\ &= \operatorname*{arg\,max}_{\xi \in \Xi_i} \Big[U\big(\, \grave{p}_i[\xi],\ \hat{x}_i[\xi]\big) \Big] && (5.6) \end{aligned}$$

□

In other words, the utilities of all prospective paths Ξ_i are discounted the same amount (Figure 5.3(b)).

Because the planner maximizes U_l at each iteration, it is therefore completely characterized by a single parameter:

$$w_p = \lambda_U \quad w_x = 1 - \lambda_U \quad \lambda_U \in [0, 1] \qquad (5.7)$$

This parameter uniquely describes the relative tradeoff between minimizing planning and execution cost. It is often inherent in the problem setting, or easily elicited from users.

5.2.4 *Linear Combinations and the Parameter Selection Problem*

Consider applying a parameterized planner $A(\eta)$ to the problem of maximizing a convex combination utility U_l. Such a planner may take an explicit parameter, such as the inflation factor ϵ in Weighted A*, or it may be a termination condition for an anytime planner such as an optimization time budget. The performance of A for different values of η produces a locus on the p, x plane (Figure 5.5(a)). The utility achieved by any particular planner realization $A(\eta)$ corresponds to a line (Figure 5.5(b)):

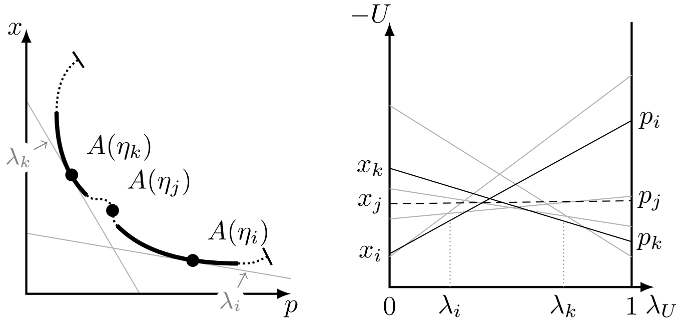
$$-U_A(\lambda_U, \eta) = \lambda_U \, p_A(\eta) + (1 - \lambda_U) \, x_A(\eta). \qquad (5.8)$$

Invoking such a planner for utility maximization requires that the parameter η be set appropriately as a function of λ_U. An ideal η schedule would follow the convex hull of all lines in Figure 5.5(b):

$$\eta^*_A(\lambda_U) = \arg\max_{\eta} U_A(\lambda_U, \eta). \qquad (5.9)$$

The locus of utility-maximizing planner parameters corresponds to the lower-left convex hull of A (bolded in Figure 5.5(a)).

Determining a suitable such schedule $\eta_A(\lambda_U)$, that is, the *parameter selection problem*, is difficult to solve for many planners across a wide range of problem domains. Tuning (e.g. setting a shortcutting time budget) is often required to achieve good utility in practice. One advantage of the utility-aware planner outlined in Algorithm 8 is that it operates directly on the problem's utility function itself.



(a) p-vs-x plot.

(b) Corresponding utility plotot.

Figure 5.5: A parameterized planner A maximizes a linear combination utility (for some value λ_U) along the subset of parameter values which constitute the lower-left convex hull of the p-vs-x plot (left). Each realization of planner A for a particular parameter value (a point at left) corresponds to a line at right, which shows the utility achieved for that realization over various values of λ_U (negative utility shown, lower is better). Applying a parameterized planner for utility maximization requires selecting η according to λ_U in such a way that the realized utility approximates the lower bound across all parameter choices (right).

5.3 *Marginal Utility on Roadmaps*

The Lazily Evaluated Marginal Utility Roadmaps (LEMUR) motion planner is an implementation of utility-guided search which unifies lazy evaluation, marginal utility, and roadmap methods. It follows the general outline from Algorithm 8 and exploits several properties and assumptions that are common in motion planning for articulated robots. The planner considers paths constrained to a roadmap graph G defined a priori in $\mathcal{C}$, and resembles repeated invocations of the LazySP algorithm from Chapter 3 on progressively densified roadmaps. The set of continuous paths Ξ considered by the planner is therefore restricted to the set of paths Π on the roadmap.

5.3.1 *Cost Estimates Additive over Edges*

Performing lazy evaluations entails repeated searches over roadmap paths. To make this efficient, we commit to path cost estimators $\check{p}_i(\pi)$ and $\hat{x}_i(\pi)$ that are additive over edges:

$$\check{p}_i(\pi) = \sum_{e \in \pi} \check{p}_i(e) \quad \text{and} \quad \hat{x}_i(\pi) = \sum_{e \in \pi} \hat{x}_i(e). \tag{5.10}$$

Many common performance metrics meet this criteria, including path length, execution time under velocity limits, and number of collision checks. These two edge estimators are provided as input to the planner in the form of an edge cost model $\mathcal{M}$. Utility maximization at each iteration i is then equivalent to solving a shortest-path problem over the roadmap with the following edge weight function:

$$w_i(e) = w_p\, \check{p}_i(e) + w_x\, \hat{x}_i(e). \tag{5.11}$$

5.3.2 *Locality of Estimator Updates*

The LEMUR algorithm evaluates a single roadmap edge e_i at each iteration i. Most common cost estimate models exhibit *update locality*: evaluating an edge does not affect the estimates of other edges on the roadmap. This allows for two optimizations. First, due to Theorem 13, the per-iteration edge weight function w_i can be stored as a single scalar value w per edge, with only a single edge weight updated per iteration. Second, an incremental shortest-path algorithm, as described in detail in Chapter 4, can be used to efficiently search for candidate paths.

Certain cost estimate models exhibit an approximate form of update locality. For example, the estimates for an edge $e_{ab} = (v_a, v_b)$ often incorporate the validity of the configurations at its end vertices v_a and v_b, which are shared with other adjacent roadmap edges. For example, if evaluating e_{ab} finds that v_a is invalid, then the estimates $\hat{x}$ and $\check{p}$ for all edges adjacent to v_a may also be updated. However, the number of edge weights to be updated remains small compared to the size of the roadmap.

Algorithm 9 Lazily Evaluated Marginal Utility Roadmaps

```
procedure LEMUR($q_{\mathrm{start}}, q_{\mathrm{goal}}, x, \mathcal{M}.\check{p}, \mathcal{M}.\hat{x}, \lambda_p$)
    $G \leftarrow$ graph with $V = \{q_{\mathrm{start}}, q_{\mathrm{goal}}\}$ and $E = \emptyset$
    $w : E \rightarrow \mathbb{R}^+$                         ▷ mutable edge weight function
    for iteration $i \in 1, 2, \ldots$ do
        $\pi_i = \arg\min_{\pi \in \Pi(G)} \mathrm{len}\,(\pi, w)$          ▷ incremental search
        if $\pi_i$ is null then
            $V_{\mathrm{new}}, E_{\mathrm{new}} \leftarrow$ new densified roadmap batch
            $G.V \overset{+}{\leftarrow} V_{\mathrm{new}}$;  $G.E \overset{+}{\leftarrow} E_{\mathrm{new}}$
            $w(e) \leftarrow \lambda_p\, \check{p}_i(e) + (1 - \lambda_p)\, \hat{x}_i(e)\ \forall\, e \in E_{\mathrm{new}}$
            continue
        if $\pi_i$ is fully evaluated then
            return $\pi_i$
        $e_i \leftarrow$ select unevaluated edge from $\pi_i$
        evaluate $e_i$
        $\hat{x}(e_i) \leftarrow x(e_i)$;  $\check{p}(e_i) \leftarrow 0$          ▷ e.g. evaluate fully
        $w(e_i) \leftarrow \lambda_p\, \check{p}(e_i) + (1 - \lambda_p)\, \hat{x}(e_i)$
```

5.3.3 *The Algorithm*

The LEMUR planner is shown in Algorithm 9. The planning query consists of the configurations $q_{\mathrm{start}}, q_{\mathrm{goal}}$ along with the evaluation function x which determines the true execution cost of an edge (e.g. ∞ if invalid). As x is usually expensive to compute, the planner is provided with an ensemble edge cost model $\mathcal{M}$ with two independent estimators: $\hat{x}(e)$ estimating the value of $x(e)$, and $\check{p}(e)$ estimating the planning cost required to compute $x(e)$ itself. The planner is also given the user's utility tradeoff $\lambda_p \in [0, 1]$ as a planner parameter.

The algorithm begins with an initial roadmap graph G comprised of the query vertices and an edge weight function w over the (initially empty) set of edges. At each iteration i, the roadmap is searched for a path π_i which maximizes estimated utility according to λ_p. This search uses an incremental search algorithm (our experiments use the IBiD algorithm from Chapter 4). If no such path exists (i.e. all paths have infinite weight), the roadmap is expanded with a new batch of vertices and edges. The algorithm is agnostic to the roadmap construction method used; our experiments use Halton sequences with an r-disk connection rule.

If the path π_i has been fully evaluated, then (a) it requires no additional planning cost, and (b) no prospective path on G has a larger estimated utility than π_i. Therefore, the algorithm immediately terminates and returns π_i.

Algorithm 10 Simple Edge Cost Model $\mathcal{M}_{\text{simple}}$

```
function $\check{p}_{\text{simple}}(e)$                ▷ remaining plan-cost estimator
    if $e$ is unevaluated then
        return $\sum_{q \in e} \check{p}_{\text{config}}(q)$
    else
        return $0$
function $\hat{x}_{\text{simple}}(e)$                  ▷ exec-cost estimator
    if $e$ is unevaluated or valid then
        return $||e||$
    else
        return $\infty$
```

Otherwise, an edge e_i on π_i that has not been fully evaluated is selected for evaluation. Our experiments simply select the unevaluated edge nearest to an end of the path, although other selection strategies are possible. Once evaluated, the estimates for e_i are recomputed, modifying its edge weight according to (5.11) for future iterations. For example, if it is fully evaluated, its remaining planning cost becomes zero.

5.3.4 *Ensemble Edge Cost Models*

The ensemble edge cost model $\mathcal{M}$ which provides the estimators $\check{p}(e)$ and $\hat{x}(e)$ depends on the implementation of the evaluation function $x(e)$ and is therefore specific to the the planning domain. Suppose that the true execution cost x of each edge is either its edge length if valid, or ∞ otherwise. Further, suppose that determining its validity requires performing discrete collision checks interpolated along the edge at some resolution. The cost model $\mathcal{M}_{\text{simple}}$ shown in Algorithm 10 captures this case. Note that this model uses the free-space assumption: unevaluated edges are assumed to be valid.

5.3.5 *Analysis*

LEMUR is an application of marginal utility to lazy roadmap methods, and therefore it shares a similar structure to Lazy PRM [10]. In fact, in the case that $\lambda_p = 0$, LEMUR reduces to Lazy PRM (for appropriate choice of the edge selection and evaluation strategy), since both algorithms will only consider execution cost when choosing candidate paths.

LEMUR conducts its searches over any progression of successively densified roadmaps over $\mathcal{C}$. Recent work has demonstrated that roadmaps constructed via randomized [61] or deterministic [56] sampling techniques (for appropriate choice of the roadmap parameters) endow motion planners which conduct systematic searches thereon with desirable properties such as resolution and probabilistic completeness. If the estimates $\hat{p}$ and $\hat{x}$ are both finite for unevaluated edges, LEMUR is guaranteed to conduct such a systematic search.

The cost model's estimators $\hat{p}$ and $\hat{x}$ can each be any function. However, it is often the case that bounds are available for both estimates relative to the true execution cost of the edge x. For example, both may be proportional to the Euclidean length of the edge (e.g. the estimated planning cost may be proportional to the number of interpolated configurations to be checked). Such bounds allow for a suboptimality bound on the execution cost of the path returned by LEMUR (Theorem 14).

Theorem 14 (Execution Suboptimality of LEMUR) *If the cost model $\mathcal{M}$ satisfies*

$$\hat{x}(e) \leq \alpha_x x(e) \text{ and } \hat{p}(e) \leq \alpha_p x(e) \tag{5.12}$$

for some constants α_x and α_p, then the execution cost of the path returned by LEMUR with $\lambda_p < 1$ is within a factor ϵ of the execution-optimal path on G, with $\epsilon = \frac{\lambda_p}{1-\lambda_p}\alpha_p + \alpha_x$.

Proof of Theorem 14 Suppose that LEMUR returns path π_L. Since it is fully evaluated, it has weight $w(\pi_L) = (1-\lambda_p)x(\pi_L)$. Now, consider an execution-optimal path π^* on G, which had $w(\pi^*) = \lambda_p \hat{p}(\pi^*) + (1-\lambda_p)\hat{x}(\pi^*)$ at the point the algorithm terminated. Since π_L was chosen over π^*, we have that $w(\pi_L) \leq w(\pi^*)$. Due to (5.12), it follows that

$$(1-\lambda_p)x(\pi_L) \leq \lambda_p \alpha_p x(\pi^*) + (1-\lambda_p)\alpha_x x(\pi^*) \tag{5.13}$$

Because we have $\lambda_p < 1$, we then have:

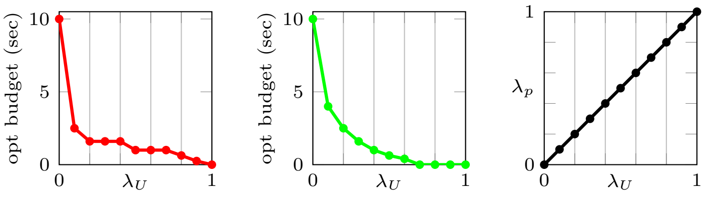
$$x(\pi_L) \leq \left(\frac{\lambda_p}{1-\lambda_p}\alpha_p + \alpha_x \right) x(\pi^*). \tag{5.14}$$

□

5.3.6 Suitability for Caching

LEMUR searches a sequence of progressively densified roadmaps in $\mathcal{C}$. Analysis of the performance of lazy search for asymptotically-optimal regimes [48] has identified the nearest-neighbor queries required to construct the roadmap itself as a significant component of planning cost. We exploit two factors to mitigate this. First, because LEMUR endeavors to maximize utility (and therefore terminates), its asymptotic behavior is of less importance. Second, we commit to roadmaps that are (a) deterministic and (b) independent of the distribution of obstacles (e.g. r-disk or k-nearest roadmaps). This allows us the option to to amortize the cost of constructing each batch of the roadmap across all planning queries performed by the robot. In our experiments, we present timing results with and without the roadmap cache (the same cache is used across all problem instances).

Beyond the roadmap structure itself, edge validity state can be persisted across planning queries yielding a multi-query planner that is similar to the Lazy PRM or Experience graphs [91]. We briefly discuss how to accommodate a changing configuration space in Chapter 6.



(a) RRT (Hybrid)

(b) BIT*

(c) LEMUR

Figure 5.6: Schedule of parameters for three of the algorithms compared. The hybrid RRT-Connect and BIT* are both anytime planners. The parameter learned was the algorithm termination time after the first returned path. The LEMUR algorithm does not require tuning; we used $\lambda_p = \lambda_U$ in our experiments.

5.4 Experiments

We conducted experiments for a robotic platform armed with a 7 DOF Barrett WAM manipulator [102]. We used the FCL collision checker [90] with a resolution of 0.01 rad. We considered three single-query instances in a tabletop manipulation scenario and three single-query instances from a bookshelf scenario (Figure 5.7). Planners were evaluated against a range of utility functions trading off between path length (rad) and planning time (sec), with $\lambda_U = 0.5$ corresponding to an equal weighting when executing at 1.0 rad/sec.

We implemented LEMUR as a planner for the Open Motion Planning Library (OMPL) [119]. We used the 7D Halton sequence to generate a low-dispersion point set adjusted by an offset drawn uniformly from $\mathcal{C}$. Each roadmap batch consists of 10,000 vertices, and the r-disk connection radius decreased from 2.0 rad at the first batch in proportion to the Halton dispersion bound. Runtime memory usage was reduced by interpolating and allocating within-edge states lazily during the search. The planning parameter was chosen as $\lambda_p = \lambda_U$.

We compared LEMUR against RRT-Connect [69] and Batch Informed Trees (BIT*) [41] as implemented in OMPL version 1.1.0 with default parameters. To address maximal-utility problems, RRT-Connect was augmented with the default OMPL path shortcutter. For each planner, the optimization budget before termination as a function of λ_U was learned to maximize average utility across the

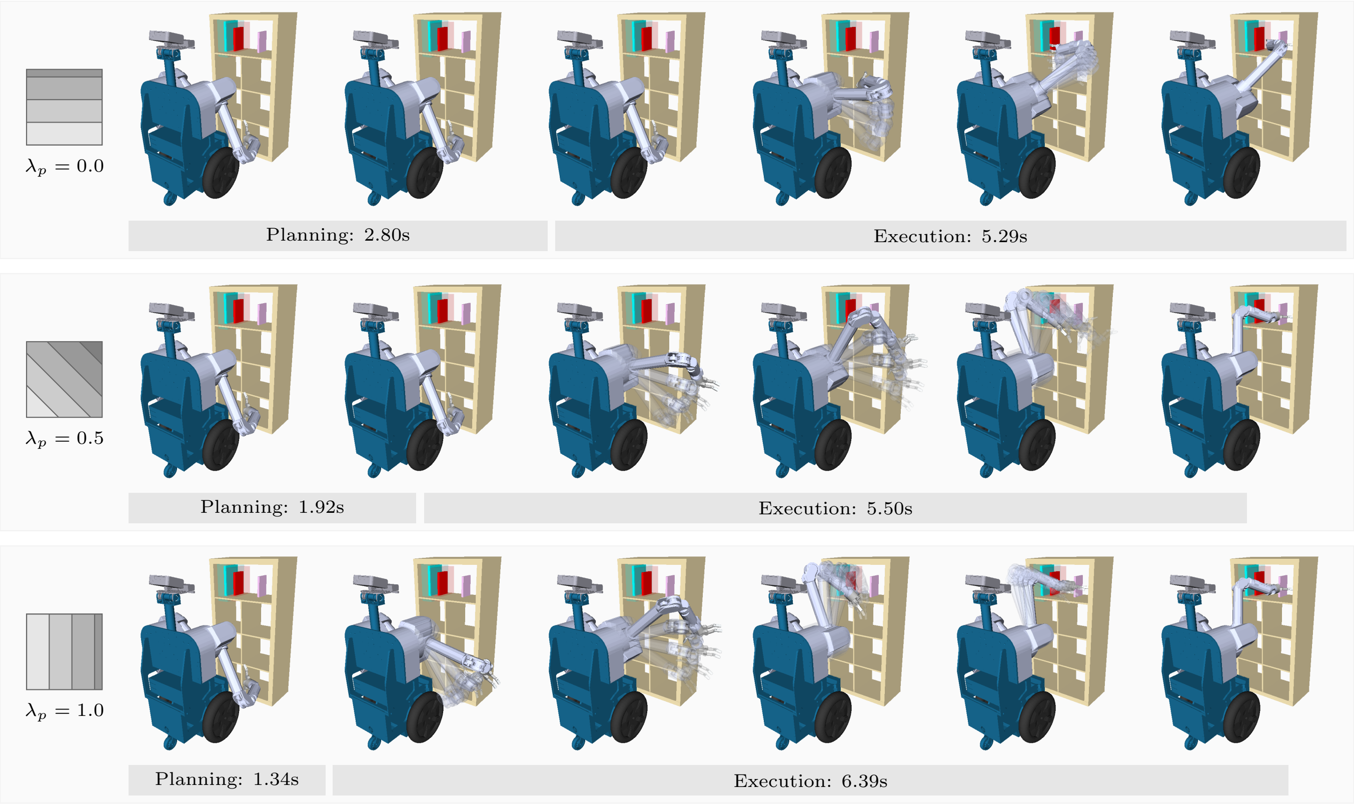

$\lambda_p = 0.0$
Planning: 2.80s
Execution: 5.29s
$\lambda_p = 0.5$
Planning: 1.92s
Execution: 5.50s
$\lambda_p = 1.0$
Planning: 1.34s
Execution: 6.39s

Figure 5.7: Illustration of three trajectories generated by LEMUR on instance 6 from our experiments. The planner was initialized with the parameters $\lambda_p = 0, 0.5$, and 1; the same roadmap was used. By increasing λ_p, the planner prefers minimizing planning cost at the expense of more costly solution paths.

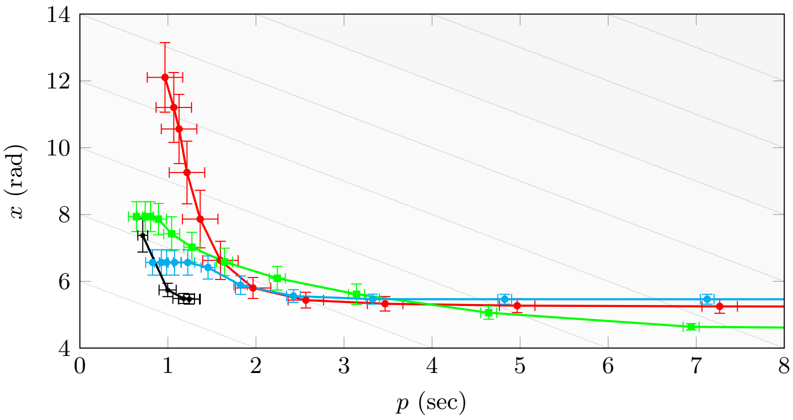


Figure 5.8: Comparison of measured planning time p and solution execution cost x for the first step of the HERB table-clearing task. Results for four parameterized planners are shown: RRT-Connect (🟥), BIT* (🟩), Lazy ARA* (🟦), and LEMUR (⬛). The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 (lower right) to 0.99 (upper left). The results for other planners show the effect of changing the total optimization time after the first returned solution. Also shown for reference are the contours for the $\lambda_U = 0.50$ utility function (i.e. 1 rad = 1 sec).

instances (Figure 5.6).

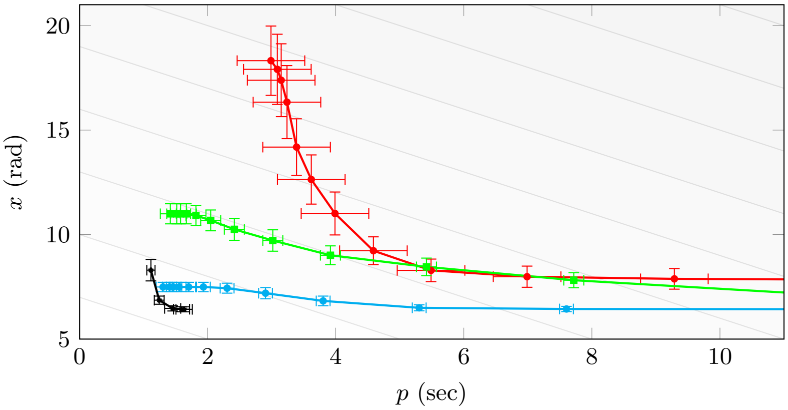


Figure 5.9: Comparison of measured planning time p and solution execution cost x for the FG step of the workcell task. Results for four parameterized planners are shown: RRT-Connect (🟥), BIT* (🟩), Lazy ARA* (🟦), and LEMUR (⬛). The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 (lower right) to 0.99 (upper left). The results for other planners show the effect of changing the total optimization time after the first returned solution. Also shown for reference are the contours for the $\lambda_U = 0.50$ utility function (i.e. 1 rad = 1 sec).

We ran 50 trials for each planner, with different random seeds. The results are shown in Figure D.1. We collected results for versions of LEMUR with and without nearest neighbor caching, with time breakdowns for each shown in Figure D.2.

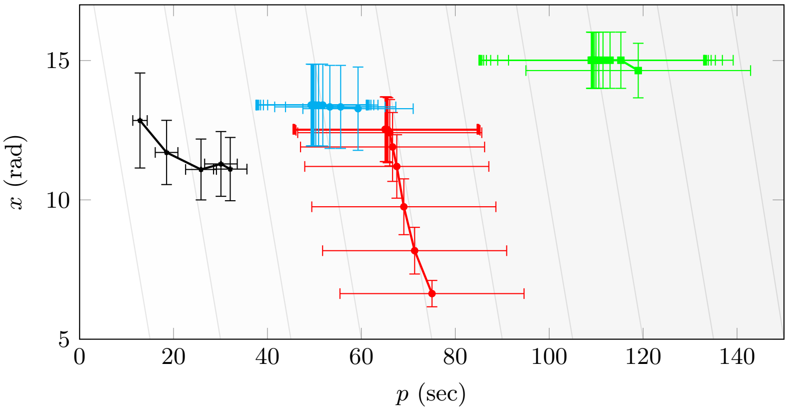


Figure 5.10: Comparison of measured planning time p and solution execution cost x for the first step of a CHIMP valve turning task. Results for four parameterized planners are shown: RRT-Connect (🟥), BIT* (🟩), Lazy ARA* (🟦), and LEMUR (⬛). The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 (lower right) to 0.99 (upper left). The results for other planners show the effect of changing the total optimization time after the first returned solution. Also shown for reference are the contours for the $\lambda_U = 0.50$ utility function (i.e. 1 rad = 1 sec).

The relative performance between the planners varies significantly between the six instances considered. For example, BIT* performs very well on some instances (e.g. nos. 2 and 6), whereas it takes more than 100 seconds on average to discover a path for instance 5. The latter three (bookshelf) instances appear to pose particular difficulty for the RRT; the constrained space and cubbies may result in its trees getting stuck easily.

As expected, loading its roadmap from a cache significantly speeds up LEMUR (by 4s - 15s on these instances). As shown in Figure D.2, a large majority of the cached algorithm's runtime is spent collision checking. The path produced by LEMUR is independent of this caching, so their utility curves meet at $\lambda_U = 0$ (when only solution quality is considered). This condition also corresponds to the behavior of the Lazy PRM.

Across the six motion planning instances, we found that LEMUR placed consistently among the best performers across the range of utility functions.

5.5 *Discussion*

As robots take on more complex and deliberative tasks, it is essential that they balance their computational and physical efficiency. Utility functions provide a natural way to represent this inherent tradeoff between planning and execution in motion planning. The LEMUR planner relies directly on this utility to guide its search for such a balance.

LEMUR relies on a domain-specific ensemble cost model to estimate the planning and execution costs of prospective motions. The parameter(s) of these models, such as the expected cost of a collision check, are easier to measure than corresponding parameters needed by other planners.

While LEMUR is agnostic to the class of roadmap used, methods which use deterministic sampling [73, 56]. have distinct advantages with regard to caching of collision state within and between planning episodes and parallel processors. Furthermore, by committing to a particular set of roadmap instantiations, we can potentially optimize them offline [103] to exhibit good behavior on a particular robot.

6

Planning over Configuration Space Families

To this point, we have considered motion planning problems in which a path is sought within a fixed valid subset $\mathcal{C}_{\text{free}} \subset \mathcal{C}$ In Chapter 3, we described a lazy search algorithm which is suited to domains in which determining the weight of an edge – perhaps due to expensive set membership tests – is expensive. Chapter 5 discussed the concept of utility to motion planning, and showed examples of its application to single-query motion planning problems.

However, many applications exhibit structure beyond simple binary belief over configuration validity. In this chapter, we introduce the *family motion planning problem*, a generalization of the motion planning problem to a family of sets over $\mathcal{C}$. We then show how such a problem over families can be represented as in the utility function framework, and given as input to the LEMUR motion planner described in Chapter 5.

This chapter proceeds as follows. Section 6.1 surveys related work. In Section 6.2, we motivate these the family motion planning problem from the standpoint of manipulation tasks. Section 6.3 formulates the problem generally. We describe our approach in Section 6.4, which details how the family problem can be represented as a utility model that can be used by LEMUR. The chapter concludes with experimental results on multi-step manipulation tasks in Section 6.5.

6.1 Related Work

The topic of reusing planning computation between similar motion planning problems has been extensively studied in the literature. We include here a broad survey of existing approaches.

Exact Algorithms. Exact planning methods construct explicit obstacles directly in the configuration space. Many such approaches allow for pre-computation of primitives, such as bitmaps [62] or C-space primitives for different workspace obstacles [88]. More recently, Lien and Lu [78] describe a method to build a PRM around obstacles in a database, and then reposes them in a new world. As described in Section 2.3.1, the exact approach is not easily applicable to articulated robots with complex mappings from workspace to C-space.

Accommodating Dynamic Subsets in Sampling-Based Planning. Strong recent interest in sampling-based planning has lead to the development of a number of approaches to handle environments in which $\mathcal{C}_{\text{free}}$ changes over time. If a given discretization is asserted a priori, this problem setting is similar to the dynamic shortest path problem from Chapter 4; we refer the reader to that chapter for a review of related work. Many sampling-based approaches attempt to prune and grow the discretizations itself in response to these changes, such as the Dynamic RRT [35], the Reconfigurable Random Forest (RRF) [77], and the Lazy Reconfiguration Forest [43]. However, these approaches do not reason explicitly about the structure of the configuration space, which we will consider in Section 6.2.

Considering Static and Dynamic Obstacles Separately. Some approaches do take advantage of such structure through a two-level dichotomy between the *permanent* and *non-permanent* configuration space obstacles that induce $\mathcal{C}_{free}$. Leven and Hutchinson [75, 76] and similar work [59] handle changing environments by pre-computing a self-collision-free roadmap offline, and then pruning it at query time using a mapping from workspace cells to roadmap edges. Other methods [55] exploit the dichotomy between static and dynamic parts of the world online. The family motion planning problem is a generalization of these formulations to more than two C-space subsets.

Task and Motion Planning. The structure in manipulation tasks that our approach leverages is similar to the *conditional reachability graph* which is part of the recent FFRob heuristic task planning framework [42]. While this framework does make use of a similar configuration space decomposition for manipulation tasks as described in Section 6.2, it differs from our approach in two ways. First, it is concerned only with manipulation tasks, and does not consider how the induced set of motion planning problems can be formulated more generally (e.g. in Section 6.3). Second, because the framework does not consider utility, its motion planner is not able to exploit the configuration space structure to the same extent as our approach.

6.2 *Motivation: Families in Manipulation Tasks*

Motion planning approaches that build graphs in the collision-free subset of configuration space as described in Chapter 2, e.g. the PRM [63] and RRT [74], have proven promising for high-dimensional articulated robotics problems in unstructured environments. These approaches devote a large amount of computational effort testing configurations and paths for collision via the free set indicator function (Section 2.2.1), and multi-query planners can then reuse the resulting graph for other queries in the same collision-free subset.

However, for manipulation problems, this subset of the robot's configuration space is sensitive to the locations and shapes of both people and objects in the environment, as well as the robot itself. It also depends on the shape and pose of any object grasped by the robot. Consider the table clearing task in Figure 6.1. The robot's valid subset $\mathcal{C}_{\text{free}}$ changes each time the object is grasped or released. Over the course of a planning episode, these changing subsets constitute a family of related sets over $\mathcal{C}$.

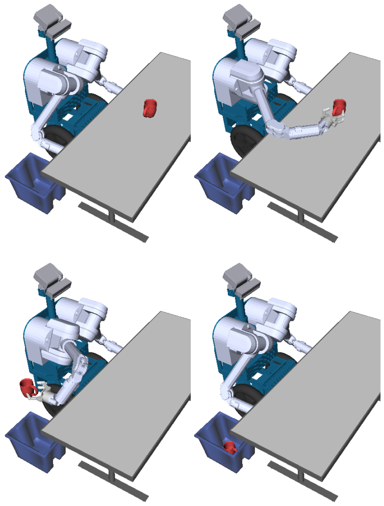


Figure 6.1: A simple manipulation task: retrieve the mug from the table, and drop it in the blue bin. This task requires plans in three distinct C-space free subsets.

This makes it difficult not only to apply the results of prior planning computation to the current problem, but also to efficiently consider multiple planned or hypothesized motions, since we must reconstruct our graph from scratch whenever the environment changes. This is especially the case for multi-step manipulation tasks that must be planned into the future.

We use this multi-step manipulation task as a motivating example for the family motion planning problem, and dive more deeply next into the structure of the problem's composite configuration space. We will consider problems over other robots and applications later in this chapter.

6.2.1 *The Composite Configuration Space*

The configuration of a quasistatic manipulation environment with multiple movable objects can be represented as a *composite configuration space*, consisting of the Cartesian product of the individual configuration spaces of the constituent objects. For example, consider an environment with a robot R and an object O; each is endowed with a configuration space:

The composite configuration space is also called the *joint configuration space*.

$$\mathcal{C}_R \quad \text{and} \quad \mathcal{C}_O. \tag{6.1}$$

For example, the robot's configuration space may be represented as its joint angles, while the object's configuration may be $SE(3)$. The composite space is then defined as

$$\mathcal{C}_{RO} = \mathcal{C}_R \times \mathcal{C}_O. \tag{6.2}$$

Of course, not all composite configurations will be feasible; some may correspond to configurations in which the robot, object, or static environment are intersecting each other (colliding), while others may denote configurations of the object where it is not at a stable placement or grasp configuration.

Visualizing the Composite Configuration Space. While the composite configuration space for any interesting manipulation task is of too high dimension to effectively visualize in full, we can make an approximation shown in Figure 6.2. Imagine that this 3D visualization represents a projection of $\mathcal{C}_{RO}$ so that the two horizontal axes x and y correspond to the robot's configuration space $\mathcal{C}_R$, while the vertical axis z corresponds to the object's configuration space $\mathcal{C}_O$. For the sake of this visualization, we ignore constraints on feasible object placements.

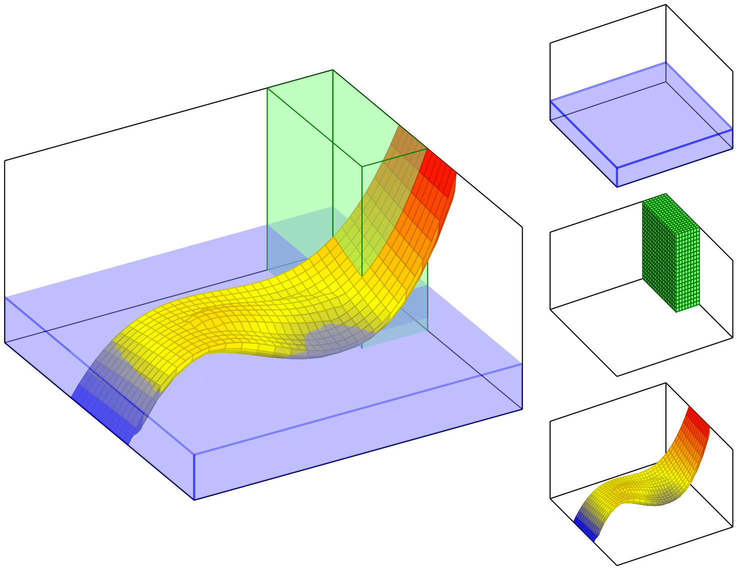


Figure 6.2: Illustration of a composite configuration space for a manipulation task.

Within the composite configuration space are three volumes corresponding to composite configuration space obstacles, shown individually at the right of Figure 6.2:

- First, in blue, is a volume which representing configurations in which the movable object collides with the static environment. Note that the obstacle is invariant to the configuration of the robot.

- Second, in green, is a volume in which the robot collides with the static environment. Similarly, note that this obstacle is invariant to the configuration of the object.

- Third, colored by height, is a volume representing composite configurations in which the robot and object are colliding.

The manipulation problem can then be formulated as a motion planning in this composite configuration space, taking the system from a starting configuration (e.g. with the robot in its home configuration, and the mug on the table from Figure 6.1) to a destination configuration(s) (e.g. with the robot returned home, and the mug in the bin). However, this composite configuration space is also encumbered by constraints which restrict allowable motion to constraint manifolds.

6.2.2 *Transit and Transfer Manifolds*

The source of these constraint manifolds within the composite configuration space is the fact that in prehensile manipulation tasks, the movable object cannot move on its own. In fact, any solution task alternates between two types of constraints: *transit* manifolds, in which the robot's configuration changes while that of the object remains constant, and *transfer* manifolds, in which the configuration of the object moves as a function of the robot's configuration (i.e. during a grasp). (This dichotomy can be extended to multiple robots or movable objects.) For an in-depth treatment of the structure of these manifolds in manipulation tasks, we refer the reader to [109].

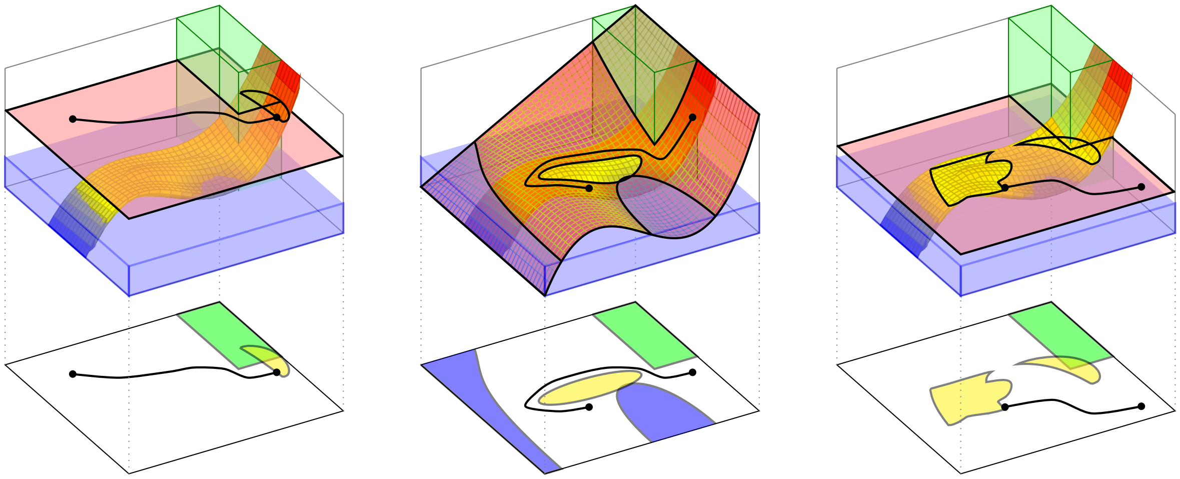


Figure 6.3: Illustration of a transit and transfer constraint manifolds in the composite configuration space for a manipulation task, along with projections of each onto the robot's configuration space.

Transit and Transfer Manifolds in $\mathcal{C}_{RO}$. To address a manipulation task, then, the robot must move through this composite configuration space $\mathcal{C}_{RO}$ while abiding by the underlying transit and transfer constraint manifolds. Consider the visualization in Figure 6.3, with manifolds shown as red surfaces. In the first step, the robot must transit from its current configuration at left to a grasp configuration at right, while constrained to the transit manifold shown in red. Next, the second step is constrained to a transfer manifold in which both the robot and the object move together to an placement location. Finally, the third step shows an addition transit away from the placement location to a desired destination configuration.

Projecting Manifolds onto $\mathcal{C}_R$. Since the full composite configuration while constrained to a manifold can be expressed as a function of the robot configuration q_R only, it is sufficient to consider eac subproblem as the projection of the composite obstacles in $\mathcal{C}_{RO}$ onto the robot's configuration space $\mathcal{C}_R$. Below each depiction of the manifolds in Figure 6.3 lies a visualization of this projection, along with the projected solution path.

The first and third subfigures correspond to transit subproblems, in which motion through the composite space is constrained so that the movable object's configuration remains constant. The second subfigure corresponds to a transfer subproblem, in which the motion of the object directly follows from the motion of the robot.

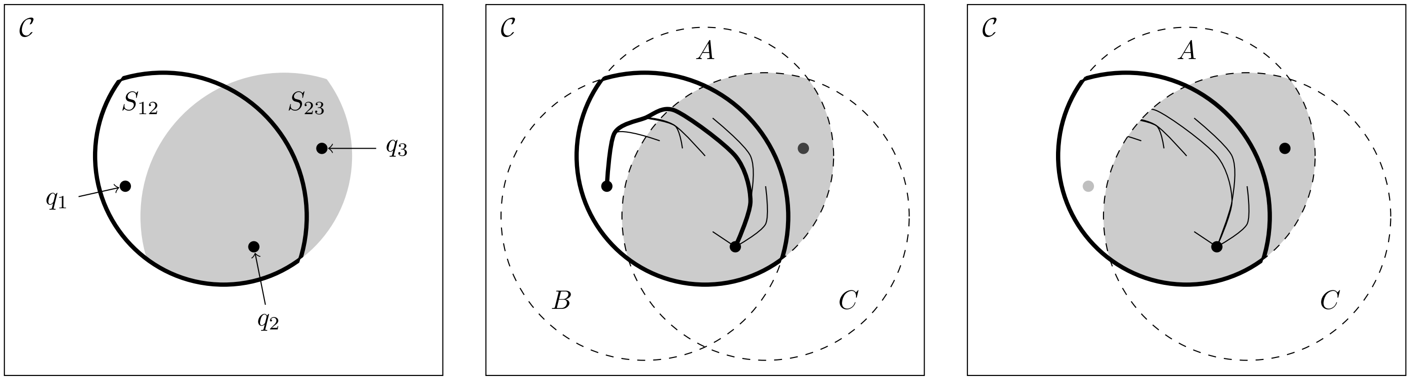
6.2.3 *Planning over a Family of Related Subsets*

The depictions of the three projections in Figure 6.3 lends a concrete picture to the description of changing free subsets depicted in Figure 6.1. Clearly, when the composite configuration space $\mathcal{C}_{RO}$ is projected onto $\mathcal{C}_R$, each of the three motion planning problems required by the task induces a different free subset $\mathcal{C}_{\text{free}} \subset \mathcal{C}_R$.

Importantly however, the free subsets are related. In the visualized example, the projection of the green composite obstacle is identical across the three free subsets.

How can we take advantage of this? Consider a roadmap method addressing the table clearing problem in Figure 6.1. Suppose that a number of edges have already been evaluated in order to find a valid path for the first step to grasp the red mug. Once grasped, the active valid subset $\mathcal{C}_{\text{free}}$ within which the second step must be planned has changed. However, any edge known to be valid for the previous step can be validated in the new subset by simply checking the grasped mug against the robot environment. A similar example in a 2D world is presented in Figure 6.4.

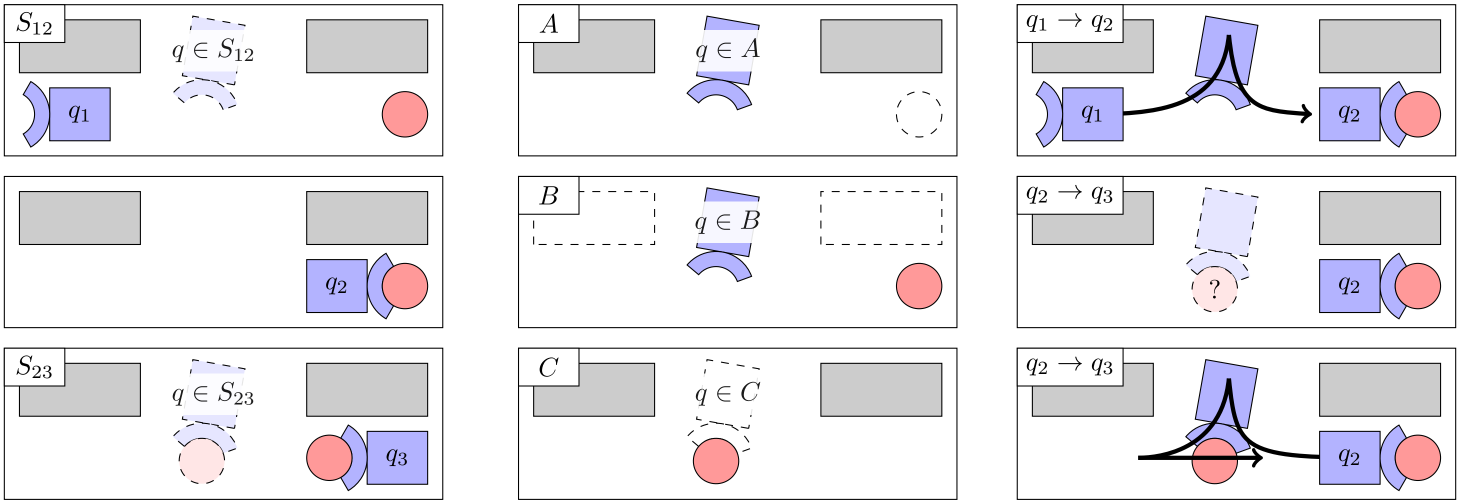
This example from a simple manipulation task motivates the formalization of the family motion planning problem in Section 6.3, which is more generally applicable than manipulation tasks in particular. We then develop our intuition from the simple examples in this section when describing our approach in greater detail in Section 6.4.



(a) A two-part family problem in $\mathcal{C}$, first between q_1 and q_2 through S_{12}, then between q_2 and q_3 through S_{23}. The two free subsets S_{12} and S_{23} are distinct but related.

(b) The free subsets are related via other underlying subsets of $\mathcal{C}$, with $S_{12} = A \cap B$ and $S_{23} = A \cap C$. A planner solving the first part (from q_1 to q_2) has found paths in S_{12}.

(c) Due to the set relations, a planner solving the second part (from q_2 to q_3 in S_{23}) can reuse any segment known to be in S_{12} by checking only for its membership in C.



(d) A forklift in a parking lot (q_1) must retrieve an object (q_2) and reverse park (q_3). This two-part problem requires plans in distinct collision-free $\mathcal{C}$-subsets S_{12} and S_{23}.

(e) Sets S_{12} and S_{23} are subsets of the configuration space of the robot $\mathcal{C} = \text{SE}(2)$, and can be represented as intersections of underlying subsets A, B, and C as in (b).

(f) After planning a path from q_1 to q_2 (top), a planner can reuse a configuration in S_{12} (middle) by checking only for its membership in subset C, resulting in plan reuse (bottom).

Figure 6.4: An illustration of a family motion planning problem in a common configuration space $\mathcal{C}$. The problem definition generalizes to an arbitrary number of configuration space subsets and set relations between them. When two queries in different subsets are solved sequentially, a family motion planner can reuse path segments less expensively. See Section 6.5 for examples in manipulation.

6.3 The Family Motion Planning Problem

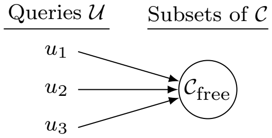
The family motion planning problem is a generalization of both the movers' problem (Section 2) and the *multi-query* planning problem [63]. The problem is multi-query in a fixed configuration space $\mathcal{C}$, in that it accommodates multiple distinct motion planning queries (e.g. between $q_{\text{start}}, q_{\text{dest}} \in \mathcal{C}$). However, unlike existing multi-query formulations in which all queries demand solution paths contained within a single common subset of $\mathcal{C}$ (i.e. the set of collision-free configurations, denoted $\mathcal{C}_{\text{free}}$), the family problem allows for the specification of a *family* of multiple such subsets $\mathcal{F} = \{A, B, \dots\}$. Like $\mathcal{C}_{\text{free}}$, each member of $\mathcal{F}$ is a subset of the common configuration space (that is, $S \subseteq \mathcal{C} \; \forall \; S \in \mathcal{F}$), and each subset S has its own indicator and planning estimator $\mathbf{1}_S[\cdot]$ and $\hat{p}_S(\cdot)$ as in Section 2.2.1. For example, in Fig. 6.4(e), $\mathcal{C}$-subset B consists of configurations free of collision between the robot and the initial object pose.

The problem supports an arbitrary number of queries $\mathcal{U}$. Each query u demands a solution path through a *single* $\mathcal{C}$-subset $U \in \mathcal{F}$ (see Fig 6.5):

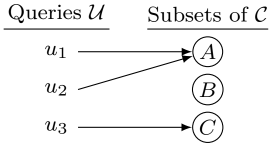
$$u : (q_{start},\ q_{goal},\ U). \tag{6.3}$$

Finally, the family problem includes a list of set relations $\mathcal{R}$ between the $\mathcal{C}$-subsets in $\mathcal{F}$. These can be expressed directly using set theoretic relations, or equivalently as logical statements on the corresponding indicator functions. Common types of such relations (containment and intersection) are illustrated in Fig. 6.6. Fig. 6.4 gives an example of intersection relations; an example of containment is a padded (conservative) robot model.

Together, these four elements (a configuration space $\mathcal{C}$, subsets $\mathcal{F}$ each with endowed indicators, a set of queries $\mathcal{U}$, and a list of subset relations $\mathcal{R}$) comprise a family motion planning problem.

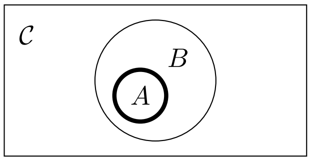


(a) Multi-query planning



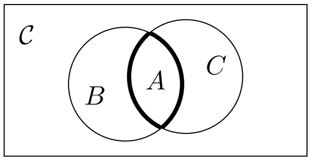
(b) Family motion planning

Figure 6.5: While queries in multi-query planning reference the same subset of $\mathcal{C}$, each family query references one of a number of such sets.



$A \subseteq B$

(a) Containment relation



$A = B \cap C$

(b) Intersection relation

Figure 6.6: Types of subset relations. Each relation can be expressed directly as set relations w.r.t a set S, or equivalently as logical statements on the corresponding indicator functions $\mathbf{1}_S(\cdot)$.

Revisiting the Example (as a Family Motion Planning Problem). Consider the diagram from Fig. 6.4. $\mathcal{F}$ consists of five $\mathcal{C}$-subsets labeled A, B, C, S_{12}, and S_{23}, and we have two queries, $u_{12} : (q_1, q_2, S_{12})$ and $u_{23} : (q_2, q_3, S_{23})$. $\mathcal{R}$ consists of the two relations $S_{12} = A \cap B$ and $S_{23} = A \cap C$. Suppose a cost model $\mathcal{M}$ wherein evaluating the indicator $\mathbf{1}_A$ incurs cost 4, evaluating $\mathbf{1}_B$ and $\mathbf{1}_C$ incurs cost 2, and evaluating $\mathbf{1}_{S_{12}}$ and $\mathbf{1}_{S_{23}}$ incurs cost 6. In the manipulation example in Fig. 6.4(d), this would be the case if each pairwise outlined shape collision check incurs unit cost.

Suppose a graph structure within S_{12} has been grown to solve the first query u_{12}. During the subsequent solve of query u_{23}, an existing path segment known to be in S_{12} can be shown to also be contained within S_{23} by only evaluating $\mathbf{1}_C$. In the manipulation example, reusing an a configuration from the previous search would require only a check of cost 2, instead of cost 6 for a new configuration. Thus, we might hope that a planner may be biased towards reusing said path segments in this case.

6.4 *Approach: A Utility Model over Family Beliefs*

Recall from Chapter 5 that the LEMUR motion planner takes as input a domain-specific ensemble cost model $\mathcal{M}$ to provide it with planning and execution cost estimates for prospective edges. This allows it to exploit domain-specific knowledge and structure that may be present.

See Section 5.3.4 for the definition of an ensemble edge cost model.

We wish to capture the structure of the family motion planning problem via such an ensemble edge cost model $\mathcal{M}_{\text{family}}$. To do so requires us to implement the remaining planning cost estimator $\grave{p}_{\text{family}}(e)$ as a function of the underly family $\mathcal{F}$ of $\mathcal{C}$-subsets and the intersection/inclusion relations between them. We accomplish this over the course of a planning episode by reasoning explicitly about belief over the subset membership of individual configurations and edges.

6.4.1 *Family Beliefs*

Consider a family $\mathcal{F}$ consisting of n subsets, and consider a configuration $q \in \mathcal{C}$ on which we may have invoked one or multiple subset indicator functions. How can we generally represent our knowledge about the subset membership of q? We can form a *belief space* $\mathcal{B}$ as follows

$$\mathcal{B} = \prod_{S \in \mathcal{F}} \{\text{Unknown}, \text{False}, \text{True}\}. \tag{6.4}$$

Recall that for subset S, invoking the indicator function $\mathbf{1}_S$ on configuration q returns a binary value representing whether $q \in S$.

That is, a belief state $b \in \mathcal{B}$ stores knowledge of subset membership for each subset $S \in \mathcal{F}$. (We will abbreviate the membership beliefs for each state in (6.4) as U, F, and T respectively.)

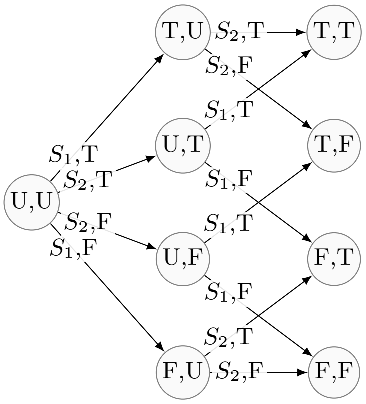


Figure 6.7: Example family belief graph for a family of two subsets, S_1 and S_2. From the initial belief $b_{\text{init}} = (\text{U}, \text{U})$, each transition (S, r) represents invoking indicator $\mathbf{1}_S$ with returned binary result r.

Composing a Family Belief Graph. Consider the initial belief state of a configuration $b_{\text{init}} = (\text{U}, \text{U}, \dots)$ for all subsets. How does a belief state b transition after invoking a particular indicator function? In the most general case, invoking the i-th indicator $\mathbf{1}_{S_i}$ returning $r \in \{\text{F}, \text{T}\}$ results in a belief state b' equal to b but with its i-th component set to r. (The existing i-th component of b must either be U or already be r if the indicators are consistent.) This simple transition model induces a directed *family belief graph* $G_{\mathcal{F}} = (V_{\mathcal{F}}, E_{\mathcal{F}})$ where $V_{\mathcal{F}} = \mathcal{B}$ and each edge $e_{\mathcal{F}} \in E_{\mathcal{F}}$ represents (S, r), the result of invoking an available indicator function $\mathbf{1}_S$ and having it return the binary result r.

For example, consider a family $\mathcal{F}$ consisting of two unrelated subsets, S_1 and S_2. The family belief graph $G_{\mathcal{F}}$ illustrated in Figure 6.7 shows all possible belief transitions on the graph.

6.4.2 *Subset Relations and the Family Belief Graph*

In a family motion planning problem, the family of subsets $\mathcal{F}$ is accompanied by a set of subset relations $\mathcal{R}$. These relations have implications on the family belief graph $G_{\mathcal{F}}$. To build the revised graph, we convert both the belief state b and the subset relations $\mathcal{R}$ into a set of *logical propositions*.

Beliefs and Subset Relations as Logical Propositions. A logical proposition is a statement consisting of propositional variables and logical operators. We will use bolded letters to denote propositional variables. A set of propositions $\mathcal{P}$ can be used as *premises* as part of an *argument* to demonstrate a *conclusion*; a logical solver can then be used validate or invalidate the argument. For example, an argument with these premises and conclusion $\{(\mathbf{A} \Rightarrow \mathbf{B}), (\neg\mathbf{B})\} \Rightarrow (\neg\mathbf{A})$ can be shown to be valid.

Any belief state b can be represented as a set of logical propositions. Consider again the simple family from Figure 6.7 consisting of subsets S_1 and S_2 (with $S_i \subseteq \mathcal{C}$). We will introduce the propositional variable $\mathbf{S}_i$ as follows: for some query configuration $q \in \mathcal{C}$, the proposition $\mathbf{S}_i$ implies $q \in S_i$, whereas $\neg\mathbf{S}_i$ implies $q \notin S_i$. Therefore, a belief state b can be converted into a set of propositions $\mathcal{P}_b$:

We keep distinct notation for the indicator function predicate $\mathbf{1}_S$ over $\mathcal{C}$ and the corresponding propositional variable $\mathbf{S}$; the former is used as a subroutine to be invoked to determine subset membership, whereas the latter is used to represent actual or hypothesized assignments to the propositional logic engine.

$$\mathcal{P}_b = \{\mathbf{S}_i \; \forall \, i \mid b[i] = \mathrm{T}\} \cup \{\neg\mathbf{S}_i \; \forall \, i \mid b[i] = \mathrm{F}\}. \tag{6.5}$$

For example, belief $b = (T, F)$ corresponds to $\mathcal{P}_b = \{\mathbf{S}_1, \neg\mathbf{S}_2\}$, and belief $b = (U, U)$ corresponds to $\mathcal{P}_b = \emptyset$.

The set of subset relations $\mathcal{R}$ can also be represented as a set of propositions. In particular, the containment relation $A \subseteq B$ (Figure 6.6a) yields $\{(\mathbf{A} \Rightarrow \mathbf{B})\}$ and the intersection relation $A = B \cap C$ (Figure 6.6b) yields $\{(\mathbf{B} \wedge \mathbf{C} \Rightarrow \mathbf{A}), (\mathbf{A} \Rightarrow \mathbf{B}), (\mathbf{A} \Rightarrow \mathbf{C})\}$. Converting each relation in $\mathcal{R}$ in this way yields an equivalent set of relational propositions $\mathcal{P}_{\mathcal{R}}$.

Subset Relations Prune the Family Belief Graph. The relational propositions $\mathcal{P}_{\mathcal{R}}$ together with the belief propositions $\mathcal{P}_b$ for each belief state b given by (6.5) can then be used along with a propositional logic engine to compute the family belief graph for a particular family motion planning problem. In particular, consider a given belief state vertex $b \in V_{\mathcal{F}}$ and a proposed out-edge (S, r). Let $\mathcal{P}_e$, the resulting additional set of propositions for this edge, be $\{\mathbf{S}\}$ if $r = $ True, or

$\{\neg\mathbf{S}\}$ otherwise. Then i-th component of the successor belief state b' is formed by considering the following two propositional arguments:

$$b'[i] = \begin{cases} \text{True} & \text{if } \mathcal{P}_{\mathcal{R}} \cup \mathcal{P}_b \cup \mathcal{P}_e \Rightarrow \mathbf{S}_i \text{ is valid} \\ \text{False} & \text{if } \mathcal{P}_{\mathcal{R}} \cup \mathcal{P}_b \cup \mathcal{P}_e \Rightarrow \neg\mathbf{S}_i \text{ is valid} \\ \text{Unknown} & \text{otherwise.} \end{cases} \qquad (6.6)$$

We show the effect of subset relations $\mathcal{R}$ on the family belief graph for our simple example family with an additional containment relation in Figure 6.8. The additional proposition from $\mathcal{P}_{\mathcal{R}}$, namely that $\mathbf{S}_1 \Rightarrow \mathbf{S}_2$, prunes the original graph in Figure 6.7 by removing inconsistent belief states and transitions.

We also illustrate the family belief graph for the more complex family motion planning problem in Figure 6.4. Recall that the family $\mathcal{F}$ consists of five $\mathcal{C}$-subsets, A, B, C, S_{12}, and S_{23}, and $\mathcal{R}$ consists of the subset relations $S_{12} = A \cap B$ and $S_{23} = A \cap C$. The family belief graph is shown in Figure E.1 in Appendix E.

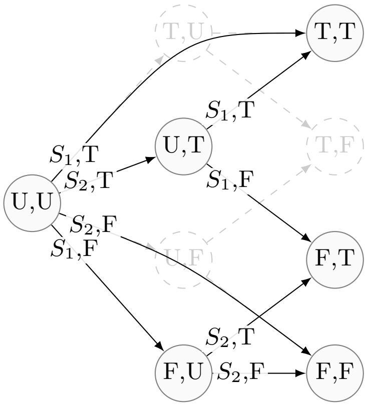


Figure 6.8: Example family belief graph for a family of two subsets, S_1 and S_2, with the subset relation that $S_1 \subseteq S_2$. Compare this family belief graph to Figure 6.7 without the relation.

6.4.3 *A Family Utility Model*

A key input to the LEMUR planner from Chapter 5 is the ensemble edge cost model $\mathcal{M}$ which captures domain-specific estimates of both the execution cost $\hat{x}$ and the remaining planning cost $\hat{p}$ for each edge on its roadmap. How can we take advantage of our family belief graph in order to implement such a model for the family motion planning problem?

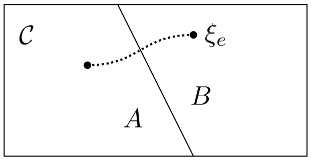
Recall that the formulation of the family motion planning problem (Section 6.3) includes for each subset indicator function $\mathbf{1}_S$ an invocation cost estimate $\hat{p}_S$. Our key insight is that our ensemble edge cost model $\mathcal{M}_{\text{family}}$ can (a) maintain for each edge e its current belief state b_e within and between each planning query, and (b) leverage the family belief graph in order to compute for any current edge an optimistic optimal sequence of indicator function invocations in order to demonstrate that the edge e is a member of the query subset S_u.

Beliefs from Configurations to Edges. Paramount in our approach is the ability to reason over edge beliefs. So far in Section 6.4, we have developed a family belief graph $G_{\mathcal{F}}$ informed by the set of subset relations $\mathcal{R}$ which allows us to reason about the evolution of our belief over the subset memberships of a query configuration $q \in \mathcal{C}$. Can we use this same belief graph to reason about edge beliefs as well?

While this is not generally true for arbitrary subset relations, we can demonstrate that it is true for the particular relations that we are considering – namely, containments and intersections. Consider the following definition for a subset $S \subseteq \mathcal{C}$. Consider an edge $e \in E$ which corresponds to a particular trajectory $\xi_e : [0,1] \rightarrow \mathcal{C}$ as described in Chapter 2. We will define edge set membership as:

$$e \in S \iff \xi_e(t) \in S \;\; \forall \;\; t \in [0,1]. \tag{6.7}$$

Consider the containment relation $A \subseteq B$; it can be easily shown by (6.7) that if $e \in A$, then $e \in B$. A similar statement can be made about the intersection relation $A = B \cap C$. If $e \in A$, then $e \in B \cap C$ and vice versa.



$A = C \setminus B$

Figure 6.9: A subset relation which does not carry over directly from configurations to edges.

An example of a set relation for which the edge membership relations do not follow directly from the configuration membership relations is shown in Figure 6.9. In this example, the subset relation $A = C \setminus B$ implies that if $q \in A$, then $q \notin B$. However, this is not true in general for edges. Fortunately, containment and intersection relations are sufficient for all instances of family motion planning problems that we consider.

Computing an Optimistic Optimal Belief Policy. Using the invocation cost estimates $\hat{p}_S$ for each subset $S \in \mathcal{F}$, we can create an edge weight function $w_{\hat{p}} : E_{\mathcal{F}} \rightarrow \mathbb{R}$ as

$$w_{\hat{p}}(e) = \hat{p}_S \;\text{ for }\; e = (S, r). \tag{6.8}$$

For the current motion planning query in the i-th subset S_u, we also identify all belief states $b \in V_{\mathcal{F}}$ for which $b[i] = \text{True}$ as goal vertices, and compute the distance function $d_{S_u} : V_{\mathcal{F}} \rightarrow \mathbb{R}$ and the corresponding optimal policy over the family belief graph $G_{\mathcal{F}}$ using a reverse Dijkstra's search. An example of this policy is shown in Figure 6.10.

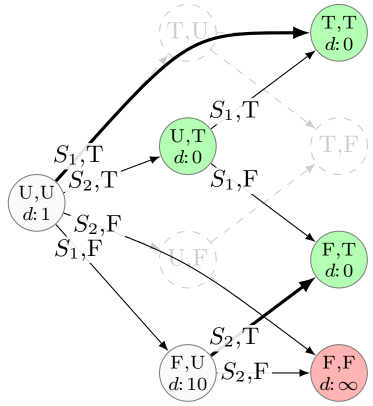


Figure 6.10: Optimistic optimal belief graph policy for a family of two subsets, S_1 and S_2, with the subset relation that $S_1 \subseteq S_2$, and with $\hat{p}_1 = 1$ and $\hat{p}_2 = 10$. The query subset is $S_u = S_2$; beliefs for which $b[2] = \text{T}$ are goal vertices (green). The value function d is shown for each believe state, and infeasible beliefs (i.e. with $d = \infty$, which are already demonstrably not in the query subset) are shown in green. The edges on the optimal policy are shown in bold.

The Belief-Informed Family Utility Model. We are now ready to define the ensemble edge cost utility model $\mathcal{M}_{\text{family}}$ in Algorithm 11. Note that the implementation of $\mathcal{M}_{\text{family}}$ depends on the current query subset S_u.

Once the optimistic optimal distance function and policy have been computed, calculating the planning and execution cost estimates for an edge e entails looking up the its current belief state b_e and then reading the value of the distance function d_{S_u} at the corresponding vertex. The conditionals inside each implementation take advantage of the fact that beliefs with $d_{S_u} = \infty$ correspond to certain knowledge that the edge e is not a member of the query subset S_u.

During each iteration of the LEMUR algorithm over its roadmap, the planner will find a candidate path according to its λ_p tradeoff parameter, and will then select an edge which is not yet fully evaluated (i.e. has $\check{p}_{\text{family}} > 0$). It will then commit to evaluating that edge. Conducting the LEMUR search with the $\mathcal{M}_{\text{family}}$ edge cost model

Algorithm 11 Family Edge Cost Model $\mathcal{M}_{\text{family}}$

```
function $\check{p}_{\text{family}}(e)$                    ▷ remaining plan-cost estimator
    $b_e \leftarrow$ stored belief state of edge $e$
    if $d_{S_u}(b_e) < \infty$ then
        return $\sum_{q \in e} d_{S_u}(b_e)$
    else
        return $0$
function $\hat{x}_{\text{family}}(e)$                              ▷ exec-cost estimator
    $b_e \leftarrow$ stored belief state of edge $e$
    if $d_{S_u}(b_e) < \infty$ then
        return $0$
    else
        return $\infty$
```

requires a modified edge cost determination procedure x_{family} shown in Algorithm 12. The purpose of this procedure is to (a) delegate the edge evaluation to the correct edge subset indicator functions $\mathbf{1}_S[\cdot]$, and (b) to maintain the correct belief state b_e for the edge as the actual indicator results are returned.

In addition, note also that depending on the particular set relations $\mathcal{R}$, a single invocation of the procedure x_{family} may not fully evaluate the edge – for example, in the belief graph from Figure 6.10, the optimistic optimal path selects the S_1 indicator, optimistically assuming that it will return True, the belief state b_e will transition to (T, T), and the edge will be demonstrated valid. But if the indicator returns False, the belief state b_e instead transitions to the complementary belief state (F, U) which is at the target of the complementary edge (S_1, F). At this point, the edge is still not fully evaluated, and is therefore available to LEMUR on a subsequent iteration if the utility objective demands it.

6.5 *Application: Multi-Step Manipulation Tasks*

We conducted an empirical test of LEMUR using the family edge cost model (Algorithm 11) on multi-step manipulation planning problems on two different robotic platforms. In each case, we compared its performance with that of the simple edge cost model (Algorithm 10 from Chapter 5). We also include in the comparison the RRT-Connect algorithm from OMPL.

6.5.1 *Table-Clearing with HERB*

The first task we considered is the table clearing task from Figure 6.1. Each condition for LEMUR was conducted over 50 randomly gener-

Algorithm 12 Family Edge Cost Procedure

```
procedure $x_{\text{family}}(e)$                                  ▷ determine exec cost
    $b_e \leftarrow$ stored belief state of edge $e$
    if $d_{S_u}(b_e) < \infty$ then
        $\pi_{\mathcal{F}} \leftarrow$ optimal path from $b_e$ on $G_{\mathcal{F}}$
        for all $e_{\mathcal{F}} \in \pi_{\mathcal{F}}$ do
            $(S_{\text{opt}}, r_{\text{opt}}) \leftarrow e_{\mathcal{F}}$
            $r_{\text{act}} \leftarrow \mathbf{1}_{S_{\text{opt}}}[e]$
            if $r_{\text{act}} \neq r_{\text{opt}}$ then
                $b_e \leftarrow$ target(complement($e_{\mathcal{F}}$))
                break
            $b_e \leftarrow$ target($e_{\mathcal{F}}$)
    return $\hat{x}_{\text{family}}(e)$
```

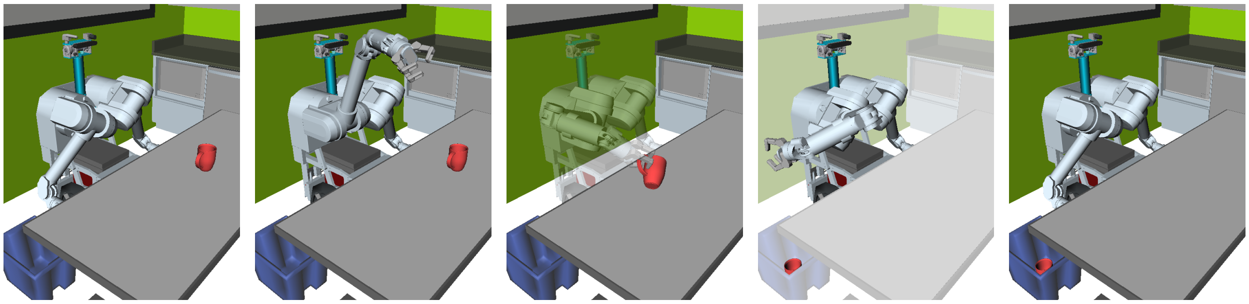

(a) Starting Configuration (b) Step 1, in S_{AB} (c) Step 2, in S_{BC} (d) Step 3, in S_{CD} (e) Ending Configuration.

Figure 6.11: A home robot performing a three-step manipulation task. It must move from its home configuration to grasp the cup, transfer it to a drop location above the bin, and return home.

ated Halton roadmaps with 10,000 milestones per batch and a r_{loglog} connection radius schedule with $\eta = 1$ (see Section 2.4). We varied the λ_p planning-vs.-execution tradeoff parameter between 0.0 to 0.99 (Chapter 5), and selected either the simple edge cost model $\mathcal{M}_{\text{simple}}$ or the family edge cost model $\mathcal{M}_{\text{family}}$ described above.

The task consists of 3 steps: the initial transit to the mug (AB), the transfer to a drop location (BC), and the final transit back to the home configuration (CD). The family module computed a decomposition of the full task and discovered 7 relevant C-space subsets over the full planning problem listed in Table 6.1. An illustration of the subsets is also shown in Figure 6.11.

Subset	Geometry 1	Geometry 2
S_1	RobotMoving	RobotMoving + EnvStatic
S_2	RobotMoving	MugPlacedTable
S_3	RobotMoving	MugPlacedBin
S_4	GraspedMug	RobotMoving + EnvStatic

Subset	Defined as
S_{AB}	$S_1 \cap S_2$
S_{BC}	$S_1 \cap S_4$
S_{CD}	$S_1 \cap S_3$

Table 6.1: Constituent subsets in the `herbbin` example problem.

We measured the length of the path returned by each algorithm (rad), as well as the total wall clock planning time (s).

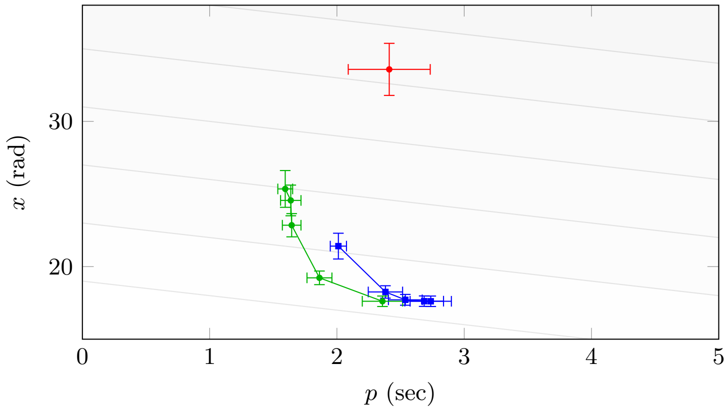


Figure 6.12: Comparison of measured planning time p and solution execution cost x for the HERB table-clearing example. RRT is shown in red 🟥. Results for LEMUR with the simple edge cost model are shown in blue 🟦, while the family edge cost model is shown in green 🟩. The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 (lower right) to 0.99 (upper left).

6.5.2 *Tending a Press Brake with an IRB 4400*

The second task we considered is a sheet metal bending task using an ABB IRB 4400 robot in an industrial workcell. See Figure 6.13 for an illustration of the task. Each condition for LEMUR was conducted over 50 randomly generated Halton roadmaps with 10,000 milestones per batch and a r_{loglog} connection radius schedule with $\eta = 1$ (see Section 2.4). We varied the λ_p planning-vs.-execution tradeoff parameter between 0.0 to 0.99 (Chapter 5), and selected either the simple edge cost model $\mathcal{M}_{\text{simple}}$ or the family edge cost model $\mathcal{M}_{\text{family}}$ described above.

The task consists of 8 steps, as described in Figure 6.14. The family module computed a decomposition of the full task and discovered 17 relevant C-space subsets over the full planning problem listed in Table 6.2. Note that the motion plans for steps BC and CD are in the same subset.

Subset	Geometry 1	Geometry 2
S_1	RobotMoving	RobotMoving + EnvStatic
S_2	RobotMoving	SheetPlacedStart
S_3	RobotMoving	SheetPlacedSettling
S_4	RobotMoving	SheetPlacedDestination
S_5	GraspedTopSide	RobotMoving + EnvStatic
S_6	GraspedTopSide1Flat	RobotMoving + EnvStatic
S_7	GraspedTopSide1Bent	RobotMoving + EnvStatic
S_8	GraspedBotSide	RobotMoving + EnvStatic
S_9	GraspedBotSide2Flat	RobotMoving + EnvStatic
S_{10}	GraspedBotSide2Bent	RobotMoving + EnvStatic

Subset	Defined as
S_{AB}	$S_1 \cap S_2$
S_{BC}	$S_1 \cap S_5 \cap S_6$
S_{CD}	$S_1 \cap S_5 \cap S_6$
S_{EF}	$S_1 \cap S_5 \cap S_7$
S_{FG}	$S_1 \cap S_3$
S_{GH}	$S_1 \cap S_8 \cap S_9$
S_{IJ}	$S_1 \cap S_8 \cap S_{10}$
S_{JA}	$S_1 \cap S_4$

Table 6.2: Constituent subsets in the `workcell` example problem.

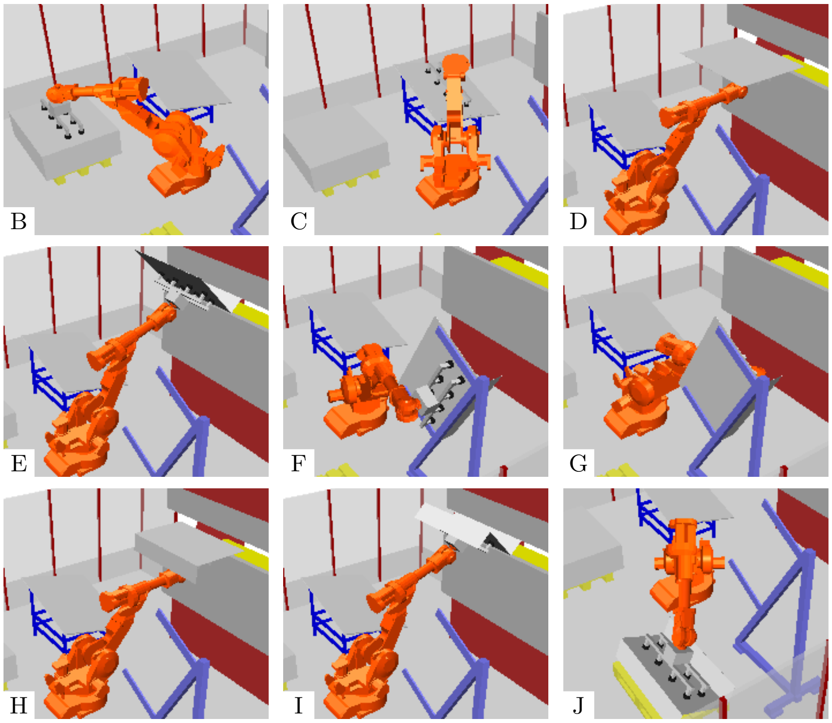


Figure 6.13: Robot tending a press brake in an industrial workcell. This example problem is reproduced from the Lazy PRM paper [10]. The multi-step manipulation task requires eight motion planning problems in seven different $\mathcal{C}$-subsets. From its initial configuration (A), the robot moves to grasp a raw sheet (B) and transfer first to a settling table (C) and subsequently to the press brake (D). After the first bend, the robot receives the partially worked part (E) and uses a regrasping fixture (F) to regrasp it on the other side (G). After its second bend (H) and (I), the finished part is moved to the final pallet (J) before the robot returns to its initial configuration (A). Results for planning these steps are shown in Figure 6.14.

We measured the length of the path returned by each algorithm (rad), as well as the total wall clock planning time (s).

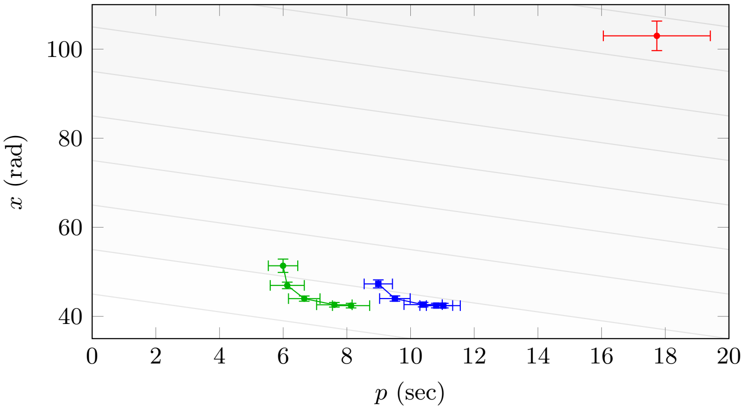


Figure 6.14: Comparison of measured planning time p and solution execution cost x for the industrial workcell example. RRT is shown in red 🟥. Results for LEMUR with the simple edge cost model are shown in blue 🟦, while the family edge cost model is shown in green 🟩. The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 (lower right) to 0.99 (upper left).

6.6 *Implementation Details*

We provide an implementation for the OpenRAVE [28] virtual kinematic planning environment which automatically discovers $\mathcal{C}$-subsets in manipulation tasks.

7
Conclusion

This dissertation presents a motion planning approach suited for multi-step manipulations tasks. Key to the approach are the complementary ideas of *lazy* and *utility-guided* search, which we integrated into the LEMUR motion planning algorithm. This approach has shown promising results on manipulation tasks when compared with state-of-the-art search-based and sampling-based anytime planners. This concluding chapter begins by summarizing the dissertation and the individual algorithmic contributions in Section 7.1. This is followed by a discussion in Section 7.2 of a few promising avenues for future research which build on our results, as well as some lessons learned in Section 7.3. Finally, we offer some concluding remarks in Section 7.4.

7.1 Summary and Contributions

This dissertation focuses on motion planning problems which arise in autonomous manipulation tasks. Such tasks induce continuous, high dimensional robot configuration spaces in which path validity checking is particularly expensive due to complex robot kinematics and workspace geometry. Furthermore, there is an inherent cost tradeoff between planning a motion and subsequently executing it. For manipulation tasks in human-scale environments in particular, whether measured in time or energy, the cost of planning (principally path validity checking) is comparable to the cost of execution. Therefore, reasoning over both sources of cost – and their tradeoff – is paramount.

To address these challenging motion planning problems, this dissertation proposes a collection of algorithms (Figure 7.1) which work in concert to minimize both planning and execution cost. We first motivate our focus on roadmap methods for motion planning, and then consider two central questions concerning the search over those roadmaps: (a) how should we conduct the optimization, and (b) what

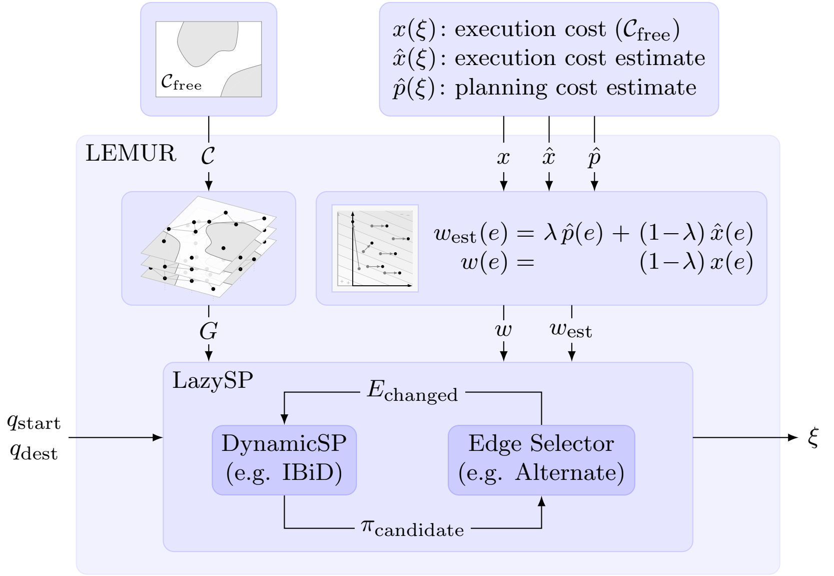

$\mathcal{C}_{\text{free}}$
$x(\xi)$: execution cost ($\mathcal{C}_{\text{free}}$)
$\hat{x}(\xi)$: execution cost estimate
$\hat{p}(\xi)$: planning cost estimate
LEMUR
$\mathcal{C}$
x
$\hat{x}$
$\hat{p}$
$w_{\text{est}}(e) = \lambda\,\hat{p}(e) + (1-\lambda)\,\hat{x}(e)$
$w(e) = (1-\lambda)\,x(e)$
G
w
w_{est}
LazySP
E_{changed}
q_{start}
q_{dest}
DynamicSP (e.g. IBiD)
Edge Selector (e.g. Alternate)
$\pi_{\text{candidate}}$
ξ

Figure 7.1: Outline of the algorithms developed in this dissertation. LEMUR is solves a continuous motion planning problem via discretization as a series of progressively densified roadmaps (Chapter 2). Since the shortest path problem over the resulting graph in characterized by edge costs which are expensive to evaluate, we exploit lazy search and edge selectors (Chapter 3) to minimize planning effort. We also develop a novel incremental bidirectional search algorithm (Chapter 4) to accommodate the resulting dynamic pathfinding problem. LEMUR conducts its search guided by a utility function (Chapter 5) which can employ distinct domain-specific planning and execution cost heuristics. In multi-step manipulation tasks, one such cost model is derived from the family motion planning problem (Chapter 6), which leads to planner invocations which minimize combined planning and execution cost.

objective should we optimize?

7.1.1 *Motion Planning via Roadmaps*

In Chapter 2, we outlined the motion planning problem, as well as a selection of algorithms designed to solve it. A wide variety of such algorithms exist, including optimization-based, search-based, and sampling-based planners. We motivate our focus on sampling-based roadmap planners based on the following advantages:

- Roadmap methods are well-studied and possess favorable theoretical properties such as asymptotic optimality.
- It is straightforward to incrementally densify the discretization by adding new samples to the roadmap.
- The cost of constructing a roadmap which is insensitive to the distribution of obstacles can be amortized across all planning queries (e.g. loaded from disk and persisted in memory between queries).

The algorithms presented in this dissertation are agnostic to the particular roadmap structure used for the discretization. Our experiments were conducted over roadmaps generated via Halton sequences, with edges between vertices within a connection radius determined by the critical value $\gamma^*_{\mathcal{C}}$. For example, the LEMUR motion planner (Algorithm 9 in Chapter 5) densifies its roadmap by introducing new batches of vertices and edges; in our experiments, we reduce the connection radius as recommended in [56]. We discuss future avenues for treating roadmaps in Section 7.2.1.

7.1.2 *How to Optimize?*

Committing to a particular discretization (e.g. roadmaps) reduces the continuous motion planning problem to a discrete graph pathfinding problem. The first central question considered in this dissertation, then, is how a motion planning objective should be optimized over such a roadmap graph.

Lazy Pathfinding. Pathfinding for motion planning problems is distinguished from other shortest-path applications because the primary component in its edge objective depends on its *validity*, which expensive to determine for articulated robots. This motivates lazy pathfinding approaches, which decouple the search for candidate solutions from the process of evaluating each solution for its cost.

The first contribution of this dissertation is a study of the LazySP algorithm outlined in Chapter 3, which allows for this edge evaluation to be specified arbitrarily by way of an *edge selector* function. We show that simple selectors are equivalent to the existing Weighted A* and Lazy Weighted A* pathfinding algorithms, and we also consider the efficacy of bidirectional and bisection selectors.

Contribution: The LazySP algorithm, which exploits an edge selector to find paths while minimizing the number of necessary edge weight evaluations.

We then introduce novel selectors based on path distributions, which focus evaluations towards edges which most likely lie on a shortest path. One such path distribution selector is based on the partition function over paths on a graph. We develop an incremental method for We show that these selectors can lead to fewer expected edge evaluations than their simple alternatives over a set of example problems. We also motivate why the alternating strategy serves as a simple proxy for the more complex path distribution selectors for common cases where no a priori knowledge of the obstacle distribution exists.

Contribution: an incremental method to maintain the partition function over paths under changing edge weights.

We note that lazy planning over roadmaps accomplishes the same evaluation focusing behavior exhibited by informed sampling-based motion planners [40, 41]. LazySP naturally focuses evaluations only to areas which might improve path cost of the particular query under consideration, but due to the edge selector is not constrained to evaluate edges only in cost-to-come order.

Dynamic Pathfinding. Conducting our roadmap search over candidate paths via lazy pathfinding induces an underlying inner dynamic shortest path problem. This arises because the decoupled nature of lazy search is indistinguishable from a search over an unknown or changing objective. Each iteration of our lazy search constitutes a new dynamic planning episode – after our edge selector nominates particular edges of our roadmap for evaluation and their true costs are determined, the inner search must accommodate these changes in order to produce new candidate paths on subsequent iterations.

Existing incremental algorithms for dynamic pathfinding problems, such as Lifelong Planning A* [67], make use of heuristic potential functions to reduce the subset of the graph that must be considered when solving each planning episode. During a lazy search, the strength of this heuristic depends on the relationship between the true and estimated edge weights (w and w_{est} in LazySP, respectively) – and in many cases, a strong lower bound on w, and therefore a strong heuristic, may not exist. This motivates our study of the dynamic problem in Chapter 4.

A common approach to solving pathfinding problems in domains without a strong heuristic is the notion of bidirectional search. Unifying bidirectional and incremental methods presents a unique challenge, since the subtle bidirectional termination condition must be posed in such a way that it remains applicable even under the changing edge weights which arise in a dynamic problem. This allows us to formulate the Incremental Bidirectional Dijkstra's (IBiD) algorithm, which generalizes the bidirectional Dijkstra's algorithm [45] in the same way that the DynamicSWSF-FP algorithm [98] generalizes the original Dijkstra's algorithm [29]. We also show how this algorithm can be adapted in the presence of heuristic potential functions. This algorithm shows promising results both on road network routing problems (Chapter 4) as well as inflated lazy search problems from motion planning.

Contribution: the IBiD algorithm, an incremental and bidirectional shortest path algorithm for dynamic graphs.

7.1.3 *What to Optimize?*

Our treatment of lazy roadmap search deliberatively leaves open the question of what objective should be optimized. The approach is applicable to any pathfinding domain in which the edge weight function w is expensive to evaluate, and where an inexpensive estimate w_{est} may be available. In the case of motion planning for articulated robots, where planning and execution cost are both important, how should we instantiate these functions?

Search over Candidate Utility. In Chapter 5, we introduce the notion of *utility*, and argue that it is the correct form of the objective in order to capture the tradeoff between planning and execution costs. Our conception of utility borrows heavily from the BUGSY algorithm [14] for conventional graph search. The key insight is that marrying utility with lazy search allows for the incorporation of planning cost estimators over concrete candidate paths – and not just over frontier vertices. This enables many domain-specific planning heuristics to be represented naturally.

We next show that by committing to (a) a utility function that is linear in planning and execution cost, and (b) estimators that are additive over path edges, we can represent our utility optimization as a lazy shortest path problem. Mediated by the tradeoff parameter λ_p, the LEMUR algorithm conducts this optimization over a sequence of progressively densified roadmaps.

Contribution: the LEMUR motion planning algorithm which conducts a lazy search guided by path utility over a set of sequence of densified roadmaps.

We examine a set of manipulation planning tasks using three robot platforms, and conduct experiments using a planning cost heuristic which captures the remaining collision checks on each candidate path. We show that LEMUR exhibits favorable performance when compared against a set of anytime planners such as RRT-Connect, BIT*, and Lazy ARA*.

Utility Estimates in Manipulation Planning. Importantly, the utility-based objective described in Chapter 5 allows arbitrary domain-specific planning cost estimates to be leveraged. Chapter 6 explores intelligent heuristics in manipulation tasks by formulating such multi-step tasks as a *family motion planning problem*. We show this representation, in which multi-step tasks are given over a family of related C-space subsets, allows for a planning cost heuristic which naturally incentivizes a utility-based planner (such as LEMUR) to reuse planning computation from prior queries.

Contribution: the family motion planning problem a formulation of manipulation planning that yields a natural planning cost estimate for utility-guided motion planners.

7.2 Future Directions

Treating motion planning as a lazy search over roadmaps prompts an array of promising directions for further work. Improvements in roadmap discretization, focusing of edge evaluations, better inner search algorithms, more expressive domain-specific planning and execution cost heuristics, and probabilistic optimization techniques might all improve performance over a wide array of planning domains. In this section, we survey a few such promising directions that arise from our approach.

7.2.1 Improved Roadmaps

One of the fundamental questions in any motion planning approach is how should the continuous problem be approximated in order to search for a solution. The approach to motion planning presented in this dissertation relies fundamentally on the roadmap discretization method. The approach can therefore benefit from any technique which improves the efficacy of this discretization.

Better Connection Heuristics. Roadmap methods rely on a connection rule to determine whether two vertices representing configurations in the C-space should be connected with an edge – the existence of such an edge implies that the local planner has high likelihood of successfully finding a valid path between them. To this end, most approaches make use of arbitrary thresholds of simple distance functions over $\mathcal{C}$ as their decision rules. In our experiments, for example, we used a simple Euclidean distance rule.

However, for articulated robots (and especially for recurring tasks in similar environments), it is easy to conceive of more intelligent connection rules. For example, consider a naïve execution cost model in which obstacles are distributed uniformly in the robot's workspace. In this case, the likelihood of collision over some edge length depends on the configuration of the arm itself – configurations in which the Jacobian matrix is large (where changes in the location of robot geometry are large with respect to changes in configuration) entail higher collision probability. In the case of recurring motion planning queries for the same robot geometry, it would seem beneficial to pre-compute a roadmap structure in which the edge connection rule depends on some similar measure of the robot's configuration.

7.2.2 *Adaptive Roadmap Densification via an Infinite Roadmap Stack*

The LEMUR planner effectively conducts its search over a sequence of progressively densified roadmaps (Figure 7.2). The results presented in this thesis have used Halton sequences as discussed in Chapter 2 for their preferable dispersion properties, and form a roadmap graph using an r-disk connection rule. As discussed in [56], the shortest path on such a sequence of roadmaps possesses asymptotic optimality if the connection radius is adjusted appropriately.

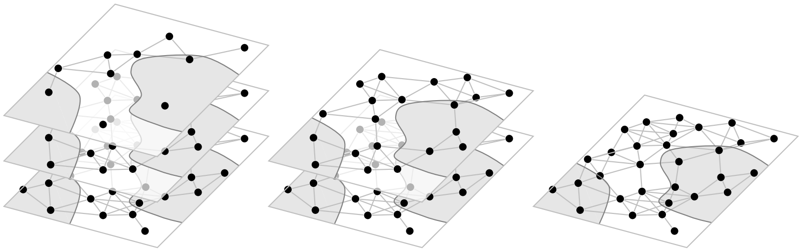


Figure 7.2: A stack of progressively densified roadmaps over a given free configuration space $\mathcal{C}$.

Hard batching. Like many existing roadmap-based algorithms [114, 41], LEMUR implements a hard batching approach to densification – a search is conducted fully over each batch of the roadmap in $\mathcal{C}$ until a solution is found. While the specification of LEMUR allows for this roadmap to be implicitly constructed, the current implementation constructs the entire batch before proceeding with its search. Once the search over that roadmap completes without a feasible path found, the next batch of vertices and edges are added to the graph, and the newly densified roadmap is searched.

Because the sequence of roadmaps that is searched is insensitive to the distribution of obstacles, the current implementation of LEMUR is able to pre-compute a sequence of roadmaps up to a certain level of discretization, and then load that roadmap structure into memory directly instead of building it from scratch for each planning query.

There are two shortcomings that arise from this hard batching approach:

- *Uniform Densification:* The fact that this roadmap is loaded uniformly across the space is certainly a limitation that will become restrictive in spaces of larger dimension.
- *Densification vs. Resolution Completeness:* The current LEMUR algorithm only moves to a denser roadmap once no finite paths are shown to be available on the present roadmap. This may not be desirable in large spaces – it may make sense to move to a denser roadmap in the vicinity of a short path before exploring all corners of the space.

There are a number of promising avenues for a solution to the problems that arise from uniform densification.. Informed anytime algorithms [40, 41] restrict densification once an initial path is found to a subset of the full space that may contain better solutions. This approach is not directly applicable because LEMUR is not an anytime algorithm.

Adaptive Roadmap Densification. An alternative strategy is to adopt an adaptive densification approach under which the search is conducted upon an infinite stack of progressively densified roadmaps.

Consider the roadmap stack depicted in Figure 7.3. Here, as before, each underlying roadmap is a superset of the one above it. But now, instead of progressively considering each layer individually (using batching), consider conducting a single search over the entire stack. Note that the stack includes additional edges (shown dotted in the figure) connecting corresponding vertices in two adjacent layers; using a road network analogy, we call these edges “offramp” edges.

A proposed edge weighting scheme involves two components: (a) an artificial inflation of edge weights by some factor on each layer according to a schedule (with edges on lower layers inflated more), and (b) an artificial constant weight assigned to each offramp edge between layers.

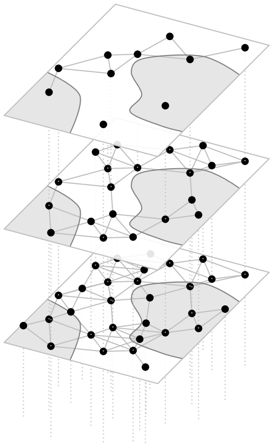


Figure 7.3: A roadmap stack with "offramp" edges.

Consider a search over this infinite stack G between two start and destination configurations corresponding to distinguished vertices on G, and consider an optimal solution path with strong δ-clearance. If the roadmap layers satisfy the appropriate conditions (e.g. connection radius [61, 56]), and all roadmap edges have nonnegative weight, and all "offramp" edges have some constant positive weight α, then we conjecture that there exists a shortest path p through G of finite length (which traverses a finite number of layers of G). Furthermore, we conjecture that the length of p approaches the length of an optimal path as $\alpha \to 0$.

We find this representation of the continuous problem compelling because it naturally integrates the densification problem with the search problem. One can imagine a unidirectional or bidirectional search effectively trading off between exploring more widely on a particular layer and descending to a denser layer in order to traverse a narrow passage. Descending to a denser layer occurs naturally during the search, and denser regions can be sampled on demand only in areas of the space adjacent to obstacles blocking the path. Furthermore, the computational cost of such densification can be captured as an additional planning cost component in the LEMUR algorithm, so it will commit to a denser layer only when the predicted path savings outweigh the requisite planning cost.

Opportunity for a more accurate planning cost estimate: adding a term for expected cost of a further roadmap densification.

7.2.3 *Improved Lazy and Dynamic Search*

The core of our approach is a lazy search over candidate paths, which makes use of an inner dynamic shortest path problem in order to select candidates at each iteration. There are a number of potential avenues available for making this search more efficient.

Intelligent Selection of Key Edges. In Chapter 3, we presented novel edge selectors which maintain a distribution over candidate paths in order to focus edge evaluations towards edges which most likely lie on a short path. While this showed promising results across a range of problem instances, maintaining this distribution can incur its own computational cost (quadratic in the number of vertices in the case of the Partition selector), and therefore it can be intractable over large roadmap graphs.

The effect of these selectors is to automatically discover and focus on the parts of the motion planning problem that are most constrained. It would be worthwhile to explore ways of more efficiently maintaining such path distributions for larger graphs. In the case of a bidirectional evaluation strategy (such as the Alternate selector), it may be beneficial to attempt to discover online which end of the problem proves to be the most constrained.

It may also be instructive to investigate the "oracle" selector. Consider that for any graph, there exists some minimal set of edges which, if evaluated, would probably demonstrate that the best remaining candidate is correct. Investigating the distribution of edges required for such a selector for particular problem instances may aid in designing new selectors. It also serves as a useful benchmark against which to compare performance.

Exploring the Interaction between Lazy and Dynamic Search. Lazy search induces an inner dynamic shortest path problem, as edge weights are updated and intermediate candidate paths are required. When LEMUR addresses a problem with $\lambda_p > 0$, so that planning cost is to be considered in its utility objective, it generally selects candidates that do not solely minimize execution cost in favor of candidates that lower remaining planning cost.

Paradoxically, as the λ_p parameter is increased, the underlying dynamic search problem becomes more difficult, because its heuristic is no longer as strong. (A consistent heuristic typically only exists for the component of the edge weight function derived from the execution cost component.) It would therefore be beneficial to investigate schemes whereby the requirement for optimal shortest paths from the inner dynamic search is relaxed. For example, consider a variant of the LazySP algorithm in which the candidate path at each iteration only maximizes utility up to a particular suboptimality bound. One suitable algorithm for this relaxed dynamic search problem is Truncated Incremental Search [1].

Adjusting the Incremental Balancing Criterion. It is well-known [94] that using the cardinality of each OPEN list as the decision rule to balance expansion of the two sides of a conventional bidirectional search generally outperforms using the distance (key) value itself. Unfortunately, it is not clear how to adapt this criterion to the incremental search setting, since the size of the queue of inconsistent vertices is no longer indicative of the density of the graph on the two sides. Can we dynamically adjust the distance balance criterion so that the location of the interface between the two trees (represented as the edge connection queue) approximates this balanced cardinality criterion?

Modeling and Incorporating Search Cost. The LEMUR planning cost heuristic allows estimates of remaining planning cost to be incorporated into the planner's utility function. For our experiments in this dissertation, this planning cost term included a measure of the expected collision checking cost remaining. However, for larger or denser graphs, the time to conduct the inner search may become important to model as well. If such a model exists, the planner could (for example) use it to determine whether to proceed with a new search, or whether to terminate immediately with the best path it has found so far.

Opportunity for a more accurate planning cost estimate: adding a term for expected search cost.

7.2.4 *Tighter Integration with Collision Validity Checking*

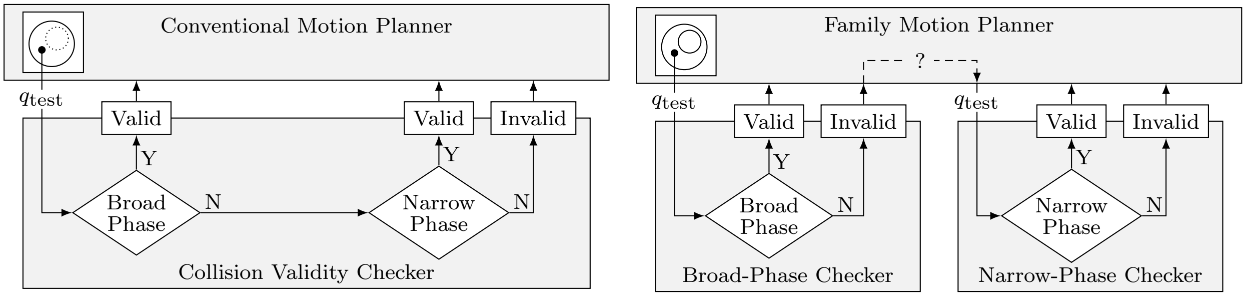
The basic motion planning problem described in Chapter 2 considers arbitrary configuration space obstacles. When most sampling and search based motion planners test candidate path segments for validity, they treat validity checking – usually via a geometric collision checker – as a "black box" operation. Given a test configuration q_{test}, such a checker considers the environment and the robot's kinematics, and returns a boolean answer – True or False. Depending on the fidelity of the geometric model of the robot and environment, each check can entail tens, hundreds, or even thousands of microseconds. There are a number of opportunities for extending LEMUR in order to better predict and account for the internal mechanisms of these checkers.

Chapter 6 explored one such opportunity – decomposing the validity checks performed over multiple steps of a manipulation task as a family motion planning problem. Doing so allows a planning cost model to be formulated naturally in terms of different subsets of C-space, so that a utility-aware planner (such as LEMUR) is incentivized to reuse partial validity information from cached or previous similar planning episodes. In this section, we discuss other opportunities for characterizing and breaking the black box collision checking abstraction.

Better Estimates of Expected Validity Checking Cost. The experiments in this dissertation relied on a simple model for the planning cost required for each collision check. First, a single averaged duration per collision check was learned over an array of problems for each robot platform. Then, during the course of a planning query, the remaining planning cost required for each candidate path consisted simply of this duration summed across all remaining configurations to be checked.

This simple model could be improved in at least two ways:

Opportunity for a more accurate planning cost estimate: a model for the different planning cost required in different parts of the space.



(a) A motion planner testing simply for membership in $\mathcal{C}_{\text{free}}$ treats a collision validity checker as a "black box." Internally, modern checkers first employ an inexpensive broad-phase check using a simple conservative representation to quickly identify non-colliding bodies before resorting to an expensive narrow-phase check.

(b) A family motion planner can explicitly reason about the conservative nature of the broad-phase check. This allows it to defer many narrow phase checks (often indefinitely) and instead prefer paths that require fewer expensive checks.

Figure 7.4: Collision validity checking is a commonly used indicator function. The family motion planning formulation allows an intelligent planner to reach inside the checker's "black box" and reduce the number of costly narrow-phase checks. Resulting paths tend to be cheaper to compute and stay further from obstacles.

- First, the planning cost required for a collision check could be learned dynamically during a planning episode. The cost per check depends on the complexity of the workspace obstacles and the areas of the configuration space actually explored, and a planner which incorporates such learning could better adapt to particular planning queries and manage its search more accurately.

- Second, the planning cost required for a collision check depends on the area of configuration space itself. For example, checks in wide open spaces tend to complete much more quickly than those near obstacle boundaries. A planning cost estimator could build and maintain a spatial model of this checking cost during the course of a planning episode, and adjust the predicted remaining planning costs for candidate paths as a result.

Exploiting Multi-Phase Collision Checkers. Many geometric collision checkers build a multi-phase hierarchy, both to quickly answer easy validity queries, and to focus computation to areas where determining validity requires checking at a finer resolution. One of the most widely used approaches entails building a simple conservative geometric approximation (e.g. a bounding box), and first checking that approximation for collision during an initial "broad-phase" check. Only geometry which fails this check is then subjected for subsequent (and more expensive) "narrow-phase" checking. See Figure 7.4 for example of such a multi-phase validity checker.

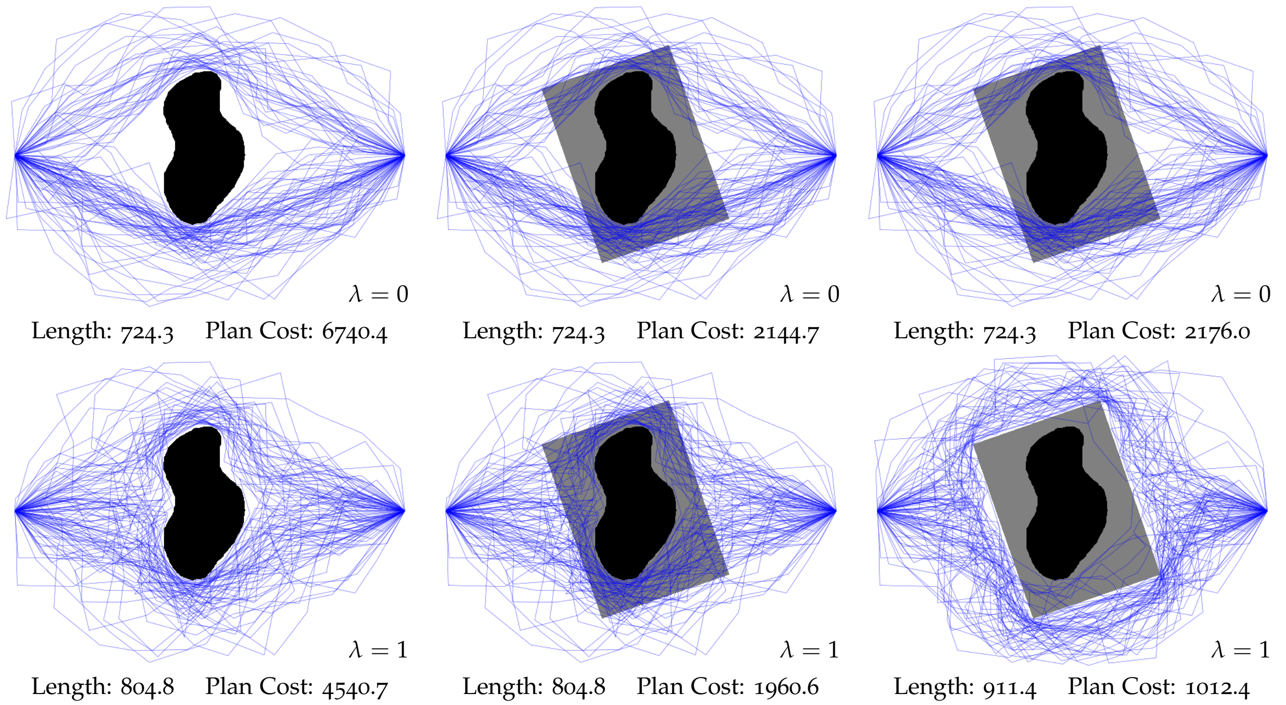
Most motion planners treat the validity checker as a binary "black box," and do not explicitly reason about this underlying multi-phase process. However, it may be advantageous for the planner to be made aware of these phases. This allows the planner to effectively identify configurations for which many (or all) geometric pairs are sufficiently far from collision that the more expensive narrow-phase checks need not be applied. In fact, this structure can be represented simply as a family motion planning problem. Consider the two subsets of configuration space $B \subseteq \mathcal{C}$ and $N \subseteq \mathcal{C}$ which satisfy the inexpensive broad-phase and expensive narrow-phase checks, respectively. It is clear that $B \subseteq N$, so that if indicators $\mathbf{1}_B[\cdot]$ and $\mathbf{1}_N[\cdot]$ are available with the former less expensive than the latter, the planner would prefer to evaluate the former before resorting to the latter.

We show an illustrative 2D example of this idea in Figure 7.5. Here, the behavior of LEMUR is shown, for the two extreme values of the λ tradeoff parameter, across three different models of the underlying collision checker. In the first model (a) at left, only the narrow-phase checker is available, which tests directly for collision against the black obstacle. At middle (b), the collision checker has available to it a broad-phase check, which first validates the test configuration against grey conservative approximation; this incurs a planning cost which is 10x less expensive than the narrow-phase checker. Since this is unknown to the motion planner, the solution paths are identical, but this does result in a significant savings in planning cost. At right (c), LEMUR is told about each checker explicitly, which allows it to defer many expensive narrow-phase checks indefinitely. The resulting paths tend to be more expensive to execute, since they largely stay free from the conservative approximation, but the planner realizes a significant savings in planning cost.

Maximizing Utility in Expectation. The strategy of progressively densifying the roadmap discretization (including the infinite stack approach discussed in Section 7.2.2) is commonly used for motion planning problems. Its purpose is to serve as a proxy for handling the spatial coherence between C-space obstacles in the scene. For example, consider motion planning in an environment with many very small randomly placed obstacles – in such a scene, it would be difficult to motivate using a coarse roadmap to confine the search. What if we handle this more explicitly?

The LEMUR algorithm exploits an optimistic assumption when considering candidate paths for prospective evaluation: it assumes all unevaluated edges are collision-free. While this assumption allows the inner loop to process candidates quickly, it requires the hard batching approach to strike an artificial balance between local and global exploration. It is worth investigating whether abandoning the optimistic assumption is worthwhile – a replacement could then reason explicitly about the collision probability across the configuration space.

We recently proposed a first step in this direction via the Pareto



(a) LEMUR with narrow-phase checking only, for $\lambda = 0$ (top) and $\lambda = 1$ (bottom).

(b) LEMUR with "black-box" broad-phase check (not exposed to planner).

(c) LEMUR with broad-phase check exposed as a distinct testable subset.

Figure 7.5: A simple 2D motion planning example in which LEMUR has a broad-phase check available. Testing against the bounding box (grey) is 10x less expensive than testing against the actual obstacle (black). Exposing the broad-phase check to LEMUR as a distinct subset allows for reduced planning cost.

Optimal Motion Planner [19]. In this work, we maintain a probabilistic model of the probability of collision across the configuration space, which we update incrementally as collision checks are performed against the environment. The planner then uses this model in order to select candidate paths for partial evaluation that minimize some combination of the path's execution cost and its probability of collision. We show that this combination is approximately equivalent to maximizing utility in expectation if each colliding path segment is assigned a penalty cost.

7.2.5 *Tighter Integration with Task Planning*

A motion planner is one component in a larger planning system for an articulated robot performing real-world tasks. For example, we have solved multi-step motion and manipulation planning tasks using a hierarchical planning architecture for the CHIMP robot [25]. LEMUR aims to serve as a performant lower-level geometric motion planner which is cognizant of both the quality of its solutions and the planning cost it incurs to find it, and we have found that LEMUR provides improved performance for such tasks.

Heuristics for Task and Motion Planners. While it can be used effectively in any task planning environment, there are opportunities for improved system performance by integrating the planner more tightly with other system components. Recent work has married symbolic reasoning with geometric planning for tasks with multiple subtasks as multi-modal planning [49], temporal logic [9], or hierarchical or bridged representations and interfaces [15], [46], [113]. In many cases, these task planners can take as input estimates of solution quality for prospective geometric groundings of symbolic subtasks. The utility model over candidate paths maintained by LEMUR could serve as an effective feedback mechanism for these higher-level task planners.

Many-to-Many Objectives. As a low-level motion planner, LEMUR has been formulated for solving single-pair ("one-to-one") motion planning queries. This can be trivially extended to address "one-to-any" or "any-to-any" queries, where the start or goal configurations are extended to multiple candidates expressed as start or goal regions. This can be captured naturally in a shortest-path framework by extending the graph representation with virtual vertices and edges connecting to all candidates starts and goals. However, a higher-level multi-step task planner is often incentivized to explore many candidate intermediate configurations, and cultivate prospective paths through them. One method for expressing these "many-to-many" queries is by modifying the objective given to the lower-level motion planner; we explored the comprehensive multi-root objective [24] to that end. It would be worthwhile to apply this objective to lazy roadmap motion planners such as LEMUR.

Edge Selectors for Manipulation Graphs. We also highlight that task planners can be viewed as searching the *manipulation graph* [109] for a feasible solution to the task. Each edge in the manipulation graph comprises a transit or transfer path (or, in more general problem domains, other non-prehensile actions). The computationally intensive part of solving these graphs is commonly finding feasible geometric groundings for each edge of this manipulation graph. We contend that this problem could be solved directly using the LazySP algorithm (Chapter 3. In this formulation, each edge of the task graph would itself entail an invocation of LEMUR – but each invocation would share the same underlying roadmap managed as described in Chapter 6. Edge selectors could then be employed to allocate motion planning resources only on edges in the manipulation graph most likely to contribute to a short solution to the task.

7.3 *Lessons Learned*

A number of issues and questions arose during the design and implementation of the algorithms in this thesis that may be useful to researchers that may build on this work.

The Importance of Search. My initial examination of motion planning begun with a clear perceived separation between search-based and sampling-based approaches. By focusing on roadmap methods, I envisioned that the question of how to construct roadmaps over the configuration space would dominate my investigation. But my exploration of lazy roadmaps resulted in two personal realizations. First, search-based and sampling-based methods are more similar than they are usually treated (a connection first identified by LaValle et. al. [73]). And second, the question of how best to search over roadmaps is far from a settled question. This motivated by study of both lazy and dynamic search problems, and lead directly to the development of LazySP and IBiD, respectively. Indeed, the interaction between edge selection and search is rich with opportunities for future work.

The Importance of Performance Metrics. Progress in robotics depends on a delicate interplay between new research ideas and motivating real-world applications. While novel ideas can often have a direct benefit on a wide variety of applications, the potential for a particular applications or examples to drive new research is equally powerful. In particular, a performance shortfall of an existing algorithm often serves not only as a motivation for a new idea, but as a springboard for developing that idea in the first place. During my thesis, I found that committing to a set of benchmark problems, and being guided by rigorous performance metrics such as path quality and planning time, allowed me to identify fruitful research areas. It also lead to many other implementation improvements – such as the preallocated (or "baked") data structures feature in our implementation of FCL – which improved performance across all motion planners in our arsenal.

Fair Comparisons with Non-Determinism. If performance metrics are to be trusted, it is of paramount importance that planner comparisons be fair and repeatable, so that only the factor in question (e.g. the search or sampling strategy) is varied between experiments. Not only must all outside factors be held constant, such as collision checker parameters, resolutions, samplers, roadmaps, etc., but I found that the most robust way to handle planners which depend on randomness is to treat the random seed as an additional parameter, and to commit to ranges of these seeds for testing. Not only does this allow for a fairer comparison between planners, but it allows for a direct comparison between revisions of a particular planner's implementation, to more easily measure improvement and spot regressions. In research we often don't have the bandwidth to implement full unit test coverage, so such ensemble testing must suffice to maintain the quality of our implementations.

Implementation Details. Writing a performant motion planning implementation, especially one which relies on cached data structures, requires focus on inner loops, profiling, and iterative development. Care must be taken so that the roadmap data structure can be loaded into memory quickly during planner initialization, and so that the roadmap can be densified efficiently without reallocating large blocks of the data structure. The core roadmap generation and search code should be well-tested – the implementation used in this thesis was developed over several years, and was used for everything from figure generation to experiments. I also found it useful to rely on common data formats, such as GraphML [12], to allow interoperability with other software packages.

7.4 Concluding Remarks

This dissertation explores the integration of two ideas for efficient motion planning – *lazy* and *utility-guided* evaluation. These key ideas are complementary; while lazy pathfinding serves to decouple the process of discovering our options from determining their validity, utility-guided planning gives us the means to select intelligently between these options for the most promising.

We exploit roadmaps as a way to discretize our continuous planning problem into a pathfinding problem on a graph. But importantly, we do not lose sight of our motion planning domain: evaluating edge weights is expensive. Not only does this motivate our approach of lazily determining edge weights, but it opens the door to intelligent strategies for allocating our precious planning resources in the right places. We proposed edge selectors as a mechanism for doing this over the edges of a graph (and we have had fruitful discussions with researchers from other domains such as satellite trajectory planning which share our problem's cost structure).

While selectors can capture the problem of resource allocation over the discrete space, there is also the question of allocating planning resources between different levels of densification. LEMUR addresses this question via batching – switching to a progressively better discrete approximation to the continuous problem until a path is found.

However, we discussed ways in Section 7.2.2 to represent this more naturally during the course of the search itself. This also opens the door to specialized motion planning hardware (e.g. GPUs and FPGAs) which can maintain roadmaps efficiently and in parallel (e.g. [87]).

Importantly, performing lazy search over roadmaps enables the use of grounded heuristic estimates over concrete prospective motion path segments. This directly enables our use of utility functions to guide the planner's evaluation, which is especially relevant for autonomous robots performing human-scale tasks – performing tasks with and around people requires robots that are efficient both in computation and in action, since people are unaccommodating both of long planning pauses and of wild and inefficient motions. The expressiveness of our utility function approach, which can take advantage of any domain-specific planning heuristics available, opens the doors to many opportunities for getting better performance out of autonomous robots performing helpful recurring tasks in our day-to-day lives.

A

Appendix: LazySP Proofs and Timing Results

A.1 LazySP Proofs

A.1.1 Theoretical Properties of LazySP

Proof of Theorem 2 Let p^* be an optimal path w.r.t. w, with $\ell^* = \mathrm{len}(p^*, w)$. Since $w_{\mathrm{est}}(e) \leq \epsilon\, w(e)$ and $\epsilon \geq 1$, it follows that regardless of which edges are stored in W_{eval}, $w_{\mathrm{lazy}}(e) \leq \epsilon\, w(e)$, and therefore $\mathrm{len}(p^*, w_{\mathrm{lazy}}) \leq \epsilon\, \ell^*$. Now, since the inner ShortestPath algorithm terminated with p_{ret}, we know that $\mathrm{len}(p_{\mathrm{ret}}, w_{\mathrm{lazy}}) \leq \mathrm{len}(p^*, w_{\mathrm{lazy}})$. Further, since the algorithm terminated with p_{ret}, each edge on p_{ret} has been evaluated; therefore, $\mathrm{len}(p_{\mathrm{ret}}, w) = \mathrm{len}(p_{\mathrm{ret}}, w_{\mathrm{lazy}})$. Therefore, $\mathrm{len}(p_{\mathrm{ret}}, w) \leq \epsilon\, \ell^*$. ☐

Proof of Theorem 1 In this case, the algorithm will evaluate at least unevaluated edge at each iteration. Since there are a finite number of edges, eventually the algorithm will terminate. ☐

A.1.2 A* Equivalence

Proof of Invariant 1 If v is discovered, then it must either be on OPEN or CLOSED. v can be on CLOSED only after it has been expanded, in which case s would be discovered (which it is not). Therefore, v must be on OPEN. ☐

Proof of Invariant 2 Clearly the invariant holds at the beginning of the algorithm, with only v_{start} on OPEN. If the invariant were to no longer hold after some iteration, then there must exist some pair of discovered vertices v and v' with v on CLOSED and $g[v] + w(v, v') < g[v']$. Since v is on CLOSED, it must have been expanded at some previous iteration, immediately after which the inequality could not have held because $g[v']$ is updated upon expansion of v. Therefore, the inequality must have newly held after some intervening iteration, with v remaining on CLOSED. Since the values g are monotonically non-increasing and w is fixed, this implies that $g[v]$ must have been updated (lower). However, if this had happened, then v would have been removed from CLOSED and placed on OPEN. This contradiction implies that the invariant holds at every iteration. $\square$

Proof of Theorem 3 Consider path p^*_{lazy} with length ℓ^*_{lazy} yielding frontier vertex v_{frontier} via SelectExpand. Construct a vertex sequence s as follows. Initialize s with the vertices on p^*_{lazy} from v_{start} to v_{frontier}, inclusive. Let N be the number of consecutive vertices at the start of s for which $f(v) = \ell^*_{\text{lazy}}$ (Note that the first vertex on p^*_{lazy}, v_{start}, must have $f(v_{\text{start}}) = \ell^*_{\text{lazy}}$, so $N \geq 1$.) Remove from the start of s the first $N-1$ vertices. Note that at most the first vertex on s has $f(v) = \ell^*_{\text{lazy}}$, and the last vertex on s must be v_{frontier}.

Now we show that each vertex in this sequence s, considered by A* in turn, exists on OPEN with minimal f-value. Iteratively consider the following procedure for sequence s. Throughout, we know that there must not be any vertex with $f(v) < \ell^*_{\text{lazy}}$, that would imply that a different path through v_b shorter than ℓ^*_{lazy} exists, in which case p^*_{lazy} could not have been chosen.

If the sequence has length > 1, then consider the first two vertices on s, v_a and v_b. By construction, $f(v_a) = \ell^*_{\text{lazy}}$ and $f(v_b) \neq \ell^*_{\text{lazy}}$. In fact, from above we know that $f(v_b) > \ell^*_{\text{lazy}}$. Therefore, we have that $f(v_a) < f(v_b)$, therefore and $g[v_a] + w(v_a, v_b) < g[v_b]$. By Invariant 2, v_a must be on OPEN, and with $f(v_a) = \ell^*_{\text{lazy}}$, it can therefore be considered by A*. After it is expanded, $f(v_b) = \ell^*_{\text{lazy}}$, and we can repeat the above procedure with the sequence formed by removing the v_a from s.

If instead the sequence has length 1, then it must be exactly (v_{frontier}), with $f(v_{\text{frontier}}) = \ell^*_{\text{lazy}}$. Since the edge after $f(v_{\text{frontier}})$ is not evaluated, then by Invariant 1, v_{frontier} must be on OPEN, and will therefore be expanded next. $\square$

Proof of Theorem 4 Given that all vertices in $s_{\text{candidate}}$ besides the last are re-expansions, they can be expanded with no edge evaluations. Once the last vertex, v_{frontier}, is to be expanded by A*, suppose it has f-value ℓ.

First, we will show that there exists a path with length ℓ w.r.t. w_{lazy} wherein all edges before v_{frontier} have been evaluated, and the first edge after v_{frontier} has not. Let p_a be a shortest path from v_{start} to v_{frontier} consisting of only evaluated edges. The length of this p_a must be equal to $g[v_{\text{frontier}}]$; if it were not, there would be some previous vertex on p_a with lower f-value than v_{frontier}, which would necessarily have been expanded first. Let p_b be the a shortest path from v_{frontier} to v_{goal}. The length of p_b must be $h_{\text{lazy}}(v_{\text{frontier}})$ by definition. Therefore, the path (p_a, p_b) must have length ℓ, and since v_{frontier} is a new expansion, the first edge on p_b must be unevaluated.

Second, we will show that there does not exist any path shorter than ℓ w.r.t. w_{lazy}. Suppose p' were such a path, with length $\ell' < \ell$. Clearly, v_{start} would have f-value ℓ' (although it may not be on OPEN). Consider each pair of vertices (v_a, v_b) along p' in turn. In each case, if v_b were either undiscovered, or if $g[v_a] + w(v_a, v_b) < g[v_b]$, then v_a would be on OPEN (via Invariants 1 and 2, respectively) with $f(v_a) = \ell'$, and would therefore have been expanded before v_{frontier}. Otherwise, we know that $f(v_b) = \ell'$, and we can continue to the next pair on p'. □

A.1.3 LWA Equivalence*

Proof of Invariant 3 Clearly the invariant holds at the beginning of the algorithm with only $g[v_{\text{start}}] = 0$, since the inequality holds only for the out-edges of v_{start}, with v_{start} on Q_v. Consider each subsequent iteration. If a vertex v is popped from Q_v, then this may invalidate the invariant for all successors of v; however, since all out-edges are immediately added to Q_e, the invariant must hold. Consider instead if an edge (v, v') which satisfies the inequality is popped from Q_e. Due to the inequality, we know that $g[v']$ will be recalculated as $g[v'] = g[v] + w(v, v')$, so that the inequality is no longer satisfied for edge (v, v'). However, reducing the value $g[v']$ may introduce satisfied inequalities across subsequent out-edges of v', but since v' is added to Q_v, the invariant continues to hold. □

Proof of Theorem 5 In the first component of the equivalence, we will show that for any path p minimizing w_{lazy} allowable to LazySP-Forward, with (v_a, v_b) the first unevaluated edge on p, there exists a sequence of vertices and edges on Q_v and Q_e allowable to LWA* such that edge (v_a, v_b) is the first to be newly evaluated. Let the length of p w.r.t. w_{lazy} be ℓ.'

We will first show that no vertex on Q_v or edge on Q_e can have $f(\cdot) < \ell$. Suppose such a vertex v, or edge e with source vertex v, exists. Then $g[v] + h(v) < \ell$, and there must be some path p' consisting of an evaluated segment from v_{start} to v of length $g[v]$, followed a segment from v to v_{goal} of length $h(v)$. But then this path should have been chosen by LazySP.

Next, we will show a procedure for generating an allowable sequence for LWA*. We will iteratively consider a sequence of path segments, starting with the segment from v_{start} to v_a, and becoming progressively shorter at each iteration by removing the first vertex and edge on the path. We will show that the first vertex on each segment v_f has $g[v_f] = \ell - h(v_f)$. By definition, this is true of the first such segment, since $g[v_{\text{start}}] = 0$. For each but the last such segment, consider the first edge, (v_f, v_s). If v_s has the correct $g[\cdot]$, we can continue to the next segment immediately. Otherwise, either v_f is on Q_v or (v_f, v_s) is on Q_e by Invariant 3. If the former is true, then v_f can be popped from Q_v with $f = \ell$, thereby adding (v_f, v_s) to Q_e. Then, (v_f, v_s) can be popped from Q_e with $f = \ell$, resulting in $g[v_s] = \ell - h(v_s)$. We can then move on to the next segment.

At the end of this process, we have the trivial segment (v_a), with $g[v_a] = \ell - h(v_a)$. If v_a is on Q_v, then pop it (with $f(v_a) = \ell$), placing e_{ab} on Q_e; otherwise, since e_{ab} is unevaluated, it must already be on Q_e. Since $f(e_{ab}) = \ell$, we can pop and evaluate it. □

Proof of Theorem 6 Given that all vertices in $s_{\text{candidate}}$ entail no edge evaluations, and all edges therein are re-expansions, they can be considered with no edge evaluations. Once the last edge e_{xy} is to be expanded by LWA*, suppose it has f-value ℓ.

First, we will show that there exists a path with length ℓ w.r.t. w_{lazy} which traverses unevaluated edge e_{xy} wherein all edges before v_x have been evaluated. Let p_x be a shortest path segment from v_{start} to v_x consisting of only evaluated edges. The length of p_x must be equal to $g[v_x]$; if it were not, there would be some previous vertex on p_x with lower f-value than v_x, which would necessarily have been expanded first. Let p_y be the a shortest path from v_y to v_{goal}. The length of p_y must be $h_{\text{lazy}}(v_y)$ by definition. Therefore, the path (p_x, e_{xy}, p_y) must have length ℓ.

Second, we will show that there does not exist any path shorter than ℓ w.r.t. w_{lazy}. Suppose p' were such a path, with length $\ell' < \ell$, and with first unevaluated edge e'_{xy}. Clearly, v_{start} has $g[v_{\text{start}}] = \ell' - h(v_{\text{start}})$. Consider each evaluated edge e'_{ab} along p' in turn. In each case, if $g[v'_b] \neq \ell' - h(v'_b)$, then either v'_a or e'_{ab} would be on Q_v or Q_e with $f(\cdot) = \ell'$, and would therefore be expanded before e_{xy}. Therefore, e'_{xy} would then be popped from Q_e with $f(e'_{xy}) = \ell'$, and it would have been evaluated before e_{xy} with $f(e_{xy}) = \ell$. □

A.2 LazySP Timing Results

We include an accounting of the cumulative computation time taken by each component of LazySP for each of the seven selectors for each problem class (Figure A.1).

	Expand	Forward	Reverse	Alternate	Bisect	WeightSamp	Partition
PartConn							
total (ms)	**1.22** ± 0.04	1.96 ± 0.06	1.86 ± 0.06	**1.20** ±0.03	2.41 ±0.06	4807.19 ±135.22	15.81 ± 0.16
sel-init (ms)	–	–	–	–	–	–	12.49 ± 0.11
online (ms)	**1.22** ± 0.04	1.96 ± 0.06	1.86 ± 0.06	**1.20** ±0.03	2.41 ±0.06	4807.19 ±135.22	3.32 ± 0.10
search (ms)	0.48 ± 0.01	1.12 ± 0.03	1.05 ± 0.03	0.68 ±0.02	1.38 ±0.04	0.70 ± 0.02	0.68 ± 0.02
sel (ms)	0.02 ± 0.00	0.01 ± 0.00	0.01 ± 0.00	0.01 ±0.00	0.03 ±0.00	4805.64 ±135.18	2.07 ± 0.06
eval (ms)	–	–	–	–	–	–	–
eval (edges)	87.10 ± 2.39	35.86 ± 1.04	34.84 ± 1.04	22.23 ±0.60	44.81 ±1.11	**20.66** ± 0.57	**20.39** ± 0.56
UnitSquare							
total (ms)	**0.91** ± 0.03	1.47 ± 0.06	1.49 ± 0.06	**0.94** ±0.03	1.71 ±0.04	3864.95 ±117.66	15.13 ± 0.14
sel-init (ms)	–	–	–	–	–	–	13.41 ± 0.12
online (ms)	**0.91** ± 0.03	1.47 ± 0.06	1.49 ± 0.06	**0.94** ±0.03	1.71 ±0.04	3864.95 ±117.66	1.72 ± 0.06
search (ms)	0.35 ± 0.01	0.79 ± 0.03	0.82 ± 0.03	0.51 ±0.02	0.92 ±0.02	0.75 ± 0.02	0.45 ± 0.01
sel (ms)	0.01 ± 0.00	0.01 ± 0.00	0.01 ± 0.00	0.01 ±0.00	0.02 ±0.00	3863.49 ±117.62	0.87 ± 0.03
eval (ms)	–	–	–	–	–	–	–
eval (edges)	69.21 ± 2.55	27.29 ± 1.03	27.69 ± 1.02	17.82 ±0.60	32.62 ±0.72	15.58 ± 0.47	**14.08** ± 0.46
ArmPlan (avg)							
total (s)	269.82 ± 17.95	**5.90** ± 0.46	8.22 ± 0.53	**5.96** ±0.31	7.34 ±0.43	3402.21 ±172.20	496.57 ± 5.53
sel-init (s)	–	–	–	–	–	–	490.77 ± 5.51
online (s)	269.82 ± 17.95	**5.90** ± 0.46	8.22 ± 0.53	**5.96** ±0.31	7.34 ±0.43	3402.21 ±172.20	**5.80** ± 0.28
search (s)	0.02 ± 0.00	0.02 ± 0.00	0.02 ± 0.00	0.02 ±0.00	0.02 ±0.00	0.02 ± 0.00	0.04 ± 0.00
sel (s)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ±0.00	0.00 ±0.00	3392.76 ±171.74	1.54 ± 0.09
eval (s)	269.78 ± 17.95	5.87 ± 0.45	8.20 ± 0.52	5.94 ±0.31	7.31 ±0.43	9.39 ± 0.57	4.21 ± 0.22
eval (edges)	949.05 ± 63.46	63.62 ± 4.15	74.94 ± 5.07	55.48 ±2.95	68.01 ±3.86	56.93 ± 3.37	**48.07** ± 2.44
ArmPlan1							
total (s)	109.09 ± 14.15	**4.81** ± 0.49	14.81 ± 1.45	7.03 ±0.63	7.91 ±0.70	3375.35 ±319.81	496.74 ± 8.22
sel-init (s)	–	–	–	–	–	–	489.49 ± 8.18
online (s)	109.09 ± 14.15	**4.81** ± 0.49	14.81 ± 1.45	7.03 ±0.63	7.91 ±0.70	3375.35 ±319.81	7.25 ± 0.66
search (s)	0.02 ± 0.00	0.02 ± 0.00	0.03 ± 0.00	0.02 ±0.00	0.02 ±0.00	0.02 ± 0.00	0.04 ± 0.00
sel (s)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ±0.00	0.00 ±0.00	3358.82 ±318.17	1.61 ± 0.16
eval (s)	109.07 ± 14.15	4.78 ± 0.49	14.77 ± 1.44	7.01 ±0.63	7.88 ±0.70	16.47 ± 1.68	5.59 ± 0.51
eval (edges)	344.74 ± 39.63	**49.72** ± 4.25	95.58 ± 9.67	59.44 ±5.06	58.90 ±4.74	73.72 ± 7.63	**50.66** ± 4.43
ArmPlan2							
total (s)	166.19 ± 9.29	**3.27** ± 0.25	7.36 ± 0.69	5.95 ±0.52	5.63 ±0.45	4758.04 ±407.56	495.21 ±12.65
sel-init (s)	- -	–	–	–	–	–	489.22 ±12.64
online (s)	166.19 ± 9.29	**3.27** ± 0.25	7.36 ± 0.69	5.95 ±0.52	5.63 ±0.45	4758.04 ±407.56	5.99 ± 0.48
search (s)	0.01 ± 0.00	0.01 ± 0.00	0.02 ± 0.00	0.01 ±0.00	0.01 ±0.00	0.02 ± 0.00	0.03 ± 0.00
sel (s)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ±0.00	0.00 ±0.00	4750.16 ±406.98	2.03 ± 0.22
eval (s)	166.17 ± 9.28	**3.26** ± 0.25	7.34 ± 0.69	5.93 ±0.52	5.61 ±0.45	7.82 ± 0.61	3.91 ± 0.27
eval (edges)	657.02 ± 29.24	**62.24** ± 6.12	98.54 ±10.89	69.96 ±6.98	75.88 ±7.47	**66.24** ± 6.36	**62.16** ± 6.10
ArmPlan3							
total (s)	534.16 ± 55.64	9.61 ± 1.33	**2.50** ± 0.23	4.91 ±0.56	8.47 ±0.99	2073.23 ±198.75	497.76 ±10.27
sel-init (s)	–	–	–	–	–	–	493.59 ±10.21
online (s)	534.16 ± 55.64	9.61 ± 1.33	**2.50** ± 0.23	4.91 ±0.56	8.47 ±0.99	2073.23 ±198.75	4.17 ± 0.43
search (s)	0.02 ± 0.01	0.02 ± 0.00	0.02 ± 0.00	0.02 ±0.00	0.02 ±0.00	0.03 ± 0.01	0.04 ± 0.00
sel (s)	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ±0.00	0.00 ±0.00	2069.29 ±198.53	0.98 ± 0.13
eval (s)	534.10 ± 55.63	9.58 ± 1.33	2.48 ± 0.23	4.89 ±0.56	8.44 ±0.99	3.90 ± 0.31	3.15 ± 0.32
eval (edges)	1845.38 ±195.57	78.90 ±10.36	**30.70** ± 3.62	37.04 ±4.59	69.26 ±7.97	**30.82** ± 3.60	**31.38** ± 3.80

Figure A.1: Detailed timing results for each selector. The actual edge weights for the illustrative PartConn and UnitSquare problems were pre-computed, and therefore their timings are not included. The Partition selector requires initialization of the Z-values (3.7) for the graph using only the estimated edge weights. Since this is not particular to either the actual edge weights (e.g. from the obstacle distribution) or the start/goal vertices from a particular instance, this initialization (*sel-init*) is considered separately. The online running time (*online*) is broken into LazySP's three primary steps: the inner search (*search*), invoking the edge selector (*sel*), and evaluating edges (*eval*). We also show the number of edges evaluated.

B

Appendix: Incrementally Calculating Partition Functions on Graphs

Problem Definition. Consider a directed graph $G = (V, E)$ endowed with an edge weight function $w : E \to \mathbb{R}$. Between any two vertices $x, y \in V$, there exist a (potentially infinite) set of paths P_{xy}, with the length of any path given by $\operatorname{len} : P_{xy} \to \mathbb{R}$ as in (3.1). Between each such pair of vertices x, y, one can compute the *partition function* denoted Z_{xy}, defined as a sum over all possible paths as follows:

Note that we do not restrict consideration to only simple paths on G.

$$Z_{xy} = \sum_{p_{xy} \in P_{xy}} \exp(-\beta \operatorname{len}(p_{xy})), \tag{B.1}$$

with β a fixed real-valued parameter (commonly called an inverse temperature in statistical mechanics). We wish to compute and maintain this partition function value between all pairs of vertices x and y on G.

An Incremental Approach. We propose an incremental method for calculating Z between all pairs on a graph. We will proceed by initializing the values Z_{xy} for each pair on a graph with no edges, and then show how we can update these values as edges are added or removed.

First, consider the graph with no edges $G_0 = (V, \emptyset)$. On this graph, there exists no paths between each pair of vertices x, y with $x \neq y$. On the other hand, for $x = y$ there exists exactly one path containing no edges (and therefore with length 0). Thus, we can initialize our values Z_{xy} with:

$$Z_{xy} = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{otherwise.} \end{cases} \tag{B.2}$$

Adding Edges. We can next establish the inductive step. Suppose we compare two directed graphs G and G', where $V' = V$ and $E' =$

$E \cup \{e_{ab}\}$, and with new edge $e_{ab} : a \rightarrow b$ having weight w_{ab}. For arbitrary vertices x and y, we can write Z'_{xy} as follows:

$$Z'_{xy} = S^{(0)} + S^{(1)} + S^{(2)} + \ldots \tag{B.3}$$

where $S^{(k)}$ is the sum from (B.1) over only the subset of paths that traverse e_{ab} exactly k times. We note immediately that $S^{(0)} = Z_{xy}$.

Next, we consider the sum $S^{(1)}$, that is, the sum over all paths which use the new edge e_{ab} exactly once. We note that each path which contributes to this sum consists of three segments: (a) a path segment from x to a, followed by (b) the new edge e_{ab}, followed by (c) a path segment from b to y. Since the sum in question consists exactly of the sum over all unique paths which follow this template, we see that we can write $S^{(1)}$ as follows:

$$S^{(1)} = \sum_{p_{xa}} \sum_{p_{by}} \exp\left(-\beta \left[\operatorname{len}(p_{xa}) + w_{ab} + \operatorname{len}(p_{by})\right]\right) \tag{B.4}$$

Note that both $p_{xa} \in P_{xa}$ and $p_{by} \in P_{by}$ are over all paths on the original graph G. We can then rewrite this simply as:

$$S^{(1)} = Z_{xa} \, \exp(-\beta \, w_{ab}) \, Z_{by}. \tag{B.5}$$

We can likewise write $S^{(2)}$ as:

$$S^{(2)} = Z_{xa} \, \exp(-\beta \, w_{ab}) \, Z_{ba} \, \exp(-\beta \, w_{ab}) \, Z_{by}. \tag{B.6}$$

That is, this sum consists of paths which arrive from x to a through G, traverse the edge e_{ab} once, then traverse the original G in some way from b back to a, traverse the edge e_{ab} a second time, and then proceed from b to y. This decomposition allows us to rewrite the entire sum from (B.3) as:

$$Z'_{xy} = Z_{xy} + Z_{xa} \, \exp(-\beta \, w_{ab}) \, Z_{by} \sum_{k=0}^{\infty} \left[Z_{ba} \, \exp(-\beta \, w_{ab})\right]^k . \tag{B.7}$$

As this is a geometric series, it will converge as long as $Z_{ba} < \exp(\beta \, w_{ab})$. In this case, we have:

$$Z'_{xy} = Z_{xy} + \frac{Z_{xa} \, Z_{by}}{\exp(\beta \, w_{ab}) - Z_{ba}}. \tag{B.8}$$

This allows us to accommodate arbitrary edge additions while maintaining the correct values Z_{xy} over all pairs of vertices on G.

Note that the addition of an undirected edge between a and b must entail two successive applications of (B.8).

Removing Edges. We can establish a similar result in the case that an existing edge is removed from the graph. Consider the case that you have a graph G', with partition function values Z' between all pairs of vertices. You then remove some existing edge e_{ab} with weight w_{ab}, yielding new graph G. What can we say about the new values Z?

We proceed by applying (B.8) to the edge to be removed e_{ab}:

$$Z'_{ba} - Z_{ba} - \frac{Z_{ba}^2}{\exp(\beta w_{ab}) - Z_{ba}} = 0 \tag{B.9}$$

$$\left[\exp(\beta w_{ab}) - Z_{ba}\right] Z'_{ba} - \exp(\beta w_{ab}) Z_{ba} = 0 \tag{B.10}$$

$$\left[\exp(\beta w_{ab}) - Z_{ba}\right] \left[\exp(\beta w_{ab}) + Z'_{ba}\right] = \exp(2\beta w_{ab}) \tag{B.11}$$

$$\exp(\beta w_{ab}) - Z_{ba} = \frac{\exp(2\beta w_{ab})}{\exp(\beta w_{ab}) + Z'_{ba}} \tag{B.12}$$

We can then use (B.12) to express Z_{xa} and Z_{by} in terms of Z'_{xa} and Z'_{by}, respectively:

$$Z_{xa} + \frac{Z_{xa}\, Z_{ba}}{\exp(\beta w_{ab}) - Z_{ba}} = Z'_{xa} \longrightarrow Z_{xa} = \frac{\exp(\beta w_{ab})}{\exp(\beta w_{ab}) + Z'_{ba}} Z'_{xa} \tag{B.13}$$

$$Z_{by} + \frac{Z_{ba}\, Z_{by}}{\exp(\beta w_{ab}) - Z_{ba}} = Z'_{by} \longrightarrow Z_{by} = \frac{\exp(\beta w_{ab})}{\exp(\beta w_{ab}) + Z'_{ba}} Z'_{by} \tag{B.14}$$

Lastly, we can combine (B.8) and (B.12) to express the desired value Z_{xy} only in terms of values on G':

$$Z_{xy} = Z'_{xy} - \frac{\left[\frac{\exp(\beta w_{ab})}{\exp(\beta w_{ab}) + Z'_{ba}} Z'_{xa}\right] \left[\frac{\exp(\beta w_{ab})}{\exp(\beta w_{ab}) + Z'_{ba}} Z'_{by}\right]}{\left[\frac{\exp(2\beta w_{ab})}{\exp(\beta w_{ab}) + Z'_{ba}}\right]} \tag{B.15}$$

$$Z_{xy} = Z'_{xy} - \frac{Z'_{xa} Z'_{by}}{\exp(\beta w_{ab}) + Z'_{ba}} \tag{B.16}$$

This allows us to accommodate edge deletions.

C

Appendix: IBiD Proofs

Proof of Theorem 7 Consider any vertex x. If $d^*(x) = \infty$, then by (4.3a) we must have that $d(x) = \infty$. Otherwise, by (4.1), there exists a path p^* of length $d^*(x)$; consider this path. The first vertex on p^* is s with $d^*(s) = 0$, and by (4.3), $d(s) = d^*(s)$. For each edge e_{uv} on p^* with $d(u) = d^*(u)$, we will show that $d(v) = d^*(v)$. By definition of the shortest path, $d^*(u) + w(e_{uv}) = d^*(v)$. Therefore $d(u) + w(e_{uv}) = d^*(v)$, and by (4.3d), we have $d^*(v) \geq d(v)$, and by (4.3a) we have $d^*(v) = d(v)$. By induction along the path p^*, we have that $d(x) = d^*(x)$. $\square$

Proof of Theorem 8 The proof proceeds as follows. First, if $d' = \infty$, then no edges can be in tension, and so $d = d^*$ everywhere as shown in Section 4.2.2. Otherwise, suppose that $d(x) \neq d^*(x)$ for some vertex x with $d(x) \leq d'$. By (4.3), it would have to be that $d^*(x) < d(x)$. Consider a true shortest path p from s to x; by (4.3) such a path exists and has finite length $d^*(x)$. By (4.3), we have that $d^*(s) = d(s) = 0$ (and so s and x must be distinct). Let e_{uv} be the first edge along p such that $d^*(u) = d(u)$ but $d^*(v) < d(v)$. Since p is a shortest path, edge e_{uv} must therefore be in tension. Since $w \geq 0$, it must be that $d^*(u) \leq d^*(x)$; further, since $d^*(u) = d(u)$, $d^*(x) < d(x)$, and $d(x) \leq d'$ it must be that $d(u) < d'$. But then edge e_{uv} is in tension with lower $d(u)$! This contradiction implies that every vertex x with $d(x) \leq d'$ must have $d(x) = d^*(x)$. $\square$

Proof of Theorem 9 For any vertex x with $d(x) \leq k$, consider the following construction of a path from s to x. Initialize the path $p \leftarrow \{x\}$; note that by Theorem 8, the first vertex v on p has $d(v) = d^*(v)$.

At each iteration, terminate construction if $v = s$. Otherwise, let u^* be the predecessor of v which minimizes $d(u) + w(e_{uv})$, and prepend u^* to p. By (4.3c), it follows that $d(u^*) + w(e^*_{uv}) \leq d(v)$, and since $d(v) = d^*(v)$ and $d^*(v) \leq d^*(u) + w(e^*_{uv})$ by definition of the distance function, it follows that $d(u^*) \leq d^*(u^*)$. In combination with (4.3a), this implies $d(u^*) = d^*(u^*)$, and we can iterate.

The result of this construction is a path p from s to x on which all vertices have $d(v) = d^*(v)$. Therefore p is a shortest path of length $d^*(x)$. $\square$

Proof of Theorem 10 Note first that since $d_s(u) \leq D_s$, by Theorem 8, $d_s(u) = d_s^*(u)$, and so there exists a path from s to u of length $d_s(u)$. Similarly, there exists a path from v to t of length $d_t(v)$, and as a result, there exists a path through e_{uv}^* with length $\ell_e(e_{uv}^*)$.

We will prove that this constitutes a shortest path by contradiction. Suppose that a different shortest path p' exists with $\text{len}(p') < \ell_e(e_{uv}^*)$. Then it must also be that $\text{len}(p') < D_s + D_t$. Note that since $s \neq t$, p' contains at least one edge. We will consider two cases.

First, consider the case where $\text{len}(p') < D_s$, so that $d_s^*(t) < D_s$. In this case, the last edge e'_{ut} on p' has $d_s^*(u') \leq d_s^*(t) < D_s$; by Theorem 8, u' therefore has $d_s(u') = d_s^*(u')$. In addition, since $D_t > 0$, t must be t-consistent with $d_t(t) = 0$. Therefore, it follows that $d_s^*(u') + w(e'_{ut}) < d_s(u') + w(e'_{ut})$, which contradicts the supposition.

In the second case with $D_s \leq \text{len}(p')$, identify on p' the edge e'_{uv} adjoining the vertices u', v' such that $d_s^*(u') < D_s \leq d_s^*(v')$. (Since $D_s > 0$, this edge will exist.) Since $d_s^*(u') < D_s$, by Theorem 11, u' is therefore s-consistent with $d_s(u') = d_s^*(u')$. Consider our supposition that $d_s^*(u') + w(e'_{uv}) + d_t^*(v') < D_s + D_t$. Since $d_s^*(u') + w(e'_{uv}) = d_s^*(v')$ and $D_s \leq d_s^*(v')$, it follows that $d_t^*(v') < D_t$. Therefore, by Theorem 11, v' is t-consistent with $d_t(v') = d_t^*(v')$. As a consequence, the edge e'_{uv} must be in E_{conn}. Therefore, $d_s^*(u') + w(e'_{uv}) + d_t^*(v') < d_s(u') + w(e'_{uv}) + d_t(v')$, which is a contradiction. $\square$

Proof of Theorem 11 The proof relies upon a path construction from s to x described by Lemma 1. Lemma 2 then demonstrates that the length of the path is $d(x)$, and therefore that $d(x)$ can be no less than $d^*(x)$. Finally, Lemma 3 shows that $d(x)$ can be no greater than $d^*(x)$. As a result, the value $d(x)$ is correct, and the path constructed via Lemma 1 is a shortest path. $\square$

Lemma 1 *For any consistent vertex x with $d(x) \leq k_{min}$, there exists a path p from s to x in which each vertex is consistent and each edge e_{uv} satisfies $d(u) + w(e_{uv}) = d(v)$.*

Proof of Lemma 1 Construct the path p as follows. Initialize the path with the single vertex x. Iteratively consider the first vertex v on the path, which is known to be consistent. In the first case, if $v \neq s$, then there exists a predecessor vertex u and edge e_{uv} with $d(u) + w(e_{uv}) = r(v)$. Since $w > 0$ and $d(v) = r(v)$, we have $d(u) < d(v) \leq d(x)$. As a consequence, u is consistent; prepend to the path the vertex u and the edge e_{uv}, and iterate. In the second case, if $v = s$, then we finish our construction of p. Since the values

$d(u)$ decrease monotonically for all inserted vertices, this process will terminate with a path p beginning at s. $\square$

Lemma 2 *Any consistent vertex x with $d(x) \leq k_{min}$ has $d(x) \geq d^*(x)$.*

Proof of Lemma 2 This follows directly from Lemma 1. Since all vertices on the path are known consistent, we must have $d(s) = 0$. Further, since the d-values across each edge in p satisfy $d(u) + w(e_{uv}) = d(v)$, it follows that $d(x) = \sum_{e \in p} w(e)$. Therefore, a path exists from s to x of length $d(x)$, and so the true distance $d^*(x)$ must be upper-bounded by $d(x)$. $\square$

Lemma 3 *Any consistent vertex v with $d(x) \leq k_{min}$ has $d(x) \leq d^*(x)$.*

Proof of Lemma 3 We demonstrate that $d(x) \leq d^*(x)$ by contradiction. Suppose a vertex x exists for which $d^*(x) < d(x)$. Then there must exist a path p' from s to x of length $d^*(x)$, with $d^*(s) = 0$ and $d^*(u) + w(e_{uv}) = d^*(v)$ for each edge in p'; as a consequence, we must have $d^*(v) < d(x)$ for all vertices v on p'. By Lemma 1, we know that s is consistent, so $d(s) = 0$. We will show that walking along each edge e_{uv} on p' starting at s, if $d(u) \leq d^*(u)$, then $d(v) \leq d^*(v)$.

By definition, we have $d(u) + w(e_{uv}) \geq r(v)$, so that $d(u) - d^*(u) \geq r(v) - d^*(v)$. Therefore, it follows that $r(v) \leq d^*(v)$. Since $k(v) \leq d^*(v)$, it follows that v is consistent, so $d(v) \leq d^*(v)$. We can replicate this logic down the path. As a result, it follows that $d(x) \leq d^*(x)$. But this contradicts our supposition that $d^*(x) < d(x)$, and therefore such a vertex x cannot exist. $\square$

Proof of Theorem 12 We will prove this by contradiction. Suppose that a path p' exists with $\text{len}(p') < \min_{e \in E_{\text{conn}}} (d_s(u) + w(e_{uv}) + d_t(v))$. Then it must also be that $\text{len}(p') < K_s + K_t$. We will consider two cases.

First, consider the case where $K_s > \text{len}(p')$, so that $K_s > d_s^*(t)$. In this case, the last edge e'_{ut} on p' has $d_s^*(u') < d_s^*(t) < K_s$; by Theorem 11, u' is therefore s-consistent with $d_s(u') = d_s^*(u')$. In addition, since $K_t > 0$, t must be t-consistent with $d_t(t) = 0$. Therefore, it follows that $d_s^*(u') + w(e'_{ut}) < d_s(u') + w(e'_{ut})$, which contradicts the supposition.

In the second case with $K_s \leq \text{len}(p')$, identify on p' the edge e'_{uv} adjoining the vertices u', v' such that $d_s^*(u') < K_s \leq d_s^*(v')$. (Since $K_s > 0$, this edge will exist.) Since $d_s^*(u') < K_s$, by Theorem 11, u' is therefore s-consistent with $d_s(u') = d_s^*(u')$. Consider our supposition that $d_s^*(u') + w(e'_{uv}) + d_t^*(v') < K_s + K_t$. Since $d_s^*(u') + w(e'_{uv}) = d_s^*(v')$ and $K_s \leq d_s^*(v')$, it follows that $d_t^*(v') < K_t$. Therefore, by Theorem 11, v' is t-consistent with $d_t(v') = d_t^*(v')$. As a consequence, the edge e'_{uv} must be in E_{conn}. Therefore, $d^*_s(u') + w(e'_{uv}) + d^*_t(v') < d_s(u') + w(e'_{uv}) + d_t(v')$, which is a contradiction. $\square$

D

Appendix: LEMUR Results

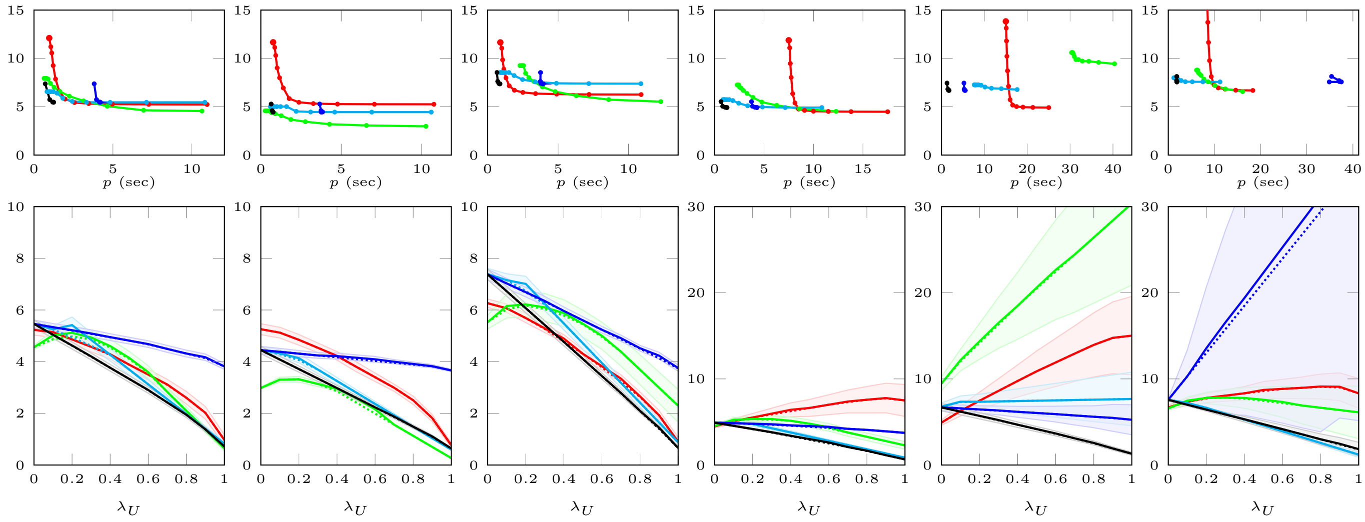


Figure D.1: Experimental results across six single-query motion planning instances for a 7-DOF robot arm. Top: expected planning cost p (in seconds) vs. execution cost x (in radians) for each parameterized planner for various values of their parameters. Bottom: the mean negative utility $-U = \lambda_U p + (1-\lambda_U)x$ (solid lines) measured for each planner (lower on plot is better) as the utility parameter λ_U is varied. The 95% confidence interval of the mean is also shown. Planners used the same parameter schedule across the problems as shown in Figure 5.6. The per-problem maximum achievable mean utilities (5.9) for each planner are shown as dotted lines. Planners are 🟥 RRT-Connect with shortcutting, 🟩 BIT*, 🟦 LEMUR (no roadmap cache), and ⬛ LEMUR (with roadmap cache).

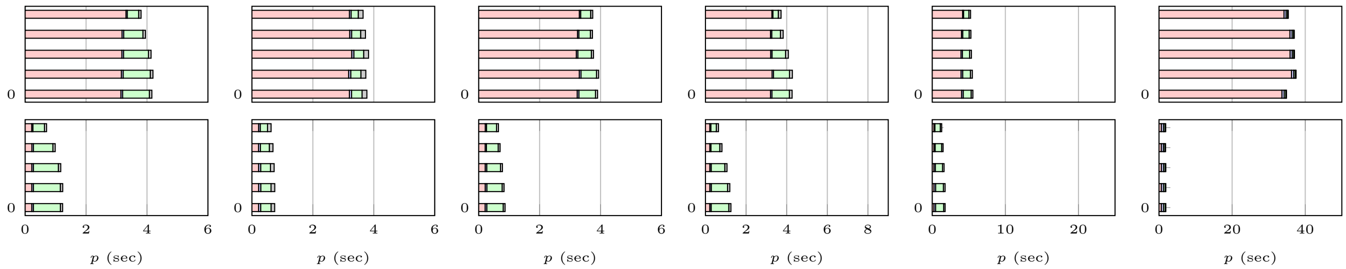


Figure D.2: Time breakdown of LEMUR for each of the six experimental problems across various values of λ_p. Cumulative time is shown performing the following operations: 🟥 roadmap generation, 🟦 graph search, 🟩 collision checking, and ⬜ unaccounted for. The bottom row shows the results of the algorithm over the same set of 50 roadmaps when reading the cached roadmap from disk to avoid online nearest-neighbor queries.

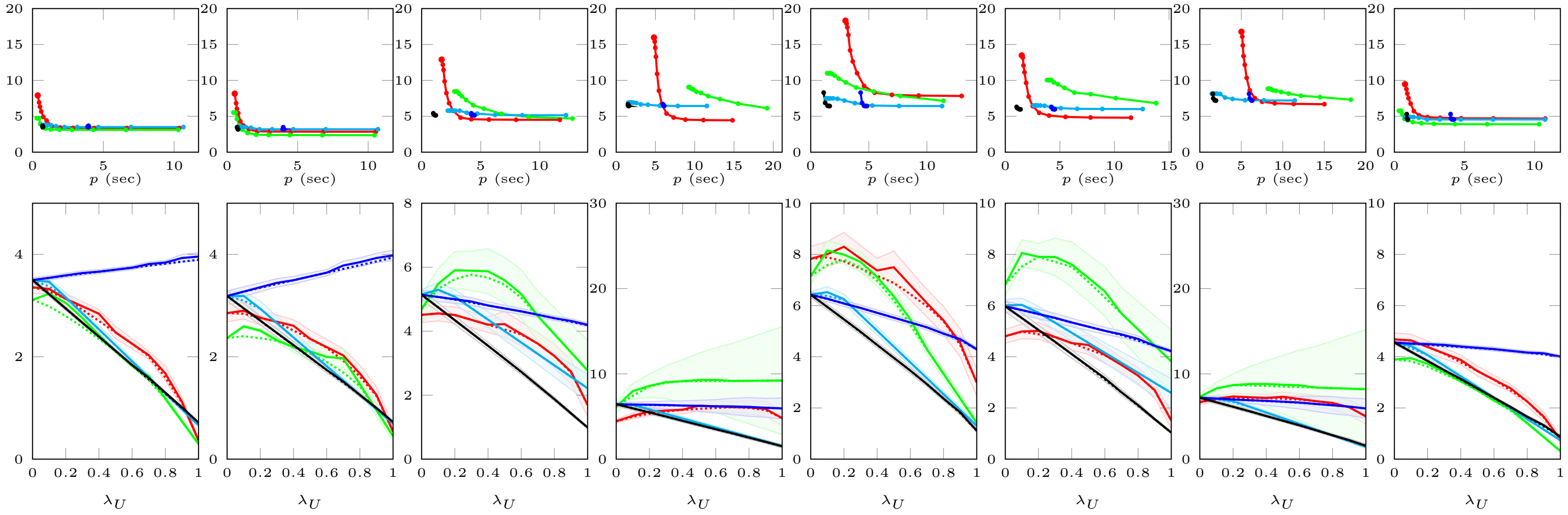


Figure D.3: Summary of p-vs-x and utility results for each step in the workcell problem.

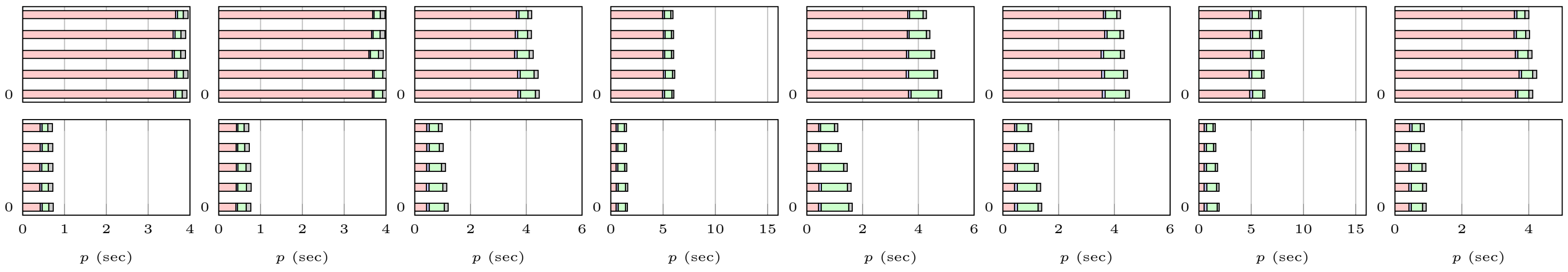


Figure D.4: Timing results for each step in the workcell problem.

E

Appendix: Family Motion Planning

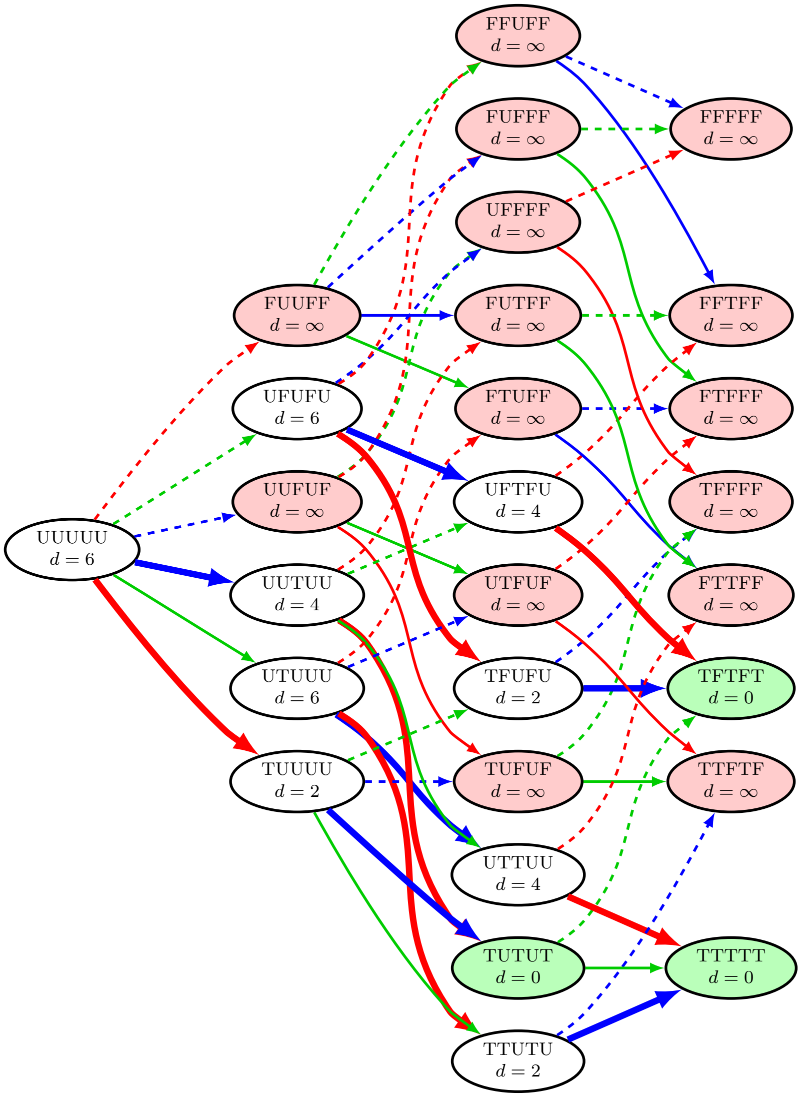

FFUFF
$d = \infty$
FUFFF
$d = \infty$
FFFFF
$d = \infty$
UFFFF
$d = \infty$
FUUFF
$d = \infty$
FUTFF
$d = \infty$
FFTFF
$d = \infty$
UFUFU
$d = 6$
FTUFF
$d = \infty$
FTFFF
$d = \infty$
UUFUF
$d = \infty$
UFTFU
$d = 4$
TFFFF
$d = \infty$
UUUUU
$d = 6$
UUTUU
$d = 4$
UTFUF
$d = \infty$
FTTFF
$d = \infty$
UTUUU
$d = 6$
TFUFU
$d = 2$
TFTFT
$d = 0$
TUUUU
$d = 2$
TUFUF
$d = \infty$
TTFTF
$d = \infty$
UTTUU
$d = 4$
TUTUT
$d = 0$
TTTTT
$d = 0$
TTUTU
$d = 2$

Figure E.1: Family belief graph for the 2D example problem from Figure 6.4 in Chapter 6. The underlying family consists of five sets: A, B, C, S_{12}, and S_{23}. Transitions between states are shown for indicators $\mathbf{1}_A$ (🟥), $\mathbf{1}_B$ (🟩), and $\mathbf{1}_C$ (🟦), with solid lines (⁄) showing transitions in which the indicator returns True, and dashed lines (⋰) showing transitions in which the indicator returns False. Also shown is the distance function and optimal policy for a query subset $S_u = S_{23}$, with green beliefs as goal states. Bolded edges (⁄) represent transitions on an optimal policy.

List of Figures

1.1 Outline of the dissertation document. Motivated by recurring manipulation tasks, we consider two questions: (a) how to optimize a motion planning objective over a C-space roadmap, and (b) which objective best captures the task's planning/execution tradeoff? 11
1.2 The three robotic platforms considered in this dissertation. 13

2.1 The original mover's problem entails finding a collision-free path for a geometric body amongst obstacles, or finding that no path exists. 15
2.2 The motion planning problem entails finding a continuous path among obstacles in an abstract configuration space. 16
2.3 Path cost model for the (feasible) motion planning problem. 16
2.4 Example dynamical path cost function for the optimal motion planning problem. Here, the arc length of ξ using the L_2 norm cost is used. 16
2.5 A stack of progressively densified roadmaps over a given configuration space $\mathcal{C}$. Each deeper roadmap constitutes a more accurate approximation to the true continuous problem space. 22
2.6 Examples of three roadmap types over the unit square: an axis-aligned lattice (left), random geometric graph (middle), and Halton graph with primes $(2, 3)$ (right). All three roadmaps have a connection radius of 0.3. The latter two roadmaps have 30 vertices. 23

3.1 While solving a shortest path query, a shortest path algorithm incurs computation cost from three sources: examining the structure of the graph G, evaluating the edge weight function w, and maintaining internal data structures. 25

3.2 Snapshots of the LazySP algorithm using each edge selector discussed in this chapter on the same obstacle roadmap graph problem, with start and goal. At top, the algorithms after evaluating five edges (evaluated edges labeled as valid or invalid). At middle, the final set of evaluated edges. At bottom, for each unique path considered from left to right, the number of edges on the path that are already evaluated, evaluated and valid, evaluated and invalid, and unevaluated. The total number of edges evaluated is noted in brackets. Note that the scale on the Expand plot has been adjusted because the selector evaluates many edges not on the candidate path at each iteration. 30

3.3 A* comparison between the static goal heuristic h_{est} (3.3) and the dynamic goal heuristic h_{lazy} (3.4) on a simple graph from start S to goal G. The values of both the edge weight estimate w_{est} and the true edge weight w (for evaluated edges) are shown. Using either goal heuristic, the A* algorithm first expands vertices S and Y, evaluating three edges in total and leaving X and G on OPEN. After finding that edge YG has $w = 3$, the dynamic heuristic value $h_{\text{lazy}}(\text{X})$ is updated from 2 to 4. While the A* using the static h_{est} would next expand X, the A* using the dynamic h_{lazy} would next expand G and terminate, having never evaluated edge XY. 31

3.4 Illustration of the equivalence between A* and LazySP-Expand. After evaluating the same set of edges, the next edges to be evaluated by each algorithm can both be expressed as a surjective mapping onto a common set of unexpanded frontier vertices. 32

3.5 Illustration of maximum edge probability selectors. A distribution over paths (usually conditioned on the known edge evaluations) induces on each edge e a Bernoulli distribution with parameter $p(e)$ giving the probability that it belongs to the path. The selector chooses the edge with the largest such probability. 35

3.6 The WeightSamp selector uses the path distribution induced by solving the shortest path problem on a distribution over possible edge weight functions $\mathcal{D}_w$. In this example, samples from $\mathcal{D}_w$ are computed by drawing samples from $\mathcal{D}_O$, the distribution of obstacles that are consistent with the known edge evaluations. 36

3.7 Examples of the Partition selector's $p(e)$ edge score function. With no known obstacles, a high β assigns near-unity score to only edges on the shortest path; as β decreases and more paths are considered, edges immediately adjacent to the roots score highest. Since all paths must pass through the narrow passage, edges within score highly. For a problem with two a-priori known obstacles (dark gray), the score first prioritizes evaluations between the two. Upon finding these edges are blocked, the next edges that are prioritized lie along the top of the world. 37

3.8 Visualization of the first of three articulated motion planning problems in which the HERB robot must move its right arm from the start configuration (pictured) to any of seven grasp configurations for a mug. Shown is the progression of the Alternate selector on one of the randomly generated roadmaps; approximately 2% of the 7D roadmap is shown in gray by projecting onto the space of end-effector positions. 39
3.9 Experimental results for the three problem classes across each of the seven selectors, E:Expand, F:Forward, R:Reverse, A:Alternate, B:Bisection, W:WeightSamp, and P:Partition. In addition to the summary table (a), the plots (b-d) show summary statistics for each problem class. The means and standard errors in (b-c) are across the 1000 and 900 problem instances, respectively. The means and standard errors in (d) are for the average across the three constituent problems for each of the 50 sampled roadmaps. A more detailed table of results is available in Appendix A. 40

4.1 Illustrations of the three focusing techniques considered on a spatial pathfinding problem. 43
4.2 A graph of the Northeast USA from the 9th DIMACS Implementation Challenge comprises 1,524,453 vertices and 3,868,020 directed edges. A shortest path problem from a start s in New Jersey to a destination t outside Boston will be used as an example. 45
4.3 The distance function from the start vertex. 45
4.4 Ordering problems. Consider the vertices $a \rightarrow b \rightarrow c$, with edges e_{ab} and e_{bc} both in tension; if e_{bc} is relaxed before e_{ab}, then e_{bc} will need to be relaxed a second time. 47
4.5 Tensioned edge trust region for $w \geq 0$. Contours are of the current estimate d. Currently tensioned edges are bold and dotted. 47
4.6 Dijkstra's algorithm computes the start distance function d^* to solve the example shortest path problem. Darker vertices have smaller d-values. The algorithm stops upon reaching the destination vertex t after expanding 1,290,820 vertices. 48
4.7 Problem case for pathfinding with distance functions in cases where invariant (4.3c) does not hold. Here, d satisfied a and c, with edge e_{sa} tensioned, and $d' = 0$. While the approximation d is *sound* at x via Theorem 8 (i.e. $d(x)$ is correct), the path reconstructed from t is not a shortest path. 48
4.8 The bidirectional Dijkatra's algorithm computes d_s around the start vertex and d_t around the destination vertex. Darker vertices have smaller d-values in their respective regions. The algorithm terminates after expanding a total of 1,178,200 vertices using distance to balance expansions. 49

4.9 Simple illustration of a problem case for terminating a bidirectional search. With a balanced distance criterion, c will be the first vertex expanded in both directions, but it does not lie on the shortest path. 49

4.10 Initial episode: 1,287,897 expansions. Subsequent episode: 391,122 expansions. 50

4.11 Initial search: 1,181,616 expansions. Replan: 262,422 expansions. 53

4.12 A* search. 532,880 expansions. 56

4.13 Illustration of behavior of bidirectional heuristic search on a shortest path problem reproduced from [67]. Vertices that are start-expanded are shown in blue, while those that are destination-expanded are shown in red. At left, the algorithm performs only start-side expansions, which approximates the behavior of a unidirectional algorithm. At right, the algorithm performs distance-balanced expansions between the two searches. The top row shows the behavior of the algorithm with no heuristic potential function. Start and destination heuristic functions are available, and their effect are shown at bottom. The unidirectional search uses the destination heuristic h_t as its potential function, while the bidirectional search uses the averaged potential function (4.11). 59

4.14 Bidirectional A* search. 515,588 expansions. 59

4.15 Comparison of Heuristic IBiD parameters (expansion balancers and potential functions) on a dynamic grid world pathfinding problem reproduced from [67]. Heuristic IBiD with only start-side expansions and a destination-side heuristic (top) proceeds identically to Lifelong Planning A*, performing 37 expansions on the original world (left) followed by 18 expansions over 14 vertices on the chanced world (right). Heuristic IBiD with distance-balanced expansions and an average potential (bottom) performs 30 expansions on the original world followed by 18 expansions over 15 vertices on the changed world. 62

4.16 Expected number of vertex expansions required to solve a single-pair shortest path query on a Northeast USA road network. Four problems P1-P4 were considered with different numbers of expected edges with traffic changes between episodes (P1: few changes, P4 many changes). The dataset consists of 400 instances of 10 episodes each deriving from 20 randomly selected vertex pairs and 20 random traffic seeds. The "Initial" series captures the first episode for each problem instance (with identical behavior between problems P1-P4 because no traffic changes affect the initial episode). The "Replans" series capture an average over the remaining nine episodes. Note that (a) only the latter four incremental planners see savings during replanning, and (b) the initial episode is identical between each incremental planner and its corresponding complete cousin. 65

4.17 Relative performance improvement for replanning episodes relative to initial episodes for each incremental algorithm on the dynamic road network problem. 65

5.1 A robot reaching to grab a book must allocate limited resources between planning and executing its motion. Our planner reasons over a user-defined utility function (top left) to explicitly trade-off between these two significant sources of cost. As its roadmap search discovers valid edges, it mediates between high-cost edges that are fast to find and low-cost edges that require significant planning directly via their prospective utilities. Because the planner accepts arbitrary estimators for planning cost, it can naturally exploit both caching and geometric structure in $\mathcal{C}$. 68
5.2 Contours of a utility function $U(p, x)$. Also some examples of simple utility functions relevant to related work. 71
5.3 A planner reasoning with a linear utility function chooses to first evaluate trajectory ξ_1. After incurring planning cost $\bar{p}$ (less than it had expected), it finds that it is more expensive than expected; some of the planning work has also adjusted the estimates for nearby trajectory ξ_2. However, the estimated remaining planning costs $\grave{p}$ for all other unaffected trajectories have remained constant. Therefore, the relative utilities have also not changed. As such, a planner need not re-order any trajectories whose estimated planning cost to go has not changed. 73
5.4 At each iteration i, the simplified bidirectional RRT-Connect planner always selects among all paths constrained to pass through the sampled configuration q_i that which minimizes the necessary planning cost only. 74
5.5 A parameterized planner A maximizes a linear combination utility (for some value λ_U) along the subset of parameter values which constitute the lower-left convex hull of the p-vs-x plot (left). Each realization of planner A for a particular parameter value (a point at left) corresponds to a line at right, which shows the utility achieved for that realization over various values of λ_U (negative utility shown, lower is better). Applying a parameterized planner for utility maximization requires selecting η according to λ_U in such a way that the realized utility approximates the lower bound across all parameter choices (right). 75
5.6 Schedule of parameters for three of the algorithms compared. The hybrid RRT-Connect and BIT* are both anytime planners. The parameter learned was the algorithm termination time after the first returned path. The LEMUR algorithm does not require tuning; we used $\lambda_p = \lambda_U$ in our experiments. 80

5.7 Illustration of three trajectories generated by LEMUR on instance 6 from our experiments. The planner was initialized with the parameters $\lambda_p = 0, 0.5,$ and 1; the same roadmap was used. By increasing λ_p, the planner prefers minimizing planning cost at the expense of more costly solution paths. 81
5.8 Comparison of measured planning time p and solution execution cost x for the first step of the HERB table-clearing task. Results for four parameterized planners are shown: RRT-Connect, BIT*, Lazy ARA*, and LEMUR. The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 to 0.99. The results for other planners show the effect of changing the total optimization time after the first returned solution. Also shown for reference are the contours for the $\lambda_U = 0.50$ utility function (i.e. 1 rad = 1 sec). 81
5.9 Comparison of measured planning time p and solution execution cost x for the FG step of the workcell task. Results for four parameterized planners are shown: RRT-Connect, BIT*, Lazy ARA*, and LEMUR. The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 to 0.99. The results for other planners show the effect of changing the total optimization time after the first returned solution. Also shown for reference are the contours for the $\lambda_U = 0.50$ utility function (i.e. 1 rad = 1 sec). 82
5.10 Comparison of measured planning time p and solution execution cost x for the first step of a CHIMP valve turning task. Results for four parameterized planners are shown: RRT-Connect, BIT*, Lazy ARA*, and LEMUR. The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 to 0.99. The results for other planners show the effect of changing the total optimization time after the first returned solution. Also shown for reference are the contours for the $\lambda_U = 0.50$ utility function (i.e. 1 rad = 1 sec). 82

6.1 A simple manipulation task: retrieve the mug from the table, and drop it in the blue bin. This task requires plans in three distinct C-space free subsets. 87
6.2 Illustration of a composite configuration space for a manipulation task. 88
6.3 Illustration of a transit and transfer constraint manifolds in the composite configuration space for a manipulation task, along with projections of each onto the robot's configuration space. 89
6.4 An illustration of a family motion planning problem in a common configuration space $\mathcal{C}$. The problem definition generalizes to an arbitrary number of configuration space subsets and set relations between them. When two queries in different subsets are solved sequentially, a family motion planner can reuse path segments less expensively. See Section 6.5 for examples in manipulation. 91

6.5 While queries in multi-query planning reference the same subset of $\mathcal{C}$, each family query references one of a number of such sets. 92
6.6 Types of subset relations. Each relation can be expressed directly as set relations w.r.t a set S, or equivalently as logical statements on the corresponding indicator functions $\mathbf{1}_S(\cdot)$. 92
6.7 Example family belief graph for a family of two subsets, S_1 and S_2. From the initial belief $b_{\text{init}} = (\text{U}, \text{U})$, each transition (S, r) represents invoking indicator $\mathbf{1}_S$ with returned binary result r. 93
6.8 Example family belief graph for a family of two subsets, S_1 and S_2, with the subset relation that $S_1 \subseteq S_2$. Compare this family belief graph to Figure 6.7 without the relation. 95
6.9 A subset relation which does not carry over directly from configurations to edges. 96
6.10 Optimistic optimal belief graph policy for a family of two subsets, S_1 and S_2, with the subset relation that $S_1 \subseteq S_2$, and with $\hat{p}_1 = 1$ and $\hat{p}_2 = 10$. The query subset is $S_u = S_2$; beliefs for which $b[2] = \text{T}$ are goal vertices (green). The value function d is shown for each believe state, and infeasible beliefs (i.e. with $d = \infty$, which are already demonstrably not in the query subset) are shown in green. The edges on the optimal policy are shown in bold. 96
6.11 A home robot performing a three-step manipulation task. It must move from its home configuration to grasp the cup, transfer it to a drop location above the bin, and return home. 98
6.12 Comparison of measured planning time p and solution execution cost x for the HERB table-clearing example. RRT is shown in red. Results for LEMUR with the simple edge cost model are shown in blue, while the family edge cost model is shown in green. The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 to 0.99. 99
6.13 Robot tending a press brake in an industrial workcell. This example problem is reproduced from the Lazy PRM paper [10]. The multistep manipulation task requires eight motion planning problems in seven different $\mathcal{C}$-subsets. From its initial configuration (A), the robot moves to grasp a raw sheet (B) and transfer first to a settling table (C) and subsequently to the press brake (D). After the first bend, the robot receives the partially worked part (E) and uses a regrasping fixture (F) to regrasp it on the other side (G). After its second bend (H) and (I), the finished part is moved to the final pallet (J) before the robot returns to its initial configuration (A). Results for planning these steps are shown in Figure 6.14. 100

6.14 Comparison of measured planning time p and solution execution cost x for the industrial workcell example. RRT is shown in red. Results for LEMUR with the simple edge cost model are shown in blue, while the family edge cost model is shown in green. The LEMUR results show the effect of adjusting the λ_p tradeoff parameter from 0 to 0.99. 100

7.1 Outline of the algorithms developed in this dissertation. LEMUR is solves a continuous motion planning problem via discretization as a series of progressively densified roadmaps (Chapter 2). Since the shortest path problem over the resulting graph in characterized by edge costs which are expensive to evaluate, we exploit lazy search and edge selectors (Chapter 3) to minimize planning effort. We also develop a novel incremental bidirectional search algorithm (Chapter 4) to accommodate the resulting dynamic pathfinding problem. LEMUR conducts its search guided by a utility function (Chapter 5) which can employ distinct domain-specific planning and execution cost heuristics. In multi-step manipulation tasks, one such cost model is derived from the family motion planning problem (Chapter 6), which leads to planner invocations which minimize combined planning and execution cost. 102
7.2 A stack of progressively densified roadmaps over a given free configuration space $\mathcal{C}$. 106
7.3 A roadmap stack with "offramp" edges. 108
7.4 Collision validity checking is a commonly used indicator function. The family motion planning formulation allows an intelligent planner to reach inside the checker's "black box" and reduce the number of costly narrow-phase checks. Resulting paths tend to be cheaper to compute and stay further from obstacles. 111
7.5 A simple 2D motion planning example in which LEMUR has a broad-phase check available. Testing against the bounding box (grey) is 10x less expensive than testing against the actual obstacle (black). Exposing the broad-phase check to LEMUR as a distinct subset allows for reduced planning cost. 113

A.1 Detailed timing results for each selector. The actual edge weights for the illustrative PartConn and UnitSquare problems were pre-computed, and therefore their timings are not included. The Partition selector requires initialization of the Z-values (3.7) for the graph using only the estimated edge weights. Since this is not particular to either the actual edge weights (e.g. from the obstacle distribution) or the start/goal vertices from a particular instance, this initialization (*sel-init*) is considered separately. The online running time (*online*) is broken into LazySP's three primary steps: the inner search (*search*), invoking the edge selector (*sel*), and evaluating edges (*eval*). We also show the number of edges evaluated. 123

D.1 Experimental results across six single-query motion planning instances for a 7-DOF robot arm. Top: expected planning cost p (in seconds) vs. execution cost x (in radians) for each parameterized planner for various values of their parameters. Bottom: the mean negative utility $-U = \lambda_U p + (1-\lambda_U)x$ (solid lines) measured for each planner (lower on plot is better) as the utility parameter λ_U is varied. The 95% confidence interval of the mean is also shown. Planners used the same parameter schedule across the problems as shown in Figure 5.6. The per-problem maximum achievable mean utilities (5.9) for each planner are shown as dotted lines. Planners are RRT-Connect with shortcutting, BIT*, LEMUR (no roadmap cache), and LEMUR (with roadmap cache). 133
D.2 Time breakdown of LEMUR for each of the six experimental problems across various values of λ_p. Cumulative time is shown performing the following operations: roadmap generation, graph search, collision checking, and unaccounted for. The bottom row shows the results of the algorithm over the same set of 50 roadmaps when reading the cached roadmap from disk to avoid online nearest-neighbor queries. 134
D.3 Summary of p-vs-x and utility results for each step in the workcell problem. 134
D.4 Timing results for each step in the workcell problem. 134

E.1 Family belief graph for the 2D example problem from Figure 6.4 in Chapter 6. The underlying family consists of five sets: A, B, C, S_{12}, and S_{23}. Transitions between states are shown for indicators $\mathbf{1}_A$, $\mathbf{1}_B$, and $\mathbf{1}_C$, with solid lines showing transitions in which the indicator returns True, and dashed lines showing transitions in which the indicator returns False. Also shown is the distance function and optimal policy for a query subset $S_u = S_{23}$, with green beliefs as goal states. Bolded edges represent transitions on an optimal policy. 135

List of Tables

1.1 Table of four computational problems and the respective algorithms developed in this dissertation. 12

2.1 Table of roadmap connection radii parameters for various scaling rates across the different robot platforms considered in this thesis. Radii presented are for $n = 10000$ and $\eta = 1$, and are given in radians. 24

3.1 LazySP equivalence results. The A*, LWA*, and BHFFA algorithms use reopening and the dynamic h_{lazy} heuristic (3.4). 30

4.1 IBiD generalizes both the heuristic-informed bidirectional Dijkstra's search [45] and DynamicSWSF-FP [98]. There are a great many algorithms that we could place in each cell; we provide only a representative choice in each. 44
4.2 Table of algorithms compared in the Northeast USA dynamic pathfinding problem. Each algorithm is marked as bidirectional ("Bidir"), heuristic-informed ("Heur"), and/or incremental ("Inc"). 63
4.3 Traffic transition parameters for each edge of the Northeast USA graph for the four problem classes P1 – P4. 64

6.1 Constituent subsets in the `herbbin` example problem. 98
6.2 Constituent subsets in the `workcell` example problem. 99

F
Bibliography

[1] S. Aine and M. Likhachev. Truncated incremental search. *Artificial Intelligence*, 234(C):49 – 77, May 2016.

[2] D. Alberts, G. Cattaneo, U. Nanni, and C. D. Zaroliagis. A software library of dynamic graph algorithms. In *Proceedings of Workshop on Algorithms and Experiments (ALEX-98)*, pages 129–136, University of Trento, Trento, Italy, 1998.

[3] O. Arslan and P. Tsiotras. Use of relaxation methods in sampling-based algorithms for optimal motion planning. In *IEEE International Conference on Robotics and Automation*, pages 2421–2428, May 2013.

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

[5] M. J. Beckmann, C. B. McGuire, and C. B. Winsten. Studies in the economics of transportation. Technical Report RM-1488-PR, RAND Corporation, Santa Monica, CA, USA, May 1955.

[6] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[7] D. Berenson, S. Srinivasa, D. Ferguson, and J. Kuffner. Manipulation planning on constraint manifolds. In *IEEE International Conference on Robotics and Automation*, May 2009.

[8] C. Berge and A. Ghouila-Houri. *Programming, Games and Transportation Networks*. Methuen, London, NY, USA, 1965.

[9] A. Bhatia, L. Kavraki, and M. Vardi. Sampling-based motion planning with temporal goals. In *IEEE International Conference on Robotics and Automation*, pages 2689–2696, May 2010.

[10] R. Bohlin and E. Kavraki. Path planning using Lazy PRM. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 521–528 vol.1, 2000.

[11] V. Boor, M. H. Overmars, and A. F. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 1018–1023 vol.2, 1999.

[12] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report: Structural layer proposal. In P. Mutzel, M. Jünger, and S. Leipert, editors, *9th Int. Symp. Graph Drawing, Vienna, Austria, September 23–26 2001, Revised Papers*, pages 501–512. Springer, Berlin, Heidelberg, Feb. 2002. Lecture Notes in Computer Science, Volume 2265.

[13] M. S. Branicky, S. M. LaValle, K. Olson, and L. Yang. Deterministic vs. probabalistic roadmaps. Unpublished, 2002.

[14] E. Burns, W. Ruml, and M. B. Do. Heuristic search when time matters. In *International Joint Conference on Artificial Intelligence*, volume 47, pages 697–740, 2013.

[15] S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research*, 28(1):104–126, 2009.

[16] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA, USA, 1988.

[17] P. Cheng and S. M. LaValle. Resolution completeness for sampling-based motion planning with differential constraints. Unpublished, 2004.

[18] H. Choset, K. M. Lynch, S. Hutchinson, G. A. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, June 2005.

[19] S. Choudhury, C. Dellin, and S. Srinivasa. Pareto-optimal search over configuration space beliefs for anytime motion planning. In *IEEE International Conference on Intelligent Robots and Systems*, 2016.

[20] B. Cohen, M. Phillips, and M. Likhachev. Planning single-arm manipulations with n-arm robots. In *Proceedings of Robotics: Science and Systems*, Berkeley, USA, July 2014.

[21] G. B. Dantzig. Linear programming and extensions. Technical Report R-366-PR, RAND Corporation, Santa Monica, CA, USA, Aug. 1963.

[22] D. de Champeaux. Bidirectional heuristic search again. *J. ACM*, 30(1):22–32, Jan. 1983.

[23] D. de Champeaux and L. Sint. An improved bidirectional heuristic search algorithm. *J. ACM*, 24(2):177–191, Apr. 1977.

[24] C. Dellin and S. Srinivasa. A general technique for fast comprehensive multi-root planning on graphs by coloring vertices and deferring edges. In *IEEE International Conference on Robotics and Automation*, 2015.

[25] C. Dellin, K. Strabala, G. C. Haynes, D. Stager, and S. Srinivasa. Guided manipulation planning at the DARPA Robotics Challenge trials. In *International Symposium on Experimental Robotics*, June 2014.

[26] C. Demetrescu, D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah and M. Blanton, editors, *Algorithms and Theory of Computation Handbook*, pages 9–9. Chapman & Hall/CRC, 2010.

[27] C. Demetrescu, A. V. Goldberg, and D. S. Johnson. 9th DIMACS implementation challenge: Shortest paths. `http://www.dis.uniroma1.it/challenge9/`, 2006. [Online; accessed 2016-May-22].

[28] R. Diankov. *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Robotics Institute, Carnegie Mellon University, Sept. 2010. CMU-RI-TR-10-29.

[29] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerishe Mathematik*, 1(1):269–271, 1959.

[30] J. E. Doran. Doubletree searching and the graph traverser. Technical Report EPU-R-22, Department of Machine Intelligence and Perception, University of Edingurgh, Dec. 1966.

[31] J. E. Doran and D. Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 294(1437):235–259, 1966.

[32] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

[33] D. Eppstein, Z. Galil, and G. Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook, chapter 8*. CRC Press, 1999.

[34] B. Faverjon. Obstacle avoidance using an octree in the configuration space of a manipulator. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 504–512, Mar. 1984.

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

[36] D. Ferguson and A. Stentz. *Field D*: An Interpolation-Based Path Planner and Replanner*, pages 239 – 253. Springer, Berlin, Heidelberg, 2007.

[37] L. R. Ford. Network flow theory. Technical Report P-923, RAND Corporation, Santa Monica, CA, USA, Aug. 1956.

[38] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 212–221, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[39] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms*, 34(2):251–281, Feb. 2000.

[40] J. Gammell, S. Srinivasa, and T. Barfoot. Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *IEEE International Conference on Intelligent Robots and Systems*, pages 2997 – 3004, Sept. 2014.

[41] J. Gammell, S. Srinivasa, and T. D. Barfoot. Batch informed trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *IEEE International Conference on Robotics and Automation*, May 2015.

[42] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling. FFRob: An efficient heuristic for task and motion planning. In *International Workshop on the Algorithmic Foundations of Robotics*, 2014.

[43] R. Gayle, K. Klingler, and P. Xavier. Lazy reconfiguration forest (LRF) - an approach for motion planning with multiple tasks in dynamic environments. In *IEEE International Conference on Robotics and Automation*, pages 1316–1323, Apr. 2007.

[44] A. V. Goldberg. Point-to-point shortest path algorithms with preprocessing. In J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, and F. Plášil, editors, *SOFSEM 2007: Theory and Practice of Computer Science: 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007. Proceedings*, pages 88–102. Springer, Berlin, Heidelberg, 2007.

[45] A. V. Goldberg and R. Werneck. Computing point-to-point shortest paths from external memory. In *SIAM Workshop on Algorithms Engineering and Experimentation (ALENEX '05)*, Vancouver, Canada, 2005.

[46] F. Gravot, S. Cambon, and R. Alami. aSyMov: A planner that deals with intricate symbolic and geometric problems. In P. Dario and R. Chatila, editors, *The International Symposium on Robotics Research*, volume 15 of *Springer Tracts in Advanced Robotics*, pages 100–110. Springer, Berlin, Heidelberg, 2005.

[47] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[48] K. Hauser. Lazy collision checking in asymptotically-optimal motion planning. In *IEEE International Conference on Robotics and Automation*, pages 2951–2957, May 2015.

[49] K. Hauser and J.-C. Latombe. Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*, 29(7):897–915, 2010.

[50] M. Helmert. The fast downward planning system. *Artificial Intelligence Research*, 2006.

[51] D. Hsu, J.-C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *IEEE International Conference on Robotics and Automation*, volume 3, pages 2719–2726 vol.3, Apr. 1997.

[52] D. Hsu, G. Sanchez-Ante, and Z. Sun. Hybrid PRM sampling with a cost-sensitive adaptive strategy. In *IEEE International Conference on Robotics and Automation*, pages 3874 – 3880, Apr. 2005.

[53] J. Ichnowski and R. Alterovitz. Parallel sampling-based motion planning with superlinear speedup. In *IEEE International Conference on Intelligent Robots and Systems*, pages 1206–1212, Oct. 2012.

[54] T. Ikeda, M.-Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by AI search techniques. In *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pages 291–296, Aug. 1994.

[55] L. Jaillet and T. Siméon. A PRM-based motion planner for dynamically changing environments. In *IEEE International Conference on Intelligent Robots and Systems*, volume 2, pages 1606–1611 vol.2, Sept. 2004.

[56] L. Janson, B. Ichter, and M. Pavone. Deterministic sampling-based motion planning: Optimality, complexity, and performance. *The International Symposium on Robotics Research*, 2015.

[57] L. Janson, E. Schmerling, A. Clark, and M. Pavone. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *The International Journal of Robotics Research*, 34(7):883–921, 2015.

[58] H. Kaindl and G. Kainz. Bidirectional heuristic search reconsidered. *J. Artif. Int. Res.*, 7(1):283–317, Dec. 1997.

[59] M. Kallman and M. Mataric. Motion planning using dynamic roadmaps. In *IEEE International Conference on Robotics and Automation*, volume 5, pages 4399–4404 Vol.5, Apr. 2004.

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

[61] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[62] L. Kavraki. Computation of configuration-space obstacles using the fast fourier transform. *IEEE Transactions on Robotics and Automation*, 11:255–261, 1995.

[63] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, Aug. 1996.

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

[65] Klein and Subramanian. A fully dynamic approximation scheme for shortest paths in planar graphs. *Algorithmica*, 22:235–249, 1998.

[66] S. Koenig and M. Likhachev. D* Lite. In *Eighteenth National Conference on Artificial Intelligence*, pages 476 – 483, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[67] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1):93–146, 2004.

[68] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[69] J. J. Kuffner and S. M. LaValle. RRT-Connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, volume 2, pages 995–1001, 2000.

[70] S. M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report 98-11, Computer Science Dept., Iowa State University, Oct. 1998.

[71] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[72] S. M. LaValle and M. S. Branicky. On the relationship between classical grid search and probabilistic roadmaps. In *International Workshop on the Algorithmic Foundations of Robotics*, 2002.

[73] S. M. LaValle, M. S. Branicky, and S. R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. *The International Journal of Robotics Research*, 23(7-8):673–692, 2004.

[74] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 473–479, 1999.

[75] P. Leven and S. Hutchinson. Toward real-time path planning in changing environments. In *International Workshop on the Algorithmic Foundations of Robotics*, 2000.

[76] P. Leven and S. Hutchinson. A framework for real-time path planning in changing environments. *The International Journal of Robotics Research*, 21(12):999–1030, 2002.

[77] 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*, volume 4, pages 3411–3416, 2002.

[78] J.-M. Lien and Y. Lu. Planning motion in environments with similar obstacles. In *Proceedings of Robotics: Science and Systems*, Seattle, USA, June 2009.

[79] M. Likhachev. Search-Based Planning Library (SBPL). `https://github.com/sbpl/sbpl/`, 2008. [Online; accessed 2016-Aug-10].

[80] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime search in dynamic graphs. *Artificial Intelligence*, 172(14):1613 – 1643, 2008.

[81] M. Likhachev, G. J. Gordon, and S. Thrun. ARA* : Anytime A* with provable bounds on sub-optimality. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, pages 767–774. MIT Press, 2004.

[82] C.-C. Lin and R.-C. Chang. On the dynamic shortest path problem. *J. Inf. Process.*, 13(4):470–476, Apr. 1991.

[83] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C-32(2):108–120, Feb. 1983.

[84] T. Lozano-Pérez and M. A. Wesley. An algorithm for planning collison-free paths among polyhedral obstacles. *Communications of the ACM*, 22:560–570, 1979.

[85] M. Luby and P. Ragde. A bidirectional shortest-path algorithm with good average-case behavior. *Algorithmica*, 4(1):551–567, June 1989.

[86] E. F. Moore. The shortest path through a maze. In *Proceedings of an International Symposium on the Theory of Switching*, Cambridge, MA, USA, 1959. Harvard University Press.

[87] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, and G. Konidaris. Robot motion planning on a chip. In *Proceedings of Robotics: Science and Systems*, Ann Arbor, Michigan, June 2016.

[88] W. S. Newman and M. S. Branicky. Real-time configuration space transforms for obstacle avoidance. *The International Journal of Robotics Research*, 10(6):650–667, 1991.

[89] T. A. J. Nicholson. Finding the shortest route between two points in a network. *The Computer Journal*, 9(3):275–280, 1966.

[90] J. Pan, S. Chitta, and D. Manocha. FCL: A general purpose library for collision and proximity queries. In *IEEE International Conference on Robotics and Automation*, pages 3859–3866, May 2012.

[91] M. Phillips, B. J. Cohen, S. Chitta, and M. Likhachev. E-graphs: Bootstrapping planning with experience graphs. In *Proceedings of Robotics: Science and Systems*, 2012.

[92] M. Pivtoraiko and A. Kelly. Efficient constrained path planning via search in state lattices. In *The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, Sept. 2005.

[93] E. Plaku, L. E. Kavraki, and M. Y. Vardi. Motion planning with dynamics by a synergistic combination of layers of planning. *IEEE Transactions on Robotics*, 26(3):469–482, June 2010.

[94] I. Pohl. *Bi-directional and Heuristic Search in Path Problems*. PhD thesis, Stanford University, Stanford, CA, USA, May 1969. AAI7001588.

[95] I. Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3âĂŞ4):193 – 204, 1970.

[96] I. Pohl. Bi-directional search. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*, chapter 9, pages 127–140. Edinburgh University Press, 1971.

[97] S. Quinlan. *Real-time modification of collision-free paths*. PhD thesis, Stanford University, Dec. 1994.

[98] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267 – 305, 1996.

[99] J. H. Reif. Complexity of the mover's problem and generalizations. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 421–427, Oct. 1979.

[100] E. Rimon and D. E. Koditschek. The construction of analytic diffeomorphisms for exact robot navigation on star worlds. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 21–26, May 1989.

[101] W. Ruml and M. B. Do. Best-first utility-guided search. In *International Joint Conference on Artificial Intelligence*, pages 2378–2384, 2007.

[102] K. Salisbury, W. Townsend, B. Ebrman, and D. DiPietro. Preliminary design of a whole-arm manipulation system (WAMS). In *IEEE International Conference on Robotics and Automation*, pages 254–260 vol.1, Apr. 1988.

[103] O. Salzman, D. Shaharabani, P. K. Agarwal, and D. Halperin. Sparsification of motion-planning roadmaps by edge contraction. *The International Journal of Robotics Research*, 33(14):1711–1725, 2014.

[104] G. Sánchez-Ante and J.-C. Latombe. A single-query bidirectional probabilistic roadmap planner with lazy collision checking. In *ISRR*, pages 403–417, 2001.

[105] J. Schulman, J. Ho, A. Lee, I. Awwal, H. Bradlow, and P. Abbeel. Finding locally optimal, collision-free trajectories with sequential convex optimization. In *Proceedings of Robotics: Science and Systems*, Berlin, Germany, June 2013.

[106] J. T. Schwartz and M. Sharir. On the piano movers' problem: I. the case of a two-dimensional rigid polygonal body moving amidst polygonal barriers. *Communications on Pure and Applied Mathematics*, 36:345–398, 1983.

[107] A. Shimbel. Structure in communication nets. In *Proceedings of the Symposium on Information Networks (New York, 1954)*, pages 199–203, Brooklyn, New York, NY, USA, 1955. Polytechnic Press of the Polytechnic Institute of Brooklyn.

[108] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, 2001.

[109] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani. Manipulation planning with probabilistic roadmaps. *The International Journal of Robotics Research*, 23(7–8):729–746, 2004.

[110] T. Siméon, J.-P. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps for motion planning. *Journal of Advanced Robotics*, 14(6):477–493, 2000.

[111] P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4(3):375–380, 1975.

[112] S. Srinivasa, D. Berenson, M. Cakmak, A. Collet, M. Dogar, A. Dragan, R. Knepper, T. Niemueller, K. Strabala, M. V. Weghe, and J. Ziegler. HERB 2.0: Lessons learned from developing a mobile manipulator for the home. *Proceedings of the IEEE*, 100(8):2410–2428, Aug. 2012.

[113] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. In *IEEE International Conference on Robotics and Automation*, pages 639–646, May 2014.

[114] J. A. Starek, J. V. Gomez, E. Schmerling, L. Janson, L. Moreno, and M. Pavone. An asymptotically-optimal sampling-based algorithm for bi-directional motion planning. In *IEEE International Conference on Intelligent Robots and Systems*, pages 2072–2078, Sept. 2015.

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

[116] A. Stentz. The focussed D* algorithm for real-time replanning. In *In Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–âĂŞ1659, 1995.

[117] T. Stentz, H. Herman, A. Kelly, E. Meyhofer, G. C. Haynes, D. Stager, B. Zajac, J. A. Bagnell, J. Brindza, C. Dellin, M. George, J. Gonzalez-Mora, S. Hyde, M. Jones, M. Laverne, M. Likhachev, L. Lister, M. Powers, O. Ramos, J. Ray, D. Rice, J. Scheifflee, R. Sidki, S. Srinivasa, K. Strabala, J.-P. Tardif, J.-S. Valois, J. M. Vande Weghe, M. Wagner, and C. Wellington. CHIMP, the CMU Highly Intelligent Mobile Platform. *Journal of Field Robotics*, Feb. 2015.

[118] R. Stern, A. Felner, J. van den Berg, R. Puzis, R. Shah, and K. Goldberg. Potential-based bounded-cost search and anytime non-parametric A*. *Artificial Intelligence*, 214:1–25, 2014.

[119] I. A. Şucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, Dec. 2012. http://ompl.kavrakilab.org.

[120] A. G. Sukharev. Optimal strategies of the search for an extremum. *USSR Computational Mathematics and Mathematical Physics*, 11(4):119–137, 1971. Translated from Russian, Zh. Vychisl. Mat. Mat. Fiz., 11:(4):910–924, 1971.

[121] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *In Proceedings of AAAI*, 2000.

[122] M. Zucker, N. Ratliff, A. Dragan, M. Pivtoraiko, M. Klingensmith, C. Dellin, J. A. Bagnell, and S. Srinivasa. CHOMP: Covariant hamiltonian optimization for motion planning. *The International Journal of Robotics Research*, 32(9–10):1164–1193, 2013.