Efficient Robot Motion Planning in Cluttered Environments

Aditya Vamsikrishna Mandalika

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2021

Reading Committee:

Siddhartha S. Srinivasa, Chair

Maxim Likhachev

Sanjiban Choudhury

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Efficient Robot Motion Planning in Cluttered Environments

Aditya Vamsikrishna Mandalika

Chair of the Supervisory Committee:
Boeing Endowed Professor Siddhartha S. Srinivasa
Computer Science and Engineering

Robotics has become a part of the solution in various applications today: autonomous vehicles navigating busy streets, articulated robots tirelessly sorting packages in warehouses, feeding people in care homes and mobile robots assisting in rescue operations. Central to any robot that needs to navigate its environment, is *Motion Planning*: the task of computing a collision-free motion for a (robotic) system between given start and goal states in an environment cluttered with obstacles. As tasks become more complex, there is a need to develop more sophisticated motion planning algorithms that can compute *high quality* solutions for the robot *quickly*. This thesis primarily address the challenge in three phases:

First, we approximate the optimal motion planning problem in a continuous space to a search for the shortest path on a discrete graph abstraction. We investigate the computational bottlenecks in search: *graph operations* and *collision evaluations* and propose GLS, an algorithmic framework to balance the computational effort between the two operations. In addition to showing that GLS captures an oracular behaviour that minimizes planning time, we propose strategies to balance the computational effort and approximate such an oracle.

Second, we focus on the graph abstraction. A desirable graph is sparse allowing for fast search (fewer graph operations and edge evaluations) but locally dense in cluttered regions such that a feasible low-cost path exists. We note that there is structural similarity in the environments that a robot typically operates in. To this end, we propose LEGO, to

leverage a robot's experience in similar environments and learn to generate sparse graphs that adequately sample bottleneck regions in the environment while ensuring a high quality solution exists.

Third, we relax the assumption of a *fixed* discrete graph abstraction to compute the optimal solution in the continuous space. We extend the computational efficiency that GLS provides to an incremental asymptotically-optimal sampling-based algorithm. IGLS computes the optimal solution in an anytime manner by iteratively sampling increasingly dense graphs and minimizing the planning time at each iteration. We further investigate improving the convergence rate of such algorithms. Asymptotically-optimal planners typically compute an initial solution and subsequently focus sampling graphs in an Informed Set defined by the current solution cost. We propose GuILD which leverages the *search tree* constructed in every iteration to further inform sampling for a faster convergence to optimality.

Finally, we test our algorithms and evaluate their efficacy on a suite of robotic platforms and planning problems. Our results demonstrate our algorithms to significantly reduce planning times across domains and outperform competing planners.

Acknowledgements

Thank you to my advisor, Sidd Srinivasa, for shaping me into the researcher I am today. Sidd, your excitement and joy for problem solving has inspired me since the day we met. When I was completely clueless about what computer science and robotics would entail, you took me under your wing. You patiently taught me to ask the right questions, be fearless in my research and optimistic yet skeptical in my approach. I could not have asked for a better advisor to guide me on this journey. Thank you. I thank members of my committee - Dan Weld, Dieter Fox, Maxim Likhachev, Sawyer Fuller and Sanjiban Choudhury - for their invaluable technical insights and comments. Thank you Dan - your words of support and appreciation after every milestone gave me strength to do even better. Max, I thoroughly enjoyed our discussions on search and your thoughtful questions - thank you for sharing your insights and knowledge.

Sanjiban - I am grateful to have had the great fortune of working with you as one of my mentors. I lost track of how many whiteboards across school we have filled writing proofs and developing algorithms. The breadth of your technical expertise, the depth in your thoughts and your philosophy on life continue to inspire me to be a better scientist. Our research discussions going on for hours into the night are some of my most cherished moments from this journey. I'd like to thank Oren Salzman, another academic giant whose shoulders I had the fortune of standing upon. Oren, thank you for introducing me to the wonderful world of motion planning, for teaching me how to think about search and develop theory in a structured manner. Nothing beats the occasional coffee/tea breaks to the RI kitchen while discussing search with you.

Personal Robotics Lab has been my family away from home. I could not have asked for a better group of people among whom I could grow as a roboticist and as a person. It was not just their vast technical expertise, but their empathy and willingness to help, care and support each other that was inspiring. Thank you Sidd for fostering such a culture. Thank you to my labmates from CMU: Shushman, Clint, Koval, Oren, Pras, Rachel, Gilwoo, JS, Rosario, Brian, Shervin, Laura, Henny, Hanjun, Youngsun and Daqing; and from UW: Sanjiban, Tapo, Kay, Ethan, Patrick, Schmittle, Chris, Sherdil, Willie and Amal.

A special callout to Shushman, Rosario and Brian! Shushman has been my pillar of strength since my first day at CMU. Thank you for adopting me as your family's youngest Choudhury brother, for always keeping the zingers alive and the mood light - ensuring I stay

grounded, and for being someone I could constantly look up to and emulate. Rosario, for teaching me that there is more to life than work. When work got to me, the hikes, ski trips and mountaineering adventures that you took me on always managed to rejuvenate me back to life. Brian, for rolling his eyes every time I did something less-than-ideal. I cannot remember the number of times I came to you seeking help, and you'd spend hours helping me with my presentations, papers, talks and... life; all of that while occasionally walking to the kitchen to check on the cookies you make for us. With Shushman, Rosario and Brian alongside, nothing seemed too difficult to handle.

I'd like to thank Elise for making grad school easy to navigate and for always being available to advise on academic matters. I'd also like to thank Barbara (BJ) at RI for making grad school at CMU a memorable experience - this journey began with an acceptance letter that BJ sent and with her warm welcome to the US. Going back further, I'd like to thank my undergrad advisor Arun Mahindrakar for introducing me to research and to my first real robots. I'd also like to thank my school teacher, Bala Tripura Sundari, for laying the foundations to my academic interests and for instilling in me the confidence to aim high.

Thank you to the friends I made on the way, especially Shruti, Koushik, Prithvi, Poojita, Sid and Nimisha for keeping me sane, listening to my (many) problems, providing moral and emotional support, and of course, for all the laughs and joy along the way. My undergrad friends - Ravi and Yashwanth, and my school friends - Prashanth and Pranav: thank you for sticking by my side all these years, for encouraging me to chase my dreams and for the adventures we've been on and yet to go on.

I am eternally grateful to my parents and sister for always supporting me in achieving my goals, and for constantly reminding me to focus on being a better person first, a better scientist next. None of this would have been possible without you.

My final thanks to HERB... for being broken when I joined PRL. Fixing him was an absolute joy and a great learning experience. Working with him made me realize that my heart was truly in the robot.

Funding

This work contains work (partially) funded by the National Science Foundation IIS (#2007011), National Science Foundation DMS (#1839371), the Office of Naval Research, US Army Research Laboratory CCDC, Amazon, and Honda Research Institute USA.

Contents

List of Figures

List of Tables

1
Introduction

Recent years have seen a swift yet steady increase in the development of autonomous systems: self-driving vehicles navigating complex road scenarios, warehouse robots operating in constrained spaces, medical robots performing intricate surgeries and robotic arms perfoming assistive feeding. As the adoption of new robots is taking active strides, so is our expectation in their capabilities. As tasks expected of robots become more complex and their environments more geometrically challenging, there is a pressing need to develop sophisticated algorithms that efficiently plan robot motions to perform these tasks.

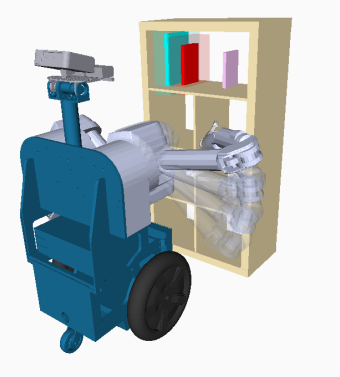


Figure 1.1: HERB[111], a bimanual mobile robot, reaching into a bookshelf to pick a book. Increasingly precise and intricate collision-free motions are necessitated as the arm reaches into the shelf, a computationally expensive problem.

Motion planning or the task of computing collision-free motions for a robot between given start and goal configurations, is a computationally challenging [104, 109] problem with decades of rich history [89]. From an algorithmic perspective, one of the foundational questions in motion planning has been on how to generate *high quality* solutions (or robot motions) *quickly*. The quality of a solution is measured with respect to the underlying desired objective function such as path length, time, energy expended in execution, distance from obstacles etc. While it is much easier to generate any *feasible* (collision-free) solution, determining a *high quality* or the *optimal* motion is often *computationally expensive* [69, 94, 93]. Figure 1.1 illustrates this with a motivating example. The complexity involved in computing optimal motion planning solutions can be attributed to the implicit difficulty of the planning problem, which inherently is a function of the following three contributing factors:

Planning in high-dimensional continuous statespaces Many robotic platforms such as (mobile) manipulators and non-holonomic mobile robots often have continuous statespaces of high-dimensionality. Since the planning procedure is expected to determine an optimal sequence of states in the robot's statespace, the high-dimensional continuous space lends to an exponential increase in computational complexity with di-

mension. This usually calls for *discretization* of the continuous space such as a graph abstraction where it still manifests as costly operations like nearest-neighor queries.

Expensive collision evaluations Robots are geometrically complex entities required to interact with the environment. In robots with many degrees of freedom, it is often intractable to explicitly reason about the collision status of given robot state. Instead, it involves computationally expensive intersection tests with the environment. When safety becomes paramount, the physics models for the environment and the robot are required to be of sufficiently high fidelity that further aggravates the computational cost. Such collision evaluations are often a bottleneck in planning for robots.

Complexity of the environment In cluttered and geometrically challenging environments, computing a collision-free optimal path requires the planning algorithm determine intricate subspaces of the robot's statespace where the optimal path lies. Since it is intractable to represent the obstacles in high-dimensional statespaces, the algorithm has no a-piori knownledge of where to focus its search for the optimal path. This often leads to the algorithm expending effort to evaluate the collision status of a large number of *potential* optimal paths and eliminate them before computing the collision-free solution.

To address the problem of efficient optimal motion planning, this thesis adopts the *graph-based search* paradigm. We begin by discussing how the aforementioned factors manifest within this abstraction and how we plan to address the problem in depth.

1.1 Problem Characterization

A popular approach to render the motion planning problem tractable is to employ graph-based search algorithms. Graphs, constructed as a discrete *approximation* of robot's continuous statespace, lend powerful tractability to robotic motion planning [89]. By considering only the vertices of the graph as possible robot states, these algorithms reduce the number sequences of robot states to consider for the (resolution-)optimal path thereby reducing complexity. These algorithms first *sample* a graph (or a roadmap) embedded in the robot's statespace and then *search* for the shortest path on this graph. The graphs can be explicit, i.e., constructed as part of a pre-processing stage [74, 72, 64], or implicit, i.e., discovered incrementally during search [92, 40, 106]. The difficulty of efficient motion planning under this abstraction can be broken down into three principal challenges:

C1: Compute the shortest path while minimizing planning time
We start by considering the challenge of computing the shortest path on a given graph approximation of the robot's statespace while minimizing the computational effort. Firstly, we are required to define what constitutes the planning time while solving the shortest path problem on a graph. Subsequently, we need to design a planning framework to enable minimizing the planning time.

C2: Sample sparse graphs with local densification in narrow regions
While a shortest path algorithm is required to be efficient irrespective of the graph it is operating on, the graph itself affects (a) the effort in search since denser graphs lead to greater number of operations like costly nearest-neighbor queries and (b) the quality of the final solution computed. Given a planning problem, we aim to generate a desirable graph that is sparse, allowing for fast search, with samples spread out at key locations such that a low cost feasible path exists.

C3: Efficiently compute the optimal solution from graph approximations
While search on a fixed graph enables tractability of solving the motion planning problem, it only computes a resolution-optimal solution on the graph. Since the planning problem is not known a-priori, it is not possible to choose the right resolution for the graph approximation. Sparse graphs can be searched quickly but could lead to low-quality solutions. Dense graphs can generate continuous optimal-solutions but they can be prohibitively expensive to search. Incremental algorithms [70, 40, 39] iteratively densify the robot statespace to generate increasingly dense graph approximations to asymptotically converge to the optimal path. These algorithms, however, are often slow to converge either due to expensive collision evaluations or inefficient sampling - the two issues we investigate for faster convergence and improved finite-time performance.

1.2 Outline of Approach

This dissertation develops algorithms and learning frameworks to generate high quality solutions while addressing the sources of computational expense in optimal motion planning. Table 1.1 shows a mapping from challenges to associated chapters.

Background and Related Works (Chapter 2): We begin by reviewing the optimal motion planning problem and providing a brief overview of sampling-based algorithms. We introduce the notation used in the thesis and discuss several existing approaches from literature. Subsequent chapters address the core challenges.

Challenge	Chapter	Algorithm
C1: Efficient Search	Ch. 3: Generalized Lazy Search for Efficient Planning	LRA*, GLS
C2: Graph Generation	Ch. 4: Leveraging Experience for Graph Generation	LEGO
C3: Convergence to Optimality	Ch. 5: Incremental Generalized Lazy Search	IGLS
	Ch. 6: Guided Incremental Local Densification	GuILD

Table 1.1: Outline of Approach to address the Challenges

Generalized Lazy Search for Efficient Planning (Chapter 3): While the graph abstraction simplifies the motion planning problem, the underlying computational costs (such as expensive graph operations and edge validity evaluations) motivate us to develop a *lazy* planning framework to compute the shortest path efficiently. By balancing the two sources of costs in search: (a) vertex expansions and (b) edge evaluations, we show that our algorithm minimizes the total planning time. By providing flexibility in the choice of an EVENT and a SELECTOR (defined in Chapter 3), the planning framework allows to toggle between lazy search and expensive edge evaluation.

Leveraging Experience for Graph Generation (Chapter 4): In Chapter 4 we address the challenge of generating desirable graphs that are sparse yet adequately sample key locations such that the search algorithm can find a high quality path with small computational effort. We observe that the different environments that a robot typically operates in share a lot of structural similarity. This motivates us to use information extracted from planning on one such environment to decide how to sample on another i.e. learn sampling distributions using tools such as conditional variational auto-encoder (CVAE). To this end, we propose a framework to focus on extracting bottleneck samples along multiple diverse shortest path in various representative environments for training a CVAE to predict a graph given a new environment and a planning problem.

Incremental Lazy Motion Planning (Chapter 5): While GLS (Chapter 3) operates on a fixed graph, we observe that the efficiency it provides by balancing search effort with edge evaluations is applicable to optimal motion planning in continuous spaces. We extend GLS to an incremental sampling-based algorithm that iteratively constructs increasingly dense graphs to improve solutions in an anytime manner while minimizing the planning time to compute each solution.

Guided Incremental Local Densification (Chapter 6): Incremental algorithms such as BIT* [43] and IGLS (Chapter 5) incrementally densify the statespace to continually improve the initial solution. These algorithms focus sampling in an Informed Set defined by the current best

solution cost, a sufficient condition to acheive optimality. However, in complex high dimensional environments, the Informed Set is often ineffective in focussing sampling adequately. In Chapter 6, we propose GuILD to leverage, in addition to the current best solution cost, the *search tree* constructed by the planner to further focus sampling and improve the convergence rate.

1.3 Summary of Contributions

- *Generalized Lazy Search* (GLS) [93] is a family of lazy search algorithms that defines a generic *toggle* (or Event as defined in Chapter 3) between lazy search and edge evaluations to *minimize* the total planning time. *Lazy Receding-Horizon A** [94] is an instance of GLS that defines such a *toggle* as a function of the depth of lazy search.
- *Leveraging Experience with Graph Oracles* (LEGO) proposes a training framework to learn a generative model capable of sampling a graph that has adequate coverage in constrained spaces.
- *Incremental Generalized Lazy Search* (IGLS) is an incremental sampling-based algorithm that extends the computational efficiency of GLS to achieve asymptotic optimality.
- *Guided Incremental Local Densification* (GuILD) leverages the search tree constructed by an incremental sampling-based algorithm to focus sampling and achieve a faster convergence rate to optimality.
- Experimental evaluation of the algorithms on a range of simulated environments including high-dimensional manipulation tasks.
- Open-source implementations of the algorithms[1] for Open Motion Planning Library (OMPL)[118].

[1] Implementations at:
`https://github.com/personalrobotics/gls`
`https://github.com/personalrobotics/lego`
`https://github.com/personalrobotics/ompl`

2
Background and Related Work

Motion planning is a central theme to a wide range of robotic applications. While robots have evolved over decades of scientific and engineering effort, the problem of motion planning or navigating complex environments is implicit in the tasks they are required to perform. In this chapter, we formally introduce the problem of motion planning. We will present definitions and notation useful for the remainder of the thesis. and discuss a wide collection of related work pertaining to shortest path search on graphs, sampling strategies, and incremental sampling-based approaches for optimal motion planning. Since this thesis focuses on search on discrete graph approximations, so will our overview. For a more comprehensive study of motion planning we refer the reader to [89].

2.1 *Optimal Motion Planning*

Informally, given start and goal poses, the optimal motion planning problem seeks to compute a continuous sequence of robot poses between the start and goal, while minimizing a given objective function and respecting the physics of the environment such as avoiding collisions. We now formally describe this problem.

Let $\mathcal{X} \subseteq \mathbb{R}^n$ be the statespace of the planning problem and $\mathcal{X}_{\text{obs}} \subset \mathcal{X}$ denote the subspace occupied by obstacles. The free space is denoted by $\mathcal{X}_{\text{free}} = \mathcal{X} \backslash \mathcal{X}_{\text{obs}}$. Given source and target states $(x_s,\ x_t) \in \mathcal{X}_{\text{free}}$, a path $\xi : [0,1] \to \mathcal{X}$ is represented as a continuous sequence of states such that $\xi(0) = x_s$ and $\xi(1) = x_t$. A motion is considered valid if it lies completely within $\mathcal{X}_{\text{free}}$ i.e. $\forall t \in [0,1],\ \xi(t) \in \mathcal{X}_{\text{free}}$. Let Ξ be the set of all non-trivial paths. Given a cost functional $c : \xi \to \mathbb{R}_{>0}$, an optimal path ξ^* is a feasible path minimizing the objective function:

$$\xi^* = \underset{\xi \in \Xi}{\arg\min}\ c(\xi) \quad s.t. \tag{2.1}$$
$$\xi(0) = x_s,\ \xi(1) = x_t,\ \forall t \in [0,1],\ \xi(t) \in \mathcal{X}_{\text{free}}$$

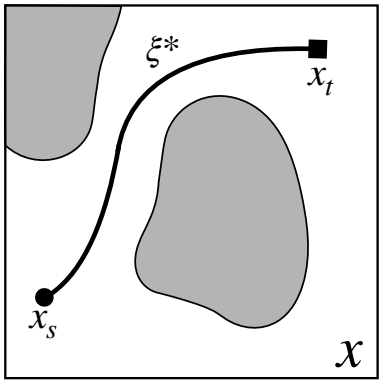


Figure 2.1: The optimal motion planning problem of computing a continuous sequence of collision-free states between start and goal that minimizes a given cost functional.

2.2 *Single Source Shortest Path Problem*

Solving the optimal motion planning problem as presented in Equation 2.1 is computationally hard [104, 109]. A common approach to make the problem tractable is to discretize the continuous statespace. These approaches typically construct a graph where vertices represent robot poses and edges represent potential movements of the robot [21, 89]. The shortest path on the graph provides an approximation to the true optimal solution in the continuous space. This approximates the Optimal Motion Planning problem as a Single Source Shortest Path (SSSP) problem on a discrete graph.

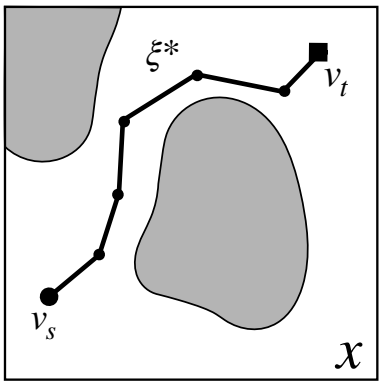


Figure 2.2: The SSSP problem of computing a sequence of discrete vertices of a graph between the start and goal vertices that minimizes the given cost functional over edges.

Consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ denote the set of vertices (or robot poses) and edges (or motions between two robot poses) in the graph respectively. Given start and goal vertices, $(v_s, v_t) \in \mathcal{V}$, a path $\xi = (v_s, v_1, v_2, \ldots, v_t)$ on the graph is a sequence of vertices where $\forall i, (v_i, v_{i+1}) \in \mathcal{E}$. To determine the validity of edges, we define the collision checking function or the world $\phi : \mathcal{E} \to \{0, 1\}$ as a mapping from edges to valid (1) or invalid (0). A path is said to be feasible if all edges are valid, i.e., $\forall e \in \xi, \phi(e) = 1$. We define an equivalence to the cost functional defined in the previous section, an edge weight $w : \mathcal{E} \to \mathbb{R}_{>0}$ which denotes the cost of traversing an edge. The cost of a path is the sum of edge costs, i.e., $w(\xi) = \sum_{e \in \xi} w(e)$. Similar to the optimal motion planning problem, the objective of the SSSP problem is to find the shortest feasible path:

$$\xi^* = \arg\min_{\xi} \; w(\xi) \quad s.t. \tag{2.2}$$
$$\xi[0] = v_s, \; \xi[1] = v_t, \; \forall e \in \xi, \; \phi(e) = 1$$

We first discuss various strategies typically employed to sample graphs and subsequently how the SSSP problem is solved.

2.2.1 *Graph Sampling Strategies*

Graphs lend powerful tractability to robotic motion planning [90]. They can be explicit, i.e., constructed as part of a pre-processing stage [75, 73, 67], or implicit, i.e., discovered incrementally during search [92, 44, 107]. While shortest path algorithms are generally agnostic to the graph they are operating on, the quality of the solution obtained and the amount of work required to compute the solution depend on the quality of the graph. If the graph is sparse and has vertices in key locations such as narrow passages, a high quality solution is obtained with little computational effort. Sampling strategies vary in their theoretical guarantees and the practical performance they lend to the shortest path algorithm.

Uniform Random Sampling One of the simplest sampling strategies, these algorithms maintain a uniform probability distribution over the statespace $\mathcal{X}$. Several sampling-based algorithms such as PRM*[72], FMT* [66], BIT* [40] and their variants construct graphs using a uniform random sampling strategy. This sampling scheme enjoys a strong theoretical foundation in ensuring graph connectedness and asymptotic optimality with the number of vertices in the graph.

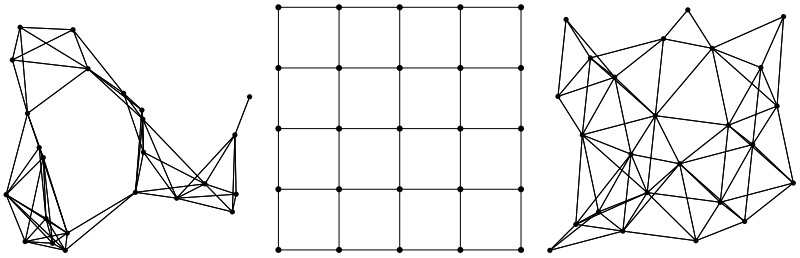


Figure 2.3: (a) Uniform random sampling (b) Lattice and (c) Halton sequence sampling

Low Dispersion Sequences An alternative to random sampling, low dispersion sequences aim to place samples such that the largest subspace of the statespace that does not contain a sample is minimized. They offer several advantages over random sampling. In addition to reducing dispersion, they are deterministic in their sampling allowing preprocessing and caching graph information such as heuristics and nearest neighbor queries. The simplest low dispersion graphs are lattices characterized by a fixed resolution along each dimension. Other popular deterministic sequences include Halton, Hammersley and Sukharev grids. For a more detailed discussion the reader is refered to LaValle (Chapter 5) [89] and Janson et al. [64].

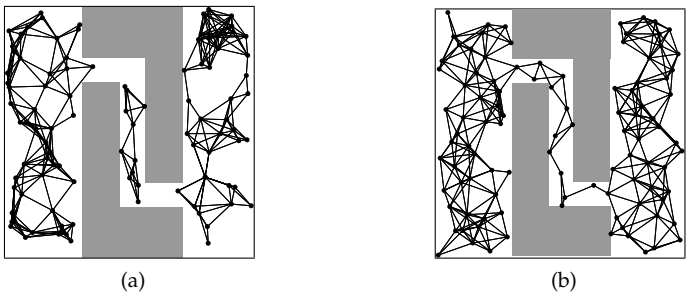


Figure 2.4: (a) Uniform random sampler and (b) Halton sampler each with 100 samples and a connection radius of 0.1.

Adaptive Sampling Strategies The seminal work of [59] provides a crisp analysis of the shortcomings of uniform sampling techniques, discussed in the previous section, in the presence of artifacts such as narrow passages. This has led to a plethora of non-uniform sampling approaches that densify selectively [55, 60, 16, 58, 88].

Adaptive sampling in the context of roadmaps aims to exploit structure of the environment to place samples in promising areas. A number of works exploited structure of the workspace to achieve this. While some of them attempt to sample between regions of collision to identify narrow passages [125, 55, 57, 87, 126, 123], others sample near or on the obstacles [4, 15]. There are approaches that divide the configuration space into regions and either select different region-specific planning [95] or sampling [19] strategies or use entropy of samples in a particular region to refine sampling [105]. Other methods try to model the free space to speed up planning [29, 100, 23]. While these techniques are quite successful in a large set of problems, they can place samples in regions where an optimal path is unlikely to traverse.

Leveraging Experience in Sampling A different class of solutions look at adapting sampling distributions online during the planning cycle. This requires a trade-off between exploration of the configuration space and exploitation of the current best solution. Preliminary approaches define a utility function to do so [121, 17] or use online learning [88]; however these are not amenable to using priors. [32] employs statistical techniques to sample around a search tree. [130, 86] formalize sampling as a model-free reinforcement learning problem and learn a parametric distribution. Since these problems are non i.i.d learning problems, they do require interactive learning and do not enjoy the strong guarantees of supervised learning.

Recently there has been effort on finding low dimensional structure in planning [124]. Particularly, generative modeling tools like variational autoencoders [35] have been used to great success [20, 48, 63, 28, 103]. We base our work on [62] which trains a CVAE to learn the shortest path distribution.

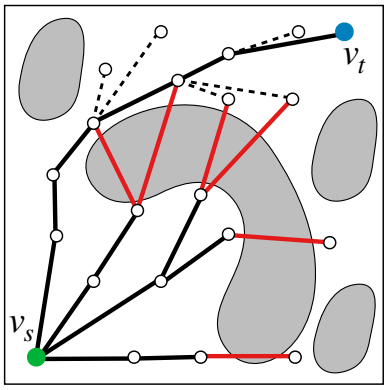


Figure 2.5: The search algorithm constructs a shortest path tree rooted at v_s extending to v_t. Solid lines visualize the evaluated edges.

2.2.2 *Solving the SSSP problem*

To solve the SSSP problem, shortest path algorithms take advantage of the Bellman's principle of optimality [7, 8] and use dynamic programming to compute the shortest path. Algorithms such as Dijkstra [33] process the vertices of the graph in an ordered fashion to minimize the number of vertices considered before terminating with the shortest path. This is done by maintaining a priority queue of vertices ordered by their *cost-to-come* and growing a search tree rooted at v_s. The algorithm terminates with the shortest path when the goal vertex $v_t$is popped from the queue for consideration. This allows Dijkstra to process only those vertices on the graph which can be reached from v_s with a cost-to-come less than the cost of the shortest path to v_t. This is an important characterstic since processing a vertex in the graph in-

volves expensive graph operations such as nearest neighbor queries, queue operations and rewiring the search tree for consistency in tracking the shortest path. Earlier work in this space saw extensive research in minimizing graph operations.

Addressing Expensive Graph Operations A* [52] and its variants have enjoyed widespread success in finding the shortest path with an optimal number of vertex expansions. Any other algorithm that guarantees to compute the shortest path on the graph will expand at least as many vertices as A* when using the same heuristic (under some mild assumptions) [30]. This motivated work that either improved the heuristic bounds on cost-to-come or cost-to-go values via preprocessing [99, 46], using domain-dependent heuristics [3] or using experience and imitation learning [9].

In robotics however, testing the collision status of edges or *edge evaluations* is more expensive than vertex expansions, and typically dominates the planning time. Since our focus is on these domains, we examine relevant works.

Addressing Expensive Collision Evaluations A common approach to algorithmically address expensive edge evaluations is to employ a *lazy approach* [14, 54, 78]. In this approach, the search tree on the graph is constructed *without* immediately testing if edges are collision-free. Only a subset of edges of the search tree are evaluated when necessary to save computation time. LazySP [31] and LazyPRM [14] extend the search tree up to the goal before evaluating edges for collision. LazySP also proposes and studies various strategies that determine which edge in the constructed search tree is evaluated. Lazy Weighted A*, LWA* [27] extends the search tree by a single edge before evaluation. In our work, we tradeoff LazySP and LWA*, by introducing an event-based toggle [94, 93] that allows the search to proceed to an arbitrary horizon in the graph.

Several alternate approaches have been explored in the literature to address expensive collision checking. Some approaches model belief over the configuration space [61, 24] to plan in collision-free regions or sample vertices in promising regions [11, 16]. To reduce multiple collision checks in nearby regions of the configuration space, efficient datastructures such as safety certificates [12] have also been introduced. Orthogonally, some methods use specialized hardware [96] to eliminate the computation bottleneck of collision checking. Other approaches forego optimality and computing near-optimal paths [108, 34]. We note that our work can benefit from some of these approaches that seek to reduce the computational cost of collision checks.

Several works have explored the use of priors as auxilliary infor-

mation in search to report solutions faster. FuzzyPRM [98] evaluates paths that minimize the probability of collision. BISECT [25] and DIRECT [26] cast search as Bayesian active learning to derive edge evaluation policies. STROLL [10] learns an edge evaluation policy for LazySP. E-graphs [102] use priors in heuristics to speed up planning. The Anytime Edge Evaluation (AEE*) framework [97] uses an anytime strategy for edge evaluation informed by priors. POMP [24] defines surrogate objectives using priors to improve anytime planning. In this work, we focus on using priors to find the shortest path while minimizing expected planning time.

In our work, we draw inspiration from several prior works that interleave planning and execution such as RTA* [85], LSS-LRTA* [83]. These approaches plan up to a horizon (lookahead) and replan when new information is obtained. Similarly, in our work every edge evaluation is new information that the algorithm obtains from the environment. To handle such new information, efficient repair of search trees, if necessary, have previously been studied in literature [81, 2].

2.3 *Optimality in the Continuous Space*

While graphs lend tractability to the optimal motion planning problem, they can only guarantee resolution-optimality. Since the graph requires an a-priori selection of a resolution, it is possible that the graph is either too dense making search prohibitively expensive, or is too sparse to ensure existence of a solution. This limitation is addressed by algorithms which interleave generating increasingly accurate approximations of the continuous space with discrete graphs, and shortest path search on the constructed graphs [70, 72, 39, 40, 66]. These algorithms typically have probabilistic guarantees on computing an optimal solution. The algorithm is almost-surely asymptotically optimal if the probability of the algorithm computing the optimal solution approaches unity with the number of samples i.e.

$$P\left(\lim_{m\to\infty} c(\xi_m^*) = c(\xi^*)\right) = 1, \tag{2.3}$$

where m is the number of samples and ξ_m is the shortest path computed from the m samples.

PRM*, RRG and RRT* [70] are some of the seminal sampling-based algorithms that guaranteed asymptotic optimality with appropriate connection strategies [72, 70]. While these algorithms have enjoyed tremendous success in quickly computing an initial solution, and refining it in an anytime manner, they are generally slow in their convergence to optimality. They are slow because after an initial solution is computed, these algorithms spend significant planning time in sampling the statespace uniformly to determine a better solution.

The Informed Set While the statespace can be uniformly sampled to improve the initial solution, sampling can be limited to an *Informed Set* that contains only those states that can ever improve the current solution. Given a state $x \in \mathcal{X}_{\text{free}}$, a path between v_s and v_t constrained to pass through x has a cost bounded below by $c(\xi_{v_s,x,v_t}) = h(v_s, x) + h(x, v_t)$, where $h(.)$ is lower bound estimate of a path between two states in the statespace. Given the current best solution cost $c(\xi)$, the informed set [42] is defined as:

$$\mathcal{X}_{\text{inf}} = \{x \in \mathcal{X} \mid h(v_s, x) + h(x, v_t) \leq c(\xi)\} \tag{2.4}$$

For any state $x \notin \mathcal{X}_{\text{inf}}$, the lower bound estimate of the cost of a path constrained to go through it is greater than $c(\xi)$, and therefore does not help in convergence. For problems seeking to minimize path length in $\mathbb{R}^n$, the Euclidean distance is an admissible heuristic. The Informed Set can then be defined as:

$$\mathcal{E}_{\text{inf}} = \{x \in \mathcal{X} \mid \|x - v_s\|_2 + \|v_t - x\|_2 \leq c(\xi)\} \tag{2.5}$$

The Informed Set (Eq. 2.5) represents an $n-$dimensional prolate hyperspheroid $\mathcal{E}(v_s, v_t, c(\xi_i))$ (Figure 2.6) defined with foci v_s and v_t, a transverse axis diameter $c(\xi_i)$ with a measure $\lambda_{\mathcal{E}}$. Such an Informed Set can be uniformly sampled analytically [42]. Informed RRT* [39] and BIT* [40] showed that planning can be dramatically sped up by leveraging the Informed Set. We develop our asymptotically-optimal sampling-based algorithms (Chapters 5 and 6) over BIT*.

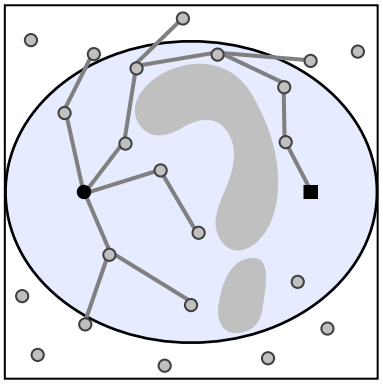


Figure 2.6: The Informed Set (blue) contains all the states that can ever improve the current solution. Sampling from the Informed Set is a sufficient condition to achieve optimality.

Improving convergence to optimality Incremental sampling-based algorithms interleave sampling with search. This observation has led to algorithms that improve search and sampling separately. While BIT* employs LWA* as its search algorithm, ABIT* [116] combines sampling with Anytime Truncated D* [1, 81, 2], AIT* [114] uses LazyPRM [14] as its search algorithm. Fast-BIT* [100] improves the time to compute its first solution by ordering the queues on cost-to-go but it does repeated search on the same approximation without reusing information. In Chapter 5, our work falls under this category where we study how *laziness* can improve the convergence to optimality. Several works have explored modifying the sampling distributions to improve the convergence rate. These algorithms focus sampling in regions that are hypothesized to contain high quality solutions. Most approaches discussed in Section 2.2.1 have proven to be effective in improving the convergence time. Our work in Chapter 6 explores non-uniform sampling of the Informed Set to further focus sampling and improve finite-time performance.

3
Generalized Lazy Search for Efficient Planning

In this chapter we consider the problem of computing the shortest path on a graph while minimizing planning time. Typically, the computational cost of evaluating whether an edge in the graph is collision-free dominates the running time of search algorithms. Lazy search algorithms have shown to efficiently solve problems where edge evaluation is the bottleneck in computation, as is the case for robotic motion planning. The optimal algorithm in this class, LazySP, lazily restricts edge evaluation to only the shortest path. Doing so comes at the expense of search effort, i.e., LazySP must *recompute the search tree* every time an edge is found to be invalid. This becomes prohibitively expensive when dealing with large graphs or highly cluttered environments. Our key insight is the need to balance both edge evaluation and search effort to minimize the total planning time. To this end, we propose Generalized Lazy Search (GLS), a framework that seamlessly toggles between search and evaluation to prevent wasted computational effort. We will show that for a choice of toggle, GLS is provably more efficient than LazySP. Subsequently, we leverage prior experience of edge probabilities to derive GLS policies that minimize expected planning time.

3.1 Introduction

We focus on the problem of finding the shortest path on a graph while minimizing total planning time. This is critical in applications such as robotic motion planning [90], where collision-free paths must be computed in real time. A typical search algorithm expands a wavefront from the start, evaluating collision status of edges discovered until it finds the shortest feasible path to the goal. The planning time then becomes the sum of the time spent in two phases – *search effort* and *edge evaluation*. While edge evaluation is generally more expensive in motion planning [54], the *actual ratio of these times varies* with problem instances and graph sizes. Our goal is to design a framework of algorithms that lets us balance this trade-off.

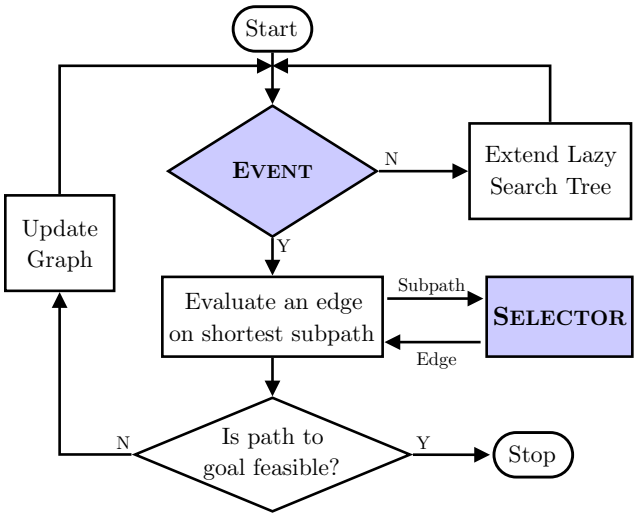


Figure 3.1: The Generalized Lazy Search (GLS) framework with two parameters - EVENT and SELECTOR (blue)

Unfortunately, current shortest path algorithms do not provide a framework flexible enough to traverse the pareto curve between search effort and edge evaluation. On one end of the spectrum, A* and its variants [52, 128, 84] evaluate edges as soon as they are discovered. Hence although A* is optimal in terms of *search effort*, it is at the cost of excessive edge evaluations. On the other hand, LAZYSP [31] amongst other lazy search techniques [14, 27, 54], expands the search wavefront all the way to the goal before evaluating edges. Hence LAZYSP is optimal in terms of edge evaluation but has to replan every time an edge is invalidated.

In this chapter, we propose a framework for *algorithmically toggling* between search effort and edge evaluation. We are guaranteed to find the shortest path as long as the following holds true; the search tree must always be repaired to be consistent, and edge evaluation must be restricted to the shortest subpath in the tree. Our framework, *Generalized Lazy Search* (GLS), has two modules - EVENT and SELECTOR (Fig. 3.1). The algorithm expands a lazy search tree without evaluating any edges till the EVENT is triggered. A SELECTOR is then invoked to evaluate an edge on the shortest *subpath* in the lazily expanded search tree. We show that by choosing different EVENT and SELECTOR pairs, we can recover several existing lazy search algorithms such as LAZYPRM [13], LWA* [27] and LAZYSP [31].

What constitutes an optimal trade-off and can this be captured by GLS? Consider the ideal scenario, one with an omniscient oracle [49] that knows ahead of time which edges are valid or invalid. In fact, the oracle can compute the minimal set of invalid edges $\mathcal{I}$ that must be invalidated to arrive at the shortest feasible path. How can we utilize such an oracle in GLS? A simple strategy is as follows; as the search

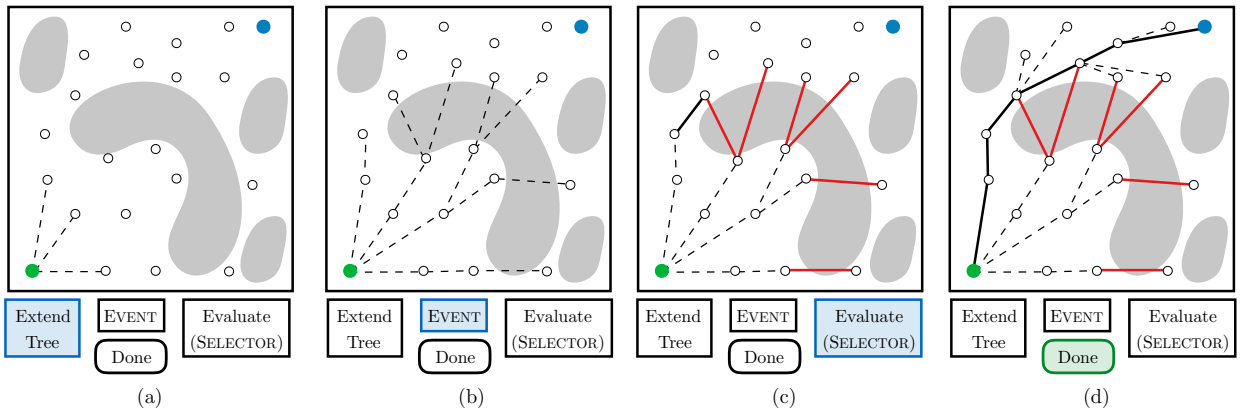


Figure 3.2: Mechanics of the GLS framework (Algorithm 1) for an ideal EVENT and SELECTOR combination.

wavefront expands from start to goal, the oracle monitors the new edges that are discovered and triggers an EVENT if it belongs to $\mathcal{I}$. A SELECTOR then evaluates that edge. This minimizes edge evaluation and curtails wasted search effort.

This insight extends to the more practical setting where we have *priors on edge validity* that are learned from experience. We derive EVENT and SELECTOR that minimize the expected planning time. This produces behaviors similar to the omniscient oracle (Fig. 3.2); the search proceeds until the EVENT is triggered due to the appearance of low probability edges on the current subpath; the SELECTOR then selects these edges to invalidate the subpath; and the process continues until the shortest feasible path is found.

3.2 *Problem Formulation*

Our goal is to design an algorithm that can solve the Single Source Shortest Path (SSSP) problem while minimizing computational effort. We begin with the SSSP problem. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ the set of edges. Given a pair of source and target vertices $\{v_s, v_t\} \in \mathcal{V}$, a path ξ is represented as a sequence of vertices $(v_1, v_2, \ldots, v_l)$ such that $v_1 = v_s, v_l = v_t, \forall i,\ (v_i, v_{i+1}) \in \mathcal{E}$. We define a *world* $\phi : \mathcal{E} \to \{0, 1\}$ as a mapping from edges to valid (1) or invalid (0). A path is said to be *feasible* if all edges are valid, i.e., $\forall e \in \xi, \phi(e) = 1$. Let $w : \mathcal{E} \to \mathbb{R}^+$ be the length of an edge. The length of a path is the sum of edge costs, i.e., $w(\xi) = \sum_{e \in \xi} w(e)$. The objective of the SSSP problem is to find the shortest feasible path:

$$\min_{\xi} \ w(\xi) \quad \text{s.t.} \quad \forall e \in \xi, \phi(e) = 1 \tag{3.1}$$

Given an SSSP, we define a shortest path algorithm $\text{ALG}(\mathcal{G}, v_s, v_t, \phi)$ that takes as input the graph $\mathcal{G}$, the source-target pair (v_s, v_t), and the

underlying world ϕ. The algorithm typically solves the problem by building, verifying and rewiring a shortest path tree from source to target. Maintaining the search tree and verifying the shortest feasible path are primarily characterized by two operations: edge evaluation and vertex rewiring.

Definition 3.2.1 (Edge Evaluation)**.** *The operation of querying the world $\phi(e)$ to check if an edge e is valid.*

Definition 3.2.2 (Vertex Rewiring)**.** *The operation of finding and assigning a new parent for a vertex u when an invalid edge is discovered.*

Edge evaluation involves the cost of querying the world. In domains like robotics, this includes intersection tests between the robot's swept volume along the edge and its environment. Vertex rewiring includes the cost of graph operations such as nearest neighbor computations, queue operations and updates to the search tree. We consider edge evaluations and vertex rewires as atomic operations in our problem formulation that are the source of computational cost.

The algorithm hence returns three terms, i.e, ξ^*, $\mathcal{E}_{\text{eval}}$, $\mathcal{V}_{\text{rwr}} = \textsc{Alg}(\mathcal{G}, v_s, v_t, \phi)$. Here, ξ^* is the shortest feasible path, $\mathcal{E}_{\text{eval}}$ is the *set* of edges evaluated during the search, and $\mathcal{V}_{\text{rwr}}$ is the *multiset*[1] of vertices rewired. Alg ensures the following certificate:

[1] $\mathcal{V}_{\text{rwr}}$ is a multiset since a vertex can potentially be rewired multiple times during the planning cycle.

1. The path ξ^* is verified to be feasible, i.e., $\forall e \in \xi^*,\ e \in \mathcal{E}_{\text{eval}},\ \phi(e) = 1$
2. All the paths shorter than ξ^* are verified to be infeasible, i.e., $\forall \xi_i,\ w(\xi_i) \leq w(\xi^*),\ \exists e \in \xi_i,\ e \in \mathcal{E}_{\text{eval}},\ \phi(e) = 0$

We now define the computational cost (planning time), of solving the SSSP problem as a function of $\mathcal{V}_{\text{rwr}}$ and $\mathcal{E}_{\text{eval}}$. Let c_e be the average cost of evaluating an edge, and c_r be the average cost of rewiring a vertex. We approximate the total planning time as a linear combination:

$$C(\mathcal{E}_{\text{eval}}, \mathcal{V}_{\text{rwr}}) = c_e \left|\mathcal{E}_{\text{eval}}\right| + c_r \left|\mathcal{V}_{\text{rwr}}\right| \tag{3.2}$$

Our motivation for defining the cost will become clearer in the following section, where we propose a general framework for Alg. This framework lets us explicitly reason about the terms $\mathcal{E}_{\text{eval}}$ and $\mathcal{V}_{\text{rwr}}$ in order to balance them.

3.3 *The Generalized Lazy Search Framework*

We propose a framework, Generalized Lazy Search (GLS), to find the shortest path while minimizing planning time. The underlying idea is to *toggle* between two processes – lazily searching the graph up to a horizon and evaluating the running estimate of the shortest path. The

Algorithm 1: Generalized Lazy Search

Input : Graph $\mathcal{G}$, source v_s, target v_t, world ϕ
Parameter: Heuristic $h(v, v_t)$, EVENT, SELECTOR
Output : ξ^*, $\mathcal{E}_{\text{eval}}$, $\mathcal{V}_{\text{rwr}}$

1 $\mathcal{E}_{\text{eval}} \leftarrow \emptyset, \mathcal{V}_{\text{rwr}} \leftarrow \emptyset$
2 $\mathcal{T}_{\text{lazy}} \leftarrow \{v_s\}$ ▷ Initialize
3 **repeat**
4 **repeat**
5 ExtendTree ($\mathcal{T}_{\text{lazy}}$, h)
6 **until** EVENT *($\mathcal{T}_{\text{lazy}}$) is true* ;
7 $\xi_{\text{sub}} \leftarrow$ subpath to the leaf vertex $\arg\min_v w(\xi_{v_s,v}) + h(v, v_t)$
8 Select edge to evaluate as $e \leftarrow$ SELECTOR(ξ_{sub})
9 Evaluate edge e by querying $\phi(e)$ ▷ Add to $\mathcal{E}_{\text{eval}}$
10 RewireTree ($\mathcal{T}_{\text{lazy}}$) ▷ Add to $\mathcal{V}_{\text{rwr}}$
11 **until** *shortest feasible path found s.t.* $\forall e \in \xi^*,\ e \in \mathcal{E}_{\text{eval}},\ \phi(e) = 1$;

deeper the search, the better the estimate of a *potential* shortest path, and hence the smaller the likelihood of wasted edge evaluations. On the other hand, a deeper search may result in significant backtracking and vertex rewiring should an invalid edge be discovered. Hence, varying the toggle allows GLS to track the pareto curve traced out by the competing computational costs of edge evaluation and vertex rewiring.

3.3.1 *The Algorithm*

Algorithm 1 describes a family of shortest path search algorithms ALG$(\mathcal{G}, v_s, v_t, \phi)$ defined in Section 3.2. There are three parameters of interest - a heuristic $h(v, v_t)$, an EVENT and a SELECTOR - each of which are explained below.

GLS solves the SSSP problem by growing a shortest path search tree $\mathcal{T}_{\text{lazy}}$ that encodes a path from source v_s to every discovered vertex v. It does so by interleaving two processes: *lazily* extending $\mathcal{T}_{\text{lazy}}$ assuming all unevaluated edges are valid, and evaluating edges on $\mathcal{T}_{\text{lazy}}$. The process terminates when $\mathcal{T}_{\text{lazy}}$ reaches the goal v_t, with all edges in the path being evaluated to be valid.

We initialize the search tree $\mathcal{T}_{\text{lazy}}$ at the source v_s (Line 2). We extend the tree by means of a heuristic guided search on graph $\mathcal{G}$, assuming any unevaluated edge to be feasible (Line 5). The heuristic $h(v, v_t)$ must be admissible [101] to ensure that $\mathcal{T}_{\text{lazy}}$ reaches v_t via a candidate shortest path.

We continue extending the tree until an event EVENT ($\mathcal{T}_{\text{lazy}}$) is triggered (Line 6).

Definition 3.3.1 (EVENT)**.** *A function that takes as input the lazy search tree $\mathcal{T}_{\text{lazy}}$ and returns a boolean denoting if search should be halted.*

The EVENT serves as a toggle between lazily extending the search tree and evaluating edges on the tree to validate it.

Every leaf vertex v of $\mathcal{T}_{\text{lazy}}$ represents a subpath $\xi_{v_s,v}$, i.e. a prefix of a potential path from source to target. We choose to validate the shortest subpath $\xi_{\text{sub}} = \xi_{v_s,v}$, i.e. the one with minimum sum of length $w(\xi_{v_s,v})$ and length-to-go $h(v, v_t)$ (Line 7). We do so by invoking a selector SELECTOR(ξ_{sub}) that specifies which edge to evaluate (Line 8).

Definition 3.3.2 (SELECTOR)**.** *A function that takes as input a subpath ξ_{sub} and returns which edge e to evaluate.*

The selected edge e is evaluated by querying the world $\phi(e)$ (Line 9). We add this edge to the set $\mathcal{E}_{\text{eval}}$ to keep track of how many edges were evaluated. If the edge is found to be invalid, the subtree in $\mathcal{T}_{\text{lazy}}$ emanating from e is no longer valid and has to be rewired (Line 10). We add these vertices to $\mathcal{V}_{\text{rwr}}$ to keep track of how many vertices are needed to be rewired.

This process of interleaving search with edge evaluation continues until the algorithm terminates with the shortest feasible path from source to goal, if one exists. While the algorithm is guaranteed to return the shortest path, the framework permits the design of EVENT and SELECTOR to reduce the total computation cost which in turn depends on $|\mathcal{E}_{\text{eval}}|$ and $|\mathcal{V}_{\text{rwr}}|$.

The implementation details of GLS can be found in Appendix A.

3.3.2 Roles of Heuristic, Event and Selector

Since the lazy search paradigm operates based on the concept of optimism under uncertainty, the search tree is extended assuming edges are collision-free. Under this paradigm, three components define the behavior of search and evaluation, and the corresponding computation cost. The Heuristic is crucial in propagating the search tree in promising directions. However, extending the search tree, even in promising directions, beyond edges that are in collision can waste computational effort. The EVENT acts as a toggle to halt a search deemed wasteful. The SELECTOR aims to quickly invalidate paths to drive the search to feasible regions. In the following subsections, we discuss the role each component plays in the framework of GLS.

Algorithm 2: Candidate EVENT Definitions

$v \leftarrow$ leaf vertex in $\mathcal{T}_{\text{lazy}}$ with least estimated cost to v_t

Function `ShortestPath()`
  if $v = v_t$ **then**
    return true;
Function `ConstantDepth(`*depth* α`)`
  $\xi_{\text{sub}} \leftarrow$ path from v_s to v
  $\alpha_v \leftarrow$ number of unevaluated edges in ξ_{sub}
  if $\alpha_v = \alpha$ ***or*** $v = v_t$ **then**
    return true;
Function `HeuristicProgress`
  $h_{\min} \leftarrow \min_{(u',v') \in \mathcal{E}_{\text{eval}}} h(v', v_t)$
  if $h(v, v_t) < h_{min}$ ***or*** $v = v_t$ **then**
    return true;
  return;
Function `SubpathExistence(`*probability* δ`)`
  $\xi_{\text{sub}} \leftarrow$ path from v_s to v
  $p \leftarrow \prod_{e \in \sigma} \mathbf{p}(e)$
  if $p \leq \delta$ ***or*** $v = v_t$ **then**
    return true;

Algorithm 3: Candidate SELECTOR Definitions

Function `Forward()`
  return {first unevaluated edge closest to v_s };
Function `Alternate()`
  if *Iteration Number is Odd* **then**
    return {first unevaluated edge closest to v_s };
  else
    return {first unevaluated edge closest to v_t};
Function `FailFast()`
  return $\{\arg\min_{e \in \xi_{\text{sub}}} \mathbf{p}(e)\}$;

Heuristic As noted in Section 3.3.1, GLS extends the lazy search tree by means of a heuristic-guided best-first search on the graph. As in A*, if the heuristic is admissible (an underestimate or equal to the true cost), then the set of vertices expanded during search is both necessary and sufficient to compute the shortest path.

Since the framework of GLS relies on toggling between search and evaluation, it is important that the heuristic guides the search tree construction in promising directions to allow EVENT and SELECTOR to

consider fewer subpaths for evaluation. In our exposition, we mainly employ two heuristics. Distance on the unevaluated graph, $h_{\mathcal{G}}(v, v_t)$, measures the length of the shortest path from a given vertex v to the goal v_t assuming all edges are valid. Euclidean heuristic, $h_e(v, v_t)$ measures the Euclidean distance between configurations corresponding to given vertex v and goal v_t.

EVENT When triggered, events must ensure that the shortest subpath ξ_{sub} in $\mathcal{T}_{\text{lazy}}$ has at least one unevaluated edge (Theorem 3.3.1). Algorithm 2 defines some candidate events.

SHORTESTPATH (SP) is triggered when a shortest path to v_t has been determined during the lazy extension of $\mathcal{T}_{\text{lazy}}$. Therefore, in every iteration, this EVENT presents the SELECTOR with the candidate shortest path from v_s to v_t on $\mathcal{G}$. Note that SHORTESTPATH exhibits algorithmic behavior similar to LAZYPRM and LAZYSP.

CONSTANTDEPTH (CD) is triggered when the procedure EXTENDTREE chooses to extend a leaf vertex $v \in \mathcal{T}_{\text{lazy}}$ such that the subpath from v_s to v has exactly α number of unevaluated edges. Therefore, in every iteration, this EVENT presents the SELECTOR with ξ_{sub} that is characterized by a constant number of unevaluated edges. We introduced Lazy Receding-Horizon A* (LRA*) [94] as a precursor to GLS [93] where the EVENT was specifically CONSTANTDEPTH.

HEURISTICPROGRESS (HP) is triggered whenever the search expands a vertex whose heuristic value is lower than any vertex whose incident edge has been evaluated. It does so by recording the minimum heurisitic value of a vertex with a parent that has been evaluated, i.e., $h_{\min} \leftarrow \min_{(u',v') \in \mathcal{E}_{\text{eval}}} h(v', v_t)$. The event is triggered whenever EXTENDTREE chooses to extend a leaf vertex $v \in \mathcal{T}_{\text{lazy}}$ with a heuristic value smaller than $h_{\min}$.

SUBPATHEXISTENCE defined in Algorithm 2 is introduced and analyzed in Section 3.4.

SELECTOR. Selectors must ensure that they select at least one unevaluated edge (Theorem 3.3.1). Algorithm 3 defines some candidate selectors.

Given ξ_{sub}, FORWARD (F) evaluates the first unevaluated edge on ξ_{sub} that is closest to v_s. Given a forward search, this constitutes one of the most natural SELECTORS available. ALTERNATE (A) toggles between evaluating the first unevaluated edge closest to v_s and v_t in every iteration. This approach is motivated by bi-directional search algorithms. Both SELECTORS were first used in [31].

FAILFAST defined in Algorithm 3 is introduced and analyzed in Section 3.4.

3.3.3 *Recovering existing algorithms with* GLS

The generality of GLS with its definitions of EVENTS and SELECTORS is noted in its ability to capture several existing algorithms from lazy paradigm of search (see Table 3.1). LAZYSP [31] and LAZYPRM [14] extend the search tree to the goal before toggling to evaluation, and therefore employ SHORTESTPATH by definition. Lazy Weighted A* (LWA*) [27] and Lazy Receding-Horizon A* (LRA*)[2] [94] toggle between search and forward evaluation when the lazy search tree is extended up to a certain depth (of 1 in LWA* and a given depth α in LRA*), and therefore are defined by CONSTANTDEPTH EVENT with a FORWARD SELECTOR.

[2] Introduced as part of this thesis

Algorithm	EVENT	SELECTOR
LAZYPRM [14]	SHORTESTPATH	Any
LAZYSP [31]	SHORTESTPATH	Any
LWA* [27]	CONSTANTDEPTH (1)	FORWARD
LRA* [94]	CONSTANTDEPTH (α)	FORWARD

Table 3.1: Existing lazy algorithms as instantiations of GLS

3.3.4 *Analysis*

We formally analyze GLS. We first show that it is complete and correct. We then analyze the effect of varying heuristic $h()$ and EVENT on the amount of edge evaluations and vertex rewirings. Specifically, we show one can vary parameters to trade-off the two competing computational costs thus making GLS applicable across a number of domains. Using this analysis, we show that existing state-of-the-art algorithm LAZYSP [31] is not pareto-optimal. Instead, we derive an instance of GLS that rewires fewer vertices than LAZYSP while minimizing edge evaluations.

Completeness, Correctness and Complexity For any choice of EVENT and SELECTOR, GLS is complete and correct.

Theorem 3.3.1 (Completeness)**.** *Let* EVENT *be a function that on halting ensures there is at least one unevaluated edge on the current shortest path or that the goal is reached. Let* SELECTOR *be a function that evaluates at least one unevaluated edge (if it exists).* GLS *instantiated using* EVENT *and* SELECTOR *on a finite graph $\mathcal{G}$ is complete.*

Theorem 3.3.2 (Correctness)**.** *If the heuristic $h(v, v_t)$ is admissible, then* GLS *terminates with the shortest feasible path.*

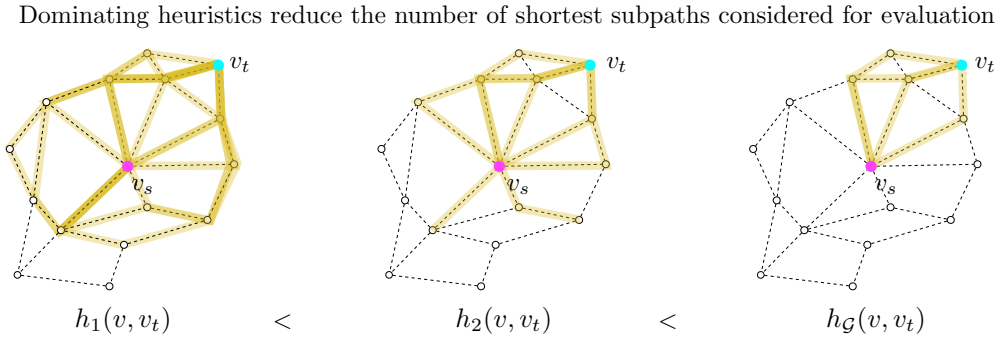


Figure 3.3: Visualization of the subpaths (in yellow, originating from v_s) that are considered for evaluation during search when using different heuristics

Proposition 3.3.1 (Correctness). *Consider* GLS *with any* SELECTOR *and any* EVENT. *If the heuristic $h(v, v_t)$ is consistent, and the* SELECTOR *evaluates an edge along a path $\xi_{v_s,v}$ such that all edges along $\xi_{v_s,v}$ have been evaluated to be feasible, then $\xi_{v_s,v}$ is the shortest feasible path to v, i.e., $w(\xi_{v_s,v}) = w^*(v)$.*

Corollary 3.3.1 (Correctness). *Consider* GLS *with* FORWARD *and any* EVENT. *If the heuristic $h(v, v_t)$ is consistent, and* FORWARD *evaluates an edge to v to be feasible, then the path from source v_s to v, $\xi_{v_s,v}$ is the shortest feasible path to v, i.e., $w(\xi_{v_s,v}) = w^*(v)$.*

Theorem 3.3.3 (Complexity). *Let α be the maximum depth of a lazy subtree $\mathcal{T}_{\text{lazy}}$ for a given* EVENT. *The total running time of the algorithm is bounded by $O(nd^{\alpha} \cdot \log(n) + m)$, where n and m are the number of vertices and edges in $\mathcal{G}$, and d is the maximal degree of a vertex.*

Informative heuristic reduces edge evaluation We now study the effect of heuristic on the computation time. Specifically, we show that as heuristics get tighter at approximating the length-to-go, the number of edges evaluated reduce. We first define a notion of tightness between two consistent heuristics as follows:

Definition 3.3.3 (Dominating heuristics). *Given two consistent heuristics h_1 and h_2, h_1 is said to strictly dominate h_2 if*

$$h_1(v, v_t) > h_2(v, v_t) \quad \forall v \in \mathcal{V}, v \neq v_t \tag{3.3}$$

Theorem 3.3.4. *Consider two instantiations of* GLS *with heuristics h_1 and h_2. Both use the selector* FORWARD. *Both use the same* EVENT. *Let $\mathcal{E}_{\text{eval},1}$ and $\mathcal{E}_{\text{eval},2}$ be the edges evaluated respectively. If h_1 strictly dominates h_2, then it evaluates a subset of edges, i.e. $\mathcal{E}_{\text{eval},1} \subseteq \mathcal{E}_{\text{eval},2}$.*

This shows that the more dominating a heuristic or tighter the approximation in length-to-go, the fewer the edge evaluations before termination.

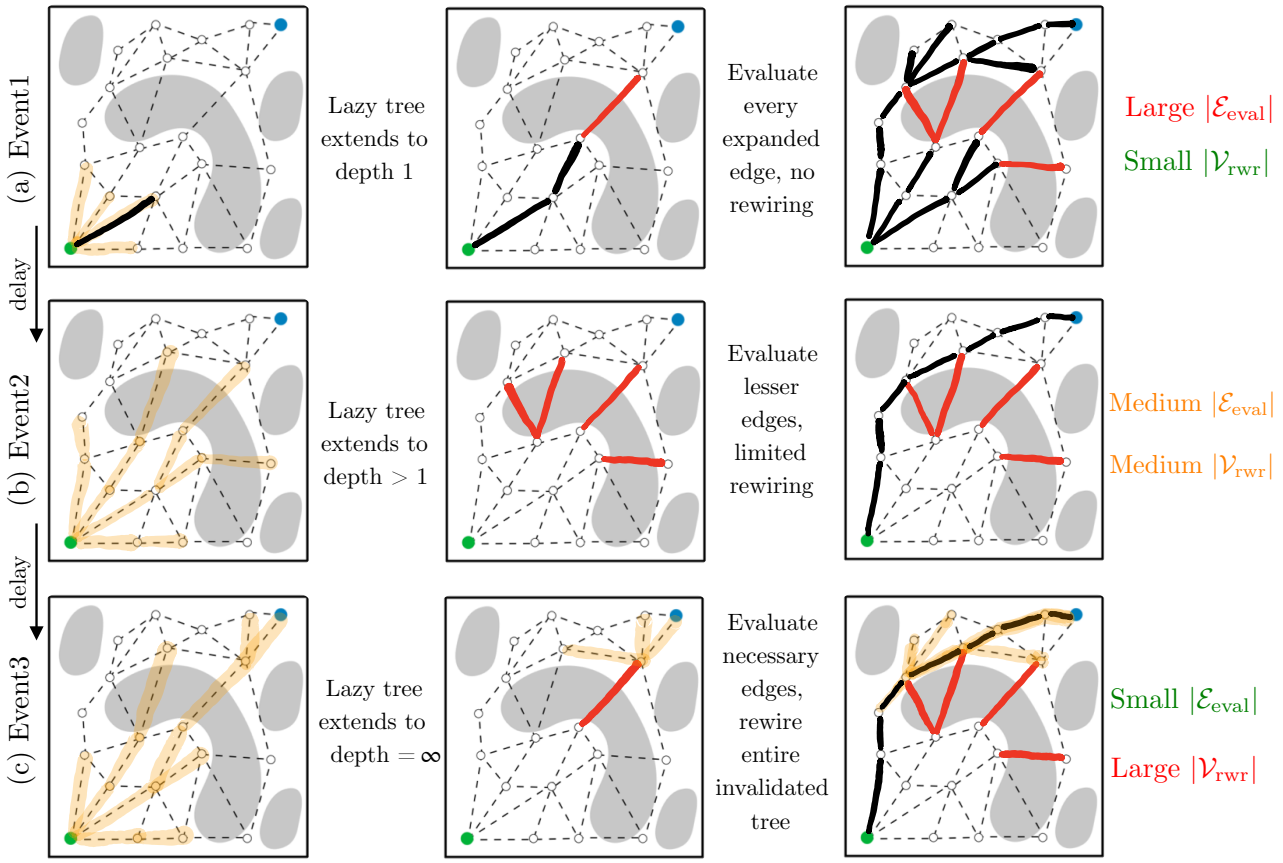


Figure 3.4: Snapshots of search and evaluation by EVENTS with increasing delays (top to bottom). Edges evaluated to be valid (black), invalid (red) and lazy search tree (yellow) are shown. As the EVENT is delayed, edge evaluations decrease while vertex rewires increase. EVENT2 (middle) minimizes neither cost but captures a balance to reduce total computational cost.

Delayed EVENT *reduces edge evaluations* $|\mathcal{E}_{\text{eval}}|$ As noted in Section 3.3.2, the EVENT toggles between search and evaluation to reduce wasted search effort. In this section we study how this toggle also effects the number of edge evaluations and hence the computational time. Specifically, we show that the more *delayed* an EVENT, the fewer the edge evaluations (Fig. 3.4). We first define the notion of *delay* in an EVENT:

Definition 3.3.4 (Delayed EVENT)**.** *Let* EVENT *be a class of functions that maps the current shortest subpath* ξ_{sub} *in* $\mathcal{T}_{\text{lazy}}$ *to a boolean flag. Let* ξ_1 *be a subpath that triggers an event* EVENT1*. Let* ξ_2 *be a prefix of* ξ_1*, i.e.,* $\xi_2 \subseteq \xi_1$*. An event* EVENT1 *is said to be a delayed w.r.t* EVENT2 *if there exists a prefix* ξ_2 *such that* EVENT1(ξ_1) *being true implies* EVENT2(ξ_2) *being true.*

Theorem 3.3.5. *Consider two instantiations of* GLS*. Let* EVENT1*,* EVENT2 *be the two events such that* EVENT1 *is delayed w.r.t* EVENT2*. Both use the* FORWARD SELECTOR*. Let* $\mathcal{E}_{\text{eval},1}$ *and* $\mathcal{E}_{\text{eval},2}$ *be the edges evaluated respectively. Then the delayed event evaluates a subset of edges, i.e.* $\mathcal{E}_{\text{eval},1} \subseteq \mathcal{E}_{\text{eval},2}$.

Corollary 3.3.2 (Edge Optimality)**.** *Given* FORWARD SELECTOR, GLS *with* SHORTESTPATH EVENT *is edge optimal.*

Given FORWARD SELECTOR, from Theorem 3.3.5 we obtain that SHORTESTPATH EVENT, being most *delayed* by definition, minimizes the number of edge evaluations. In other words, within the class of algorithms that use FORWARD SELECTOR, there is no other algorithm that terminates having evaluated fewer edges than does LAZYSP (FORWARD) or equivalently GLS (SHORTESTPATH, FORWARD).

Delayed EVENT *increases vertex rewirings* $|\mathcal{V}_{\text{rwr}}|$ Section 3.3.2 notes that the EVENT acts as a toggle between lazy search and evaluation to reduce wasted search effort. In this section we formally characterize how the definition of an EVENT effects the number of vertex rewires and hence the computational time. Specifically, we show that a more delayed EVENT exhibits greater number of vertex rewires (Fig. 3.4).

Theorem 3.3.6. *Let* Ξ *be the set of paths that are shorter than* ξ^* *and infeasible.*

Let EVENT1, EVENT2 *be sufficiently delayed such that they trigger only when the shortest subpath* $\xi_{\text{sub}} \in \mathcal{T}_{\text{lazy}}$ *is a prefix to a path* $\xi \in \Xi$*. Given* FORWARD*, they both evaluate optimal number of edges* $\mathcal{E}^*_{\text{eval}}$*. Let* $\mathcal{V}_{\text{rwr,1}}$ *and* $\mathcal{V}_{\text{rwr,2}}$ *be the vertex rewirings respectively.*

If EVENT1 *is delayed w.r.t* EVENT2*, the vertex rewiring of the delayed event is more, i.e.* $|\mathcal{V}_{\text{rwr,1}}| \geq |\mathcal{V}_{\text{rwr,2}}|$.

A point on the pareto frontier of $|\mathcal{E}_{\text{eval}}|$ *vs* $|\mathcal{V}_{\text{rwr}}|$ Theorem 3.3.5 and Corollary 3.3.2 state that LAZYSP with the FORWARD selector is *edge optimal* in the class of all shortest path algorithms that use a FORWARD selector. We now show that GLS lets us derive another algorithm that is *also edge-optimal* but reduces number of vertex rewires.

Theorem 3.3.7. GLS *evaluates the same number of edges* $|\mathcal{E}_{\text{eval}}|$ *as* LAZYSP*, i.e., is edge optimal, while having a smaller number of vertices rewired* $|\mathcal{V}_{\text{rwr}}|$ *under the following setting:*

1. *Heuristic: Distance on the unevaluated graph* $h_{\mathcal{G}}(v, v_{\text{t}})$
2. EVENT*:* HEURISTICPROGRESS
3. SELECTOR*:* FORWARD

Corollary 3.3.3. *There exists a graph* $\mathcal{G}$ *for which the number of vertex rewires* $|\mathcal{V}_{\text{rwr}}|$ *for* LAZYSP *over* GLS *is linear over logarithmic.*

Greediness in edge evaluations by SELECTOR From Algorithm 1 (Line 8) and Definition 3.3.2, we only evaluate a single edge given a subpath ξ_{sub}. However, we can choose to evaluate more than one edge, hence performing an exploitative action, while still maintaing correctness and completeness. This introduces a parameter β that indicates how many edges to evaluate along the path. However, we can show that our current formulation using a minimal greediness value of $\beta = 1$ always outperforms any other greediness value in the number of edge evaluations. This is only the case when we seek optimal paths. If we relax the algorithm to produce bounded-suboptimal paths, greediness may be of use in early termination. While we continue to constrain selection of a single edge in the rest of the exposition, we provide proofs pertaining to the superiority of no greediness for the case that optimal paths are required in Appendix A.

Summary We seek choices of EVENT, SELECTOR and Heuristic that achieve minimal edge evaluations and minimal vertex rewiring. In this section we discussed the effects of EVENT and Heuristic on the computational cost of GLS for a given SELECTOR as summarized in Fig. 3.5.

1. Delayed EVENT leads to fewer edge evaluations at the cost of more vertex rewiring.
2. Dominant Heuristic leads to fewer edge evaluations.
3. A combination of EVENT, SELECTOR and Heuristic can reduce edge evaluations and vertex rewiring.

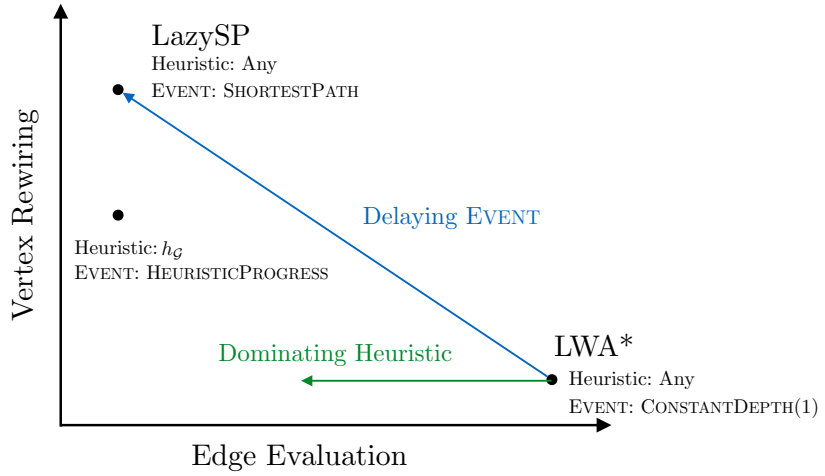


Figure 3.5: For a fixed SELECTOR, effect of EVENT and Heuristic on computational cost

In summary, an EVENT creates a trade-off between the number of edge evaluation and vertex rewires. A more delayed EVENT reduces

the number of edge evaluations (Theorem 3.3.5) but inadvertantly increases the number of vertex rewires (Theorem 3.3.6). This trade-off is further affected by the approximation of length-to-go or the heuristic. As seen in Theorem 3.3.4, tighter the approximation, fewer the number of edge evaluations. This suggests that EVENT and heuristics together provide a mechanism to balance the costs of search (vertex rewires) and edge evaluations to minimize the total computation cost of solving the SSSP.

3.4 *Leveraging Edge Priors in* GLS

The GLS framework is powerful because one can optimize EVENT and SELECTOR to minimize computational costs while still retaining guarantees. Here, we show its expressive power in a scenario where we have auxilliary side information, such as priors on the validity of edges. Such priors can be computed offline from datasets of previously encountered planning problems or online from a low resolution approximation of obstacle representation.

3.4.1 *Optimal* EVENT *and* SELECTOR

What is the lower bound on the best an EVENT and SELECTOR can do? We answer this by conjuring a clairvoyant oracle that has full knowledge about the world $\phi(e)$, i.e., knows which edges are valid or invalid. The oracle is also free to compute any terms without cost. However, it is compelled to act through the interface of EVENT and SELECTOR.

We first bound the minimum number of edges that need to be evaluated to be invalid till the shortest path can be found. We show that this is equivalent to solving a min cover problem where all candidate shortest paths to be eliminated need to be covered.

Theorem 3.4.1. *Let* ALG *be a shortest path algorithm such that* $\text{ALG}(\mathcal{G}, v_s, v_t, \phi)$ *returns shortest path* ξ^**, edges evaluated* $\mathcal{E}_{\text{eval}}$ *and vertices rewired* $\mathcal{V}_{\text{rwr}}$*. For all such* ALG*, the number of edges evaluated* $|\mathcal{E}_{\text{eval}}|$ *is lower bounded by*

$$\begin{aligned} \min_{\mathcal{E}_{\text{eval}}} \quad & |\mathcal{E}_{\text{eval}}| \\ s.t. \quad & \forall e \in \mathcal{E}_{\text{eval}},\ \phi(e) = 0 \\ & \forall \xi_i \in \{\xi \mid w(\xi) \leq w(\xi^*)\},\ \xi_i \cap \mathcal{E}_{\text{eval}} \neq \emptyset \end{aligned} \tag{3.4}$$

We now define the optimal combination of heuristic, EVENT and SELECTOR that minimizes total planning time as defined in (3.2).

1. Heuristic: Distance on the unevaluated graph, $h_{\mathcal{G}}(v, v_t)$
2. EVENT: Returns true if ξ_{sub} contains an edge $e \in \mathcal{E}_{\text{eval}}$.

3. SELECTOR: Selects $e \in \mathcal{E}_{\text{eval}}$

This will result in 0 rewires and $\min |\mathcal{E}_{\text{eval}}|$ amount of edge evaluation (in addition to evaluating the feasible edges on the optimal path). The heuristic $h_{\mathcal{G}}(v, v_{\text{t}})$ ensures that the lazy search tree $\mathcal{T}_{\text{lazy}}$ iterates over paths belonging to $\Xi = \{\xi_i \mid w(\xi_i) \leq w(\xi^*)\}$. Any time an edge $e \in \mathcal{E}_{\text{eval}}$ is encountered, the EVENT is triggered. The invalid edge is evaluated which does not result in any rewiring as the edge is at the leaf. The process continues till another edge in $e \in \mathcal{E}_{\text{eval}}$ is encountered. In this fashion, the algorithm proceeds to eliminate all paths in Ξ with minimal edge evaluation and zero rewiring.

3.4.2 *Modified Formulation: Minimize Expected Planning Time*

How does the clairvoyant oracle help in practice when we do not know the world ϕ in advance? Let's assume instead that we have a prior $P(\phi)$ from which worlds are sampled. We adopt a modified version of the problem defined in Section 3.2 by choosing to minimize the *expected* planning time. Our goal is to design a shortest path algorithm $\text{ALG}(\mathcal{G}, v_{\text{s}}, v_{\text{t}}, \phi, P(\phi))$ that leverages knowledge of the prior $P(\phi)$ to minimize the following objective:

$$\begin{aligned} \min \quad & \mathbb{E}_{\phi \sim P(\phi)} \left[c_e \left| \mathcal{E}_{\text{eval}} \right| + c_r \left| \mathcal{V}_{\text{rwr}} \right| \right] \\ \text{s.t.} \quad & \mathcal{E}_{\text{eval}},\ \mathcal{V}_{\text{rwr}} = \text{ALG}(\mathcal{G}, v_{\text{s}}, v_{\text{t}}, \phi, P(\phi)) \end{aligned} \tag{3.5}$$

We know that when the prior $P(\phi)$ is peaked about a single world ϕ, the solution is the pair of clairvoyant oracular EVENT and SELECTOR described in Section 3.4.1. When the prior is more diffused, we design EVENT and SELECTOR that approximate the oracles.

We focus on a specific class of priors $P(\phi)$ where each edge is an independent Bernoulli random variable. We are given a vector of probabilities $\mathbf{p} \in [0,1]^{|\mathcal{E}|}$, such that $P(\phi(e) = 1) = \mathbf{p}(e)$, i.e., for each edge e, we have access to $\mathbf{p}(e)$, which defines the probability of the edge being valid in the current world ϕ. Even though the problems we look at are better modelled by random variables that are actually correlated, the Bernoulli assumption serves as a sufficiently useful approximation that is compact and easy to optimize.

3.4.3 EVENT *design*

A simple, inexpensive approximation of Equation 3.4 is to simply estimate the set of invalid edges $\mathcal{E}_{\text{inv}}$. An EVENT can then try to estimate the probability of the shortest subpath being valid and trigger when this probability drops below a threshold δ.

$$P(\forall e \in \xi | \phi(e) = 1) \leq \delta \tag{3.6}$$

In the case of independent Bernoulli edges, this is equivalent to $\prod_{e \in \sigma} \mathbf{p}(e) \leq \delta$. Intuitively, the EVENT restricts lazy search from proceeding beyond a point when the search is likely to be ineffective, i.e. to a point that potentially increases the amount of rewires $\mathcal{V}_{\text{rwr}}$. We describe this event, SUBPATHEXISTENCE (SE), in Algorithm 2. We now analyze the performance of this event.

Theorem 3.4.2. *Given* SUBPATHEXISTENCE *(δ), any* SELECTOR *and distance on the unevaluated graph $h_{\mathcal{G}}(v, v_t)$ as heuristic, the expected planning time of* GLS *can be bounded above by:*

$$K \left(c_e \frac{1}{(1-\delta)} + c_r b L(\delta) \right) \tag{3.7}$$

where K is the number of paths shorter than ξ^* but infeasible, b is the maximum branching factor, and $L(\delta)$ is the maximum length of an unevaluated subpath before the event SUBPATHEXISTENCE *(δ) is triggered.*

We apply Theorem 3.4.2 to bound the planning time for the case of Bernoulli priors $\mathbf{p}(e)$.

Corollary 3.4.1. *Given* SUBPATHEXISTENCE *(δ), any* SELECTOR*, a Bernoulli prior $\mathbf{p}(e)$ and distance on the unevaluated graph $h_{\mathcal{G}}(v, v_t)$ as heuristic, the expected planning time of* GLS *can be bounded above by:*

$$K \left(c_e \frac{1}{(1-\delta)} + c_r \frac{b \log(\delta)}{\log(p_{max})} \right) \tag{3.8}$$

where K is the number of paths shorter than ξ^* but infeasible, b is the maximum branching factor, and $p_{max} = \max_e \mathbf{p}(e)$ is the maximum value of an edge prior.

We observe that low values of δ result in lower edge evaluations cost but higher vertex rewiring cost, and vice-versa. From Equation 3.8, we can obtain the δ that minimizes the upper bound on the expected computational cost:

Theorem 3.4.3. *Given a Bernoulli prior $\mathbf{p}(e)$, there exists a critical threshold $\delta \in (0, 1)$ that minimizes the upper bound on the expected computational cost*

$$\delta \approx \begin{cases} 1, & c_e \ll c_r \\ \frac{1}{\left(\frac{c_e}{b c_r} \log(\frac{1}{p_{max}}) + 2 \right)} & c_e \gg c_r \end{cases} \tag{3.9}$$

3.4.4 SELECTOR *design*

We reuse the same approximation of Equation 3.4 as the set of invalid edges $\mathcal{E}_{\text{inv}}$ to design a SELECTOR. The selector tries to select the edge that is most likely to belong to $\mathcal{E}_{\text{inv}}$

$$\underset{e \in \xi_{\text{sub}}}{\arg\min}\ P(e) \tag{3.10}$$

In the case of independent Bernoulli edges, this is equivalent to selecting $\arg\min_{e \in \xi_{\text{sub}}} \mathbf{p}(e)$. We describe this selector, FAILFAST (FF), in Algorithm 3. Intuitively, the selector tries to eliminate a subpath as quickly as possible. We show that this selector can do this near-optimally for general distributions and optimally for the case of independent Bernoulli edges.

Theorem 3.4.4. *Given a path ξ,* FAILFAST *is within a factor of 4 of the optimal expected number of edges from ξ that must be evaluated to invalidate ξ.*

However, for the case of Bernoulli prior, FAILFAST is in fact optimal.

Theorem 3.4.5. *Given a Bernoulli prior $\mathbf{p}(e)$ and a path ξ,* FAILFAST *minimizes the expected number of edges from ξ that must be evaluated to invalidate ξ.*

3.5 Experiments and Analysis

We evaluate the utility of toggling between lazy search and evaluation of edges, to solve the shortest path problem. We demonstrate GLS on simulated problems in $\mathbb{R}^2$, Piano Movers' problem in $SE(2)$ and manipulation problems in $\mathbb{R}^7$ using HERB [112], a mobile robot with 7DoF arms[3]. We report our results of GLS with (a) number of edge evaluations, $|\mathcal{E}_{\text{eval}}|$ (b) number of vertex rewires $|\mathcal{V}_{\text{rwr}}|$ and (c) total planning time as the evaluation metrics. Since $\mathbb{R}^2$ problems are *not expensive to evaluate*, we consider planning time to be a weighted combination of (a), (b) (see Eq. 3.2). We choose weights based on empirical data from manipulation planning problems in $\mathbb{R}^7$ (avg. eval time: 3.35×10^{-4}s, avg. rewire time 1.1×10^{-5}s, ratio 29.04)[4].

[3] Code is publicly available as an OMPL Planner at: `https://github.com/personalrobotics/gls`

[4] Simulations were run on a desktop machine with 16GB RAM and an Intel i5-6600K processor running a 64-bit Ubuntu 18.04

Building upon the theoretical analysis of EVENTS and SELECTORS in Section 3.3.4, we setup our experiments to test related hypotheses.

H 1. *For any* SELECTOR, *the event* SUBPATHEXISTENCE *requires less planning time compared to* SHORTESTPATH *and* CONSTANTDEPTH.

This follows from Theorem 3.4.2 and Lemma 3.4.1 which upper bound the planning time for SUBPATHEXISTENCE. SHORTESTPATH corresponds to a fixed $\delta = 0$, a consistently delayed event, and can exhibit higher planning times from edge evaluations (see Theorem 3.3.5). CONSTANTDEPTH has a fixed horizon by definition and does not adapt even with auxilliary information available such as priors. In constrast, SUBPATHEXISTENCE is characterized by the ability to leverage priors and adapt the lazy search horizon in sparse and constrained regions of the configuration space as necessary.

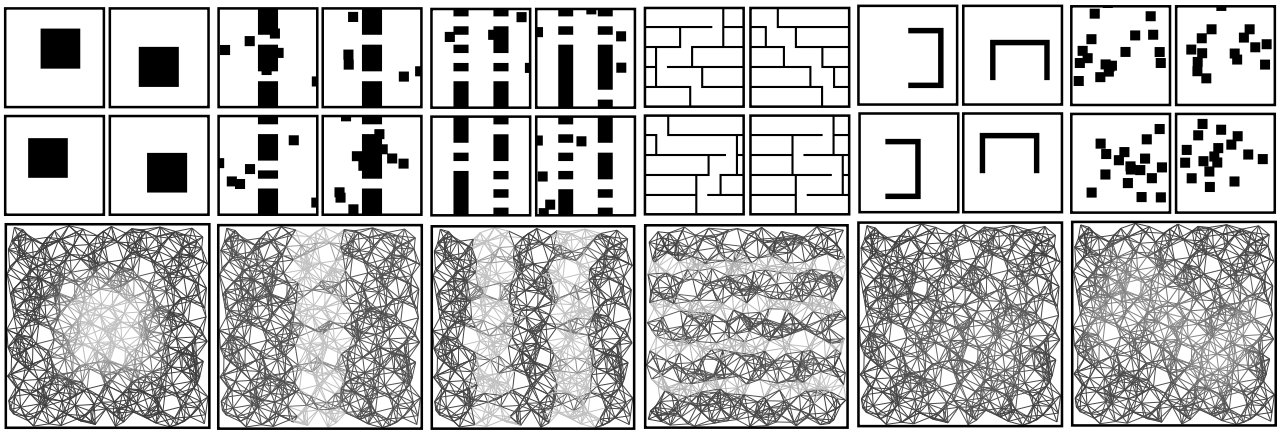


Figure 3.6: Samples (top) and priors (bottom) for $\mathbb{R}^2$ datasets of each environment type. Columnns left to right: Square, OneWall, TwoWall, Maze, BugTrap and Forest.

H 2. *For any* Event, *the selector* FailFast *evaluates fewer edges than* Forward *and* Alternate.

This follows from Theorem 3.4.5 which shows that FailFast is optimal in expectation for eliminating a path. From **H 1** and **H 2**, we hypothesize that GLS with SubpathExistence and FailFast will exhibit lowest planning times.

H 3. *The performance gain of* SubpathExistence *over* ShortestPath *increases with both graph size and problem difficulty.*

ShortestPath assumes that $\mathcal{V}_{\text{rwr}}$ is negligible. However, as graph size increases, $|\mathcal{V}_{\text{rwr}}|$ i.e. the number of vertices that ShortestPath rewires also increases. Similarly, as problem difficulty increases, so does the number of shortest paths that ShortestPath must invalidate before terminating with the collision-free shortest path, thus increasing $|\mathcal{V}_{\text{rwr}}|$. SubpathExistence, on the other hand, makes no such assumption. Informed by auxilliary information of priors, SubpathExistence adapts the search horizon to balance the computational costs of evaluation, $|\mathcal{E}_{\text{eval}}|$ and rewires, $|\mathcal{V}_{\text{rwr}}|$.

3.5.1 Environments and Priors

We consider environments in $\mathbb{R}^2$, $SE(2)$ and $\mathbb{R}^7$. For each of these environments, we construct accompanying graphs to test GLS. Each roadmap was constructed as follows: The set of vertices were generated in a unit hypercube using Halton sequences [51], characterized by low dispersion, and scaled appropriately. An edge existed in the graph between every pair of vertices whose Euclidean distance is less than a predefined threshold r. The value r was chosen to ensure that, asymptotically, the graph can capture the shortest path connecting the start to the goal [67]. The number of vertices was chosen such that the

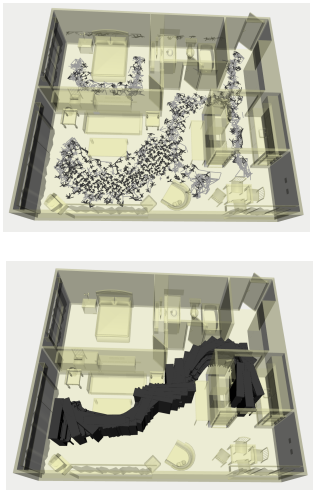


Figure 3.7: (a) Prior over validity of edges of a graph. Edges with higher prior over validity are visualized in a darker shade. (b) Sample shortest path.

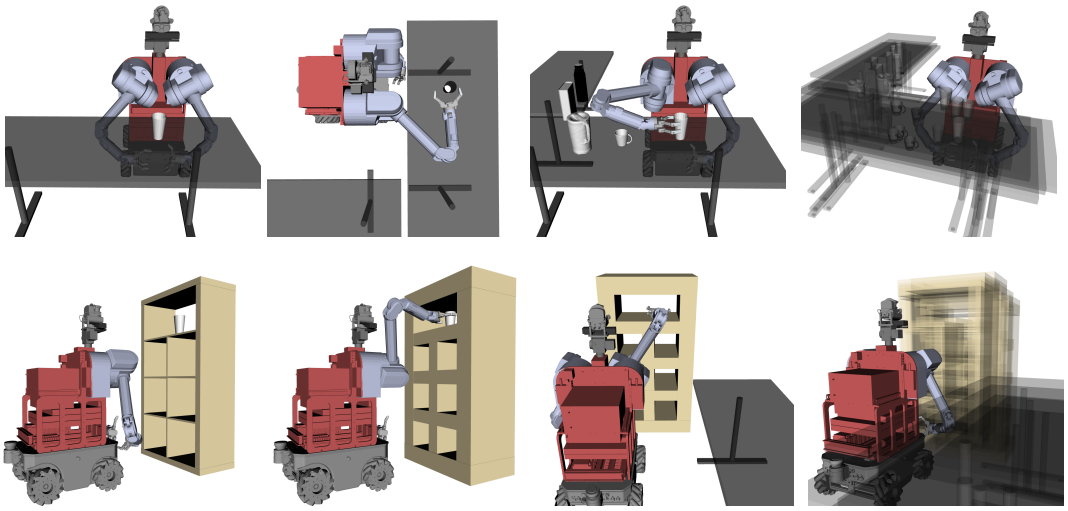


Figure 3.8: Top: Class 1 tabletop manipulation, (a) Easy, (b) Medium and (c) Hard tasks. (d) Samples from Class 1 dataset. Bottom: Class 2 shelf manipulation, (e) Easy, (f) Medium and (g) Hard tasks. (h) Samples from Class 2 dataset

roadmap contained a solution. Specifically, it was 2000 for $\mathbb{R}^2$, 8000 for $SE(2)$ and 30,000 for $\mathbb{R}^7$.

For our experiments in $\mathbb{R}^2$, we use 6 datasets, each corresponding to different parametric distribution of obstacles from which we sample 1000 worlds [25]. Priors are computed by collision checking a fixed graph on the training data and averaging edge outcomes to approximate the original parametric distribution. Figure 3.6 illustrates the prior over edges and diversity of worlds considered in $\mathbb{R}^2$. All the problems lie in a unit square with source and target locations chosen to be bottom-left at $(0.1,\ 0.1)$ and top-right at $(0.9,\ 0.9)$ respectively.

For our experiments in $SE(2)$, we consider the Piano Movers' problem, an OMPL benchmark problem [119]. Figure 3.7 illustrates the environment, and a path between the source and target positions chosen. We relied on a simple prior obtained by voxelizing the workspace environment and collision checking the graph against the voxel primitives. Since we are considering a fixed environment, we will report our results on different Halton graphs obtained by applying a random offset to the 3D Halton sequence.

For manipulation problems in $\mathbb{R}^7$, we consider two classes as shown in Figure 3.8: tabletop manipulation (top) and reaching into shelf (bottom). Within each class of manipulation, we consider three tasks increasing in degree of constraints: *Easy*(column 1), *Medium*(column 2) and *Hard*(column 3). The source and target configurations are fixed within each class. The source configuration is visualized in column 1, and the target configuration is illustrated in columns 2 and 3 for each class. Class 1 (Figure 3.8, top row) considers tabletop manipulation problems where the robot needs to reach for an object on the table. *Easy* considers a single table with a single object on the table. *Medium*

and *Hard* add further constraints and clutter in the environment as illustrated in the figure. Class 2 (Figure 3.8, bottom row) considers manipulation problems where the robot needs to reach for an object in a shelf. *Easy* considers a shelf with large open spaces. *Medium* and *Hard* add further constraints and clutter in the environment as illustrated in the figure. For our experiments, we sampled 500 random worlds where the objects in the environment are moved along the three axes of translation. Priors are computed by collision checking a fixed graph on the sampled worlds and averaging edge outcomes. The last column in Figure 3.8 illustrates some samples generated.

3.5.2 *Algorithm Details*

We implement 3 EVENTS and 3 SELECTORS described in Algorithms 2 and 3 to get a set of 9 algorithms under GLS for evaluation. As observed previously [94] and in Figures 3.9,3.10, the optimal depth depends on the planning problem and we choose the appropriate depth i.e. 3 for $\mathbb{R}^2$, $SE(2)$ problems and 5 for manipulation problems. For SUBPATHEXISTENCE, we choose δ from the pareto curve (Figure 3.11) of vertices rewired vs edges evaluated computed on the training data. The slope of the line is the ratio of their relative cost – the point of interesection corresponds to the value of δ that minimizes planning time, $1e-2$ for $\mathbb{R}^2$, $SE(2)$ and $1e-4$ for manipulation problems. For all planning problems, the algorithms use the Euclidean heuristic to search.

3.5.3 *Results and Analysis*

As noted in Section 3.5.2, we implemented 3 Events and 3 Selectors to evaluate a total of 9 algorithms under GLS. To analyze the trade-offs that EVENTS and SELECTORS induce between computational costs of evaluation and search, we test our hypotheses on the diverse set of $\mathbb{R}^2$ datasets detailed in Section 3.5.1. We then finalize on 3 algorithms: LAZYSP(SUBPATHEXISTENCE, FAILFAST), LRA*(CONSTANTDEPTH, FAILFAST) and GLS(SUBPATHEXISTENCE, FAILFAST) to further understand the effect of toggling between search and evaluation. We evaluate these algorithms on Piano MoversâĂŹ problem in $SE(2)$ and manipulation problems in $\mathbb{R}^7$.

3.5.4 *Simulated* $\mathbb{R}^2$ *worlds*

Table 3.2 shows the planning times of various algorithms under GLS. The planning times are the median quantities obtained from experiments over 100 different worlds sampled within the environment type. Fig. 3.12(c) shows the ranking of the planning times of the algorithms

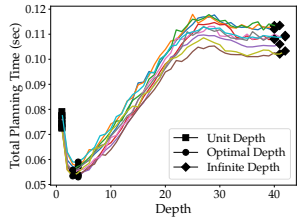


Figure 3.9: Planning time on varying α in CONSTANTDEPTH on $\mathbb{R}^2$ problems

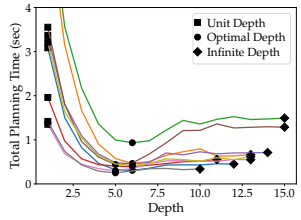


Figure 3.10: Planning time on varying α in CONSTANTDEPTH on $\mathbb{R}^7$ problems

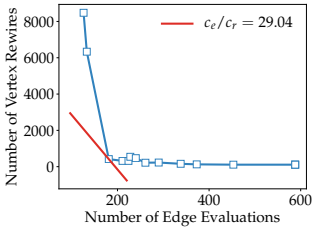


Figure 3.11: Pareto curve on varying δ in SUBPATHEXISTENCE on a $\mathbb{R}^2$ planning problem

over 400 test samples across $\mathbb{R}^2$ environment types(lower plannning time in a problem translates to a higher rank). We note that GLS with SubpathExistence and FailFast consistently outperforms remaining algorithms on a majority of environments.

	SP,F	SP,A	SP,FF	CD,F	CD,A	CD,FF	SE,F	SE,A	SE,FF
Square									
Time	0.331 0.32 ; 0.34	0.454 0.42 ; 0.48	0.372 0.34 ; 0.4	0.221 0.21 ; 0.23	0.259 0.25 ; 0.27	0.223 0.22 ; 0.23	0.171 0.16 ; 0.18	0.161 0.15 ; 0.18	**0.116** 0.11 ; 0.13
$\lvert\mathcal{E}_{\text{eval}}\rvert$	190	308	137	273	344.5	315.5	200.5	206	153
$\lvert\mathcal{V}_{\text{rwr}}\rvert$	7058.5	8502.5	9859	1076.5	641.5	69.5	1104.5	603.5	318.5
OneWall									
Time	0.25 0.23 ; 0.28	0.311 0.29 ; 0.34	0.223 0.21 ; 0.27	0.187 0.17 ; 0.21	0.212 0.2 ; 0.24	0.234 0.22 ; 0.27	0.144 0.13 ; 0.17	0.117 0.1 ; 0.15	**0.087** 0.08 ; 0.13
$\lvert\mathcal{E}_{\text{eval}}\rvert$	164	230	101	247	291	331	183	148	109
$\lvert\mathcal{V}_{\text{rwr}}\rvert$	4830	5367	5449	513	306	93	581	482	389
TwoWall									
Time	0.419 0.37 ; 0.44	0.394 0.38 ; 0.42	0.377 0.37 ; 0.47	0.262 0.22 ; 0.27	0.301 0.25 ; 0.31	0.29 0.26 ; 0.31	0.22 0.18 ; 0.23	0.169 0.15 ; 0.19	**0.162** 0.15 ; 0.2
$\lvert\mathcal{E}_{\text{eval}}\rvert$	224	242.5	144	310	393	407	287	224.5	202.5
$\lvert\mathcal{V}_{\text{rwr}}\rvert$	9360	7997	9870	1594.6	914.5	165.5	697	435	711.5
Mazes									
Time	1.334 1.26 ; 1.44	1.292 1.19 ; 1.45	1.272 1.21 ; 1.35	0.575 0.53 ; 0.61	0.777 0.72 ; 0.83	0.559 0.52 ; 0.6	0.615 0.57 ; 0.66	0.578 0.54 ; 0.61	**0.337** 0.31 ; 0.36
$\lvert\mathcal{E}_{\text{eval}}\rvert$	531	471	307.5	630	895	785.5	588	544.5	352.5
$\lvert\mathcal{V}_{\text{rwr}}\rvert$	34359	34379.5	37750	4769	5357.5	326	7266.5	7039.5	3213
BTrap									
Time	0.334 0.29 ; 0.37	0.356 0.31 ; 0.41	0.316 0.27 ; 0.37	0.329 0.3 ; 0.37	0.381 0.34 ; 0.43	0.522 0.46 ; 0.56	0.334 0.28 ; 0.37	0.365 0.32 ; 0.41	**0.292** 0.26 ; 0.36
$\lvert\mathcal{E}_{\text{eval}}\rvert$	271	297	228	447	529	743	271	313	230
$\lvert\mathcal{V}_{\text{rwr}}\rvert$	5164	5296	5590	588	368	72	5164	5214	4667
Forest									
Time	0.277 0.24 ; 0.31	0.267 0.24 ; 0.31	0.269 0.22 ; 0.33	0.231 0.21 ; 0.25	0.249 0.23 ; 0.28	0.318 0.3 ; 0.35	**0.219** 0.2 ; 0.23	0.234 0.2 ; 0.26	0.229 0.19 ; 0.28
$\lvert\mathcal{E}_{\text{eval}}\rvert$	174.5	184.5	165.5	306.5	342.5	450.5	190	220	174.5
$\lvert\mathcal{V}_{\text{rwr}}\rvert$	5524.5	4936	5467.5	579	346	95.5	3075	2855	3827.5

Table 3.2: Planning Time(millisec) and number of operations by algorithms under GLS. Median reported on 100 tests for all metrics, and 95% CI for the planning times.

We found strong evidence to support **H 1** - SubpathExistence exhibits lowest planning times in 99% of the problems (Fig. 3.12(a)). Corresponding median planning times supporting the hypothesis are reported in Table 3.2. To qualitatively understand the trend, we consider the behaviours stemming from the selection of an Event on a representative TwoWall environment in Fig. 3.16. The figure shows a comparison of ShortestPath, ConstantDepth and SubpathExistence (for the Forward selector). We can see that ShortestPath checks small number of edges but rewires significant portion of the search tree. The trend is reversed in ConstantDepth (depth 1). SubpathExistence is able to balance both by exploiting priors - it triggers events when the search reaches the walls thus reducing rewires.

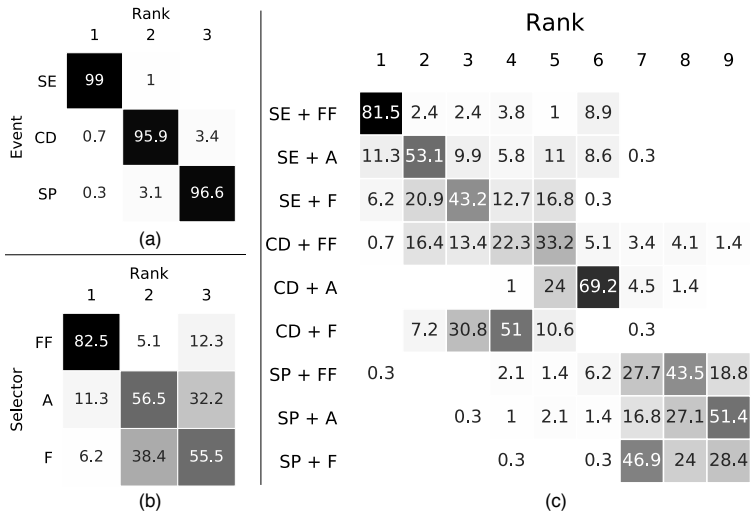


Figure 3.12: (a) Events, (b) Selectors and (c) Algorithms, ranked by planning times on $\mathbb{R}^2$ environments. Each cell indicates percentage of problems on which corresponding rank has been obtained.

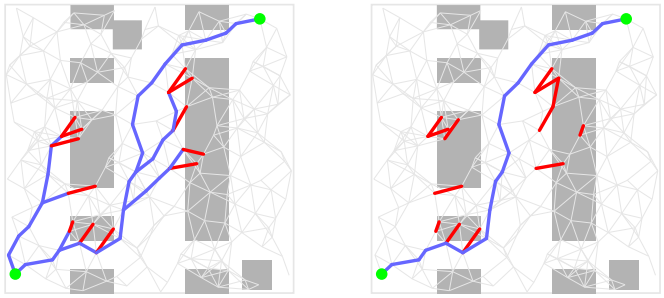


Figure 3.13: Valid (blue) and invalid (red) edges evaluated at termination of GLS with FORWARD(49 edges) and FAILFAST(32 edges).

We also found strong evidence to support **H 2** - FAILFAST exhibits the lowest planning times in 83% (Fig. 3.12(b)) of the problems across the datasets. In Table 3.2, we note that for a given event, FAILFAST has the lowest planning time in majority of the datasets. To understand this trend qualitatively, we consider the behaviors stemming from the selection of a SELECTOR on a representative TwoWall environment in Fig. 3.13. It shows a comparison of FORWARD and FAILFAST (for SHORTESTPATH event) - FAILFAST quickly eliminates paths by checking the edge most likely to be in collision (see Theorems 3.4.4, 3.4.5).

We found strong evidence to support **H 3**. Figures 3.14 show that as graphs get larger or density of obstacles (problem difficulty) increases, planning times of SHORTESTPATH grows at a faster rate than SUBPATHEXISTENCE.

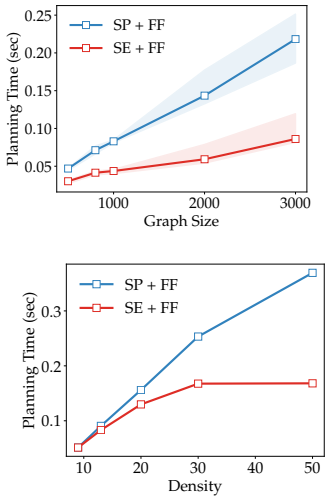
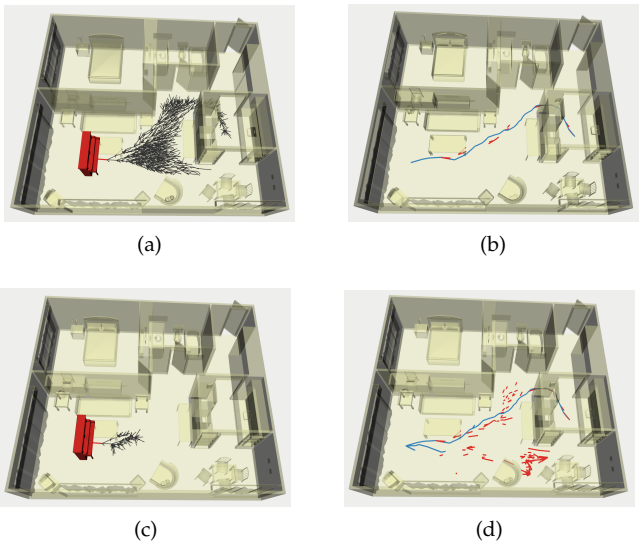


Figure 3.14: (top) Planning times as the size of the graph in TwoWall is increased. (bottom) Planning time as the density of obstacles in Forest is increased.

3.5.5 *Piano Movers' Problem*

We consider the Piano Movers' problem in $SE(2)$ from the Apartment scenario in OMPL [120] to test our hypotheses. In Table 3.3, we re-

port with 95% C.I., the mean planning times across 100 instances of the problem. We see that GLS(SUBPATHEXISTENCE, FAILFAST) outperforms the other algorithms in planning time supporting **H 1**. Additionally, we note from Table 3.3 that GLS with SUBPATHEXISTENCE neither has the lowest edge evaluation time, nor lowest vertex rewire time. However, by balancing the two computational costs, exhibits the lowest total planning time. To understand the consequence of selecting an EVENT, we consider their behaviors in Figure 3.15. It illustrates the savings of SUBPATHEXISTENCE in vertex rewires over SHORTESTPATH. LAZYSP with SHORTESTPATH has to rewire a large search tree every time a path is found to be in collision. GLS with SUBPATHEXISTENCE, on the other hand, halts the search as soon as it enters a region of low probability, eliminates the paths and hence drastically minimizes rewiring time at the cost of few additional edge evaluations over LAZYSP.



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

Figure 3.15: Search and evaluation by SHORTESTPATH(a, b) and SUBPATHEXISTENCE(c, d). (a) and (c): subtree of vertices rewired in the first iteration (12,495 and 1235 resp. at termination). (b) and (d): edges evaluated at termination (63 and 171 resp.).

3.5.6 Manipulation Problems

We consider two classes of manipulation problems each with varying degree of clutter in the environment to test **H 1** for the selection of EVENT, and **H 3** for varying problem difficulty. In Table 3.3, we report the mean planning times with 95% C.I. across 50 problems each. We note that in Easy and Medium tasks, LAZYSP (SHORTESTPATH) and LRA* (CONSTANTDEPTH) dominate while in Hard tasks GLS with SUBPATHEXISTENCE outperforms. Although this observation weakens **H 1** for simpler tasks, it strengthens both **H 1** and **H 3** for cluttered environments. In Easy and Medium tasks, since the free regions in the configuration space are dominant, algorithms terminate with

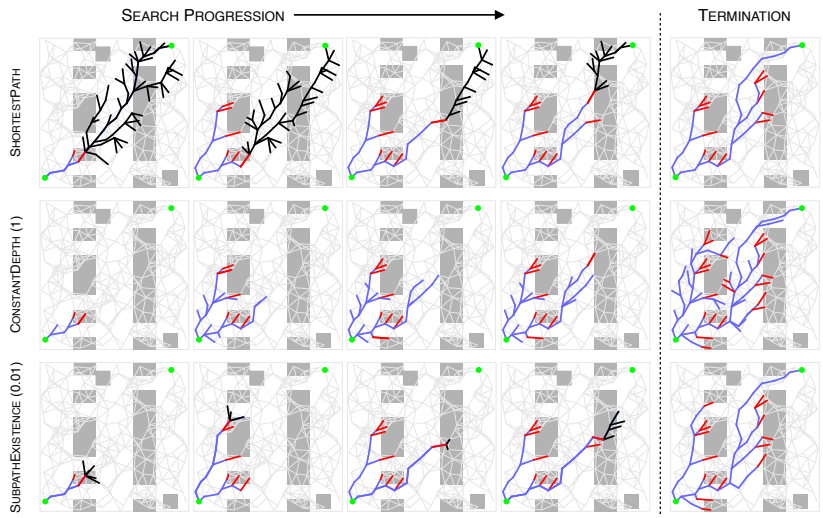


Figure 3.16: Snapshots of search and evaluation by GLS with Forward selector and different events. Edges evaluated to be valid (blue), invalid (red) and subtree of vertices to be rewired (black) are shown. From top to bottom at termination: the number of edge evaluations are (49, 97, 62) and the number of vertex rewires are (361, 21, 69).

fewer vertex rewires. Delayed Events like ShortestPath which perform optimally with edge evaluations therefore exhibit better performance. However, in cluttered environments vertex rewiring is a competing computation cost and therefore SubpathExistence which balances the two computational costs outperforms. The behaviors of GLS (SubpathExistence) and LazySP (ShortestPath) are illustrated in Fig. 3.17. LazySP rewires large subtrees every time a path is found to be in collision whereas GLS toggles search to evaluation as it enters a region of low probability. It eliminates the paths in collision to reduce wasted search effort and hence reduces rewiring time (at the cost of few additional edge evaluations) over LazySP.

3.6 *Discussion*

We presented a general framework for lazy search (GLS). The framework interleaves two phases, search and evaluation. In the search phase, it extends a lazy shortest-path tree forward without evaluating any edges until an Event is triggered. It then switches to evaluation phase. It finds the shortest subpath to a leaf node of the tree and invokes a Selector to evaluate an edge on it. Careful choice of Event and Selector allows the balance of search effort with edge evaluation to minimize overall planning time.

The framework, quite expressive, lets us capture a range of lazy search algorithms (Table 3.1). While we draw inspiration from prior work interleaving search and execution, such as IDA* and RTA* to derive LRA*, our definition of the Event makes the GLS *adaptive*. This lets us derive new algorithms that are edge optimal while saving

	GLS(SE + FF)	LRA*(CD + FF)	LazySP(SP + FF)
Piano Movers'			
Total Planning Time	**1.004** 0.89 ; 1.09	1.139 1.07 ; 1.23	1.264 0.97 ; 1.53
Edge Evaluation Time	0.172 0.14 ; 0.2	0.497 0.46 ; 0.54	0.107 0.08 ; 0.14
Vertex Rewire Time	0.832 0.75 ; 0.9	0.641 0.61 ; 0.69	1.157 0.89 ; 1.4
HERB: Tabletop Manipulation			
Task 1: Easy			
Total Planning Time	0.098 0.08 ; 0.36	**0.073** 0.05 ; 0.37	0.075 0.05 ; 0.31
Edge Evaluation Time	0.092 0.08 ; 0.32	0.066 0.04 ; 0.28	0.063 0.04 ; 0.22
Vertex Rewire Time	0.006 0.01 ; 0.04	0.007 0.01 ; 0.09	0.012 0.01 ; 0.09
Task 1: Medium			
Total Planning Time	**0.573** 0.47 ; 0.72	0.768 0.6 ; 0.85	0.633 0.47 ; 0.81
Edge Evaluation Time	0.489 0.42 ; 0.6	0.555 0.42 ; 0.63	0.337 0.27 ; 0.4
Vertex Rewire Time	0.084 0.06 ; 0.12	0.213 0.18 ; 0.23	0.296 0.2 ; 0.42
Task 1: Hard			
Total Planning Time	**0.834** 0.72 ; 0.96	0.921 0.77 ; 1.09	0.940 0.51 ; 1.34
Edge Evaluation Time	0.719 0.63 ; 0.82	0.804 0.68 ; 0.94	0.414 0.29 ; 0.56
Vertex Rewire Time	0.115 0.09 ; 0.14	0.117 0.09 ; 0.15	0.526 0.22 ; 0.77
HERB: Reach into Shelf			
Task 1: Easy			
Total Planning Time	0.340 0.15 ; 0.38	**0.281** 0.12 ; 0.3	0.288 0.12 ; 0.32
Edge Evaluation Time	0.323 0.14 ; 0.35	0.252 0.11 ; 0.27	0.250 0.11 ; 0.28
Vertex Rewire Time	0.016 0.01 ; 0.02	0.029 0.01 ; 0.04	0.038 0.01 ; 0.04
Task 1: Medium			
Total Planning Time	1.349 1.26 ; 1.82	1.280 1.18 ; 1.74	**1.152** 1.01 ; 1.78
Edge Evaluation Time	1.267 1.19 ; 1.7	0.960 0.86 ; 1.27	0.774 0.67 ; 0.99
Vertex Rewire Time	0.082 0.07 ; 0.13	0.320 0.31 ; 0.47	0.377 0.34 ; 0.79
Task 1: Hard			
Total Planning Time	**2.260** 1.25 ; 2.45	3.120 1.43 ; 3.51	3.555 1.48 ; 4.28
Edge Evaluation Time	2.143 1.19 ; 2.32	2.391 1.1 ; 2.73	1.582 0.81 ; 1.85
Vertex Rewire Time	0.117 0.07 ; 0.13	0.729 0.32 ; 0.78	1.973 0.67 ; 2.43

Table 3.3: Mean planning time (seconds) of GLS with SubpathExistence, ShortestPath and ConstantDepth as events, and FailFast as selector on $SE(2), \mathbb{R}^7$ problems. 95% CI reported.

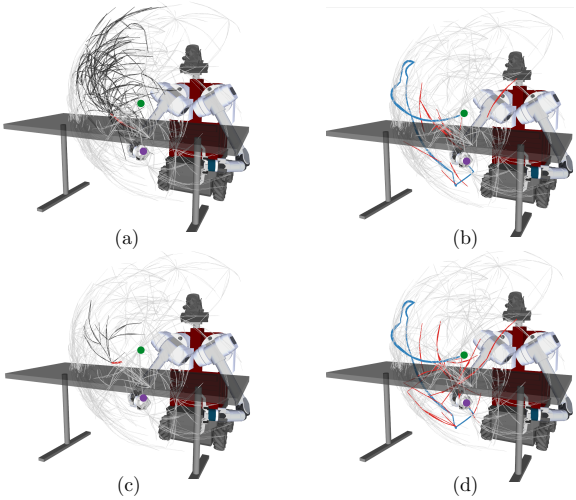


Figure 3.17: Search and evaluation by ShortestPath (top) and SubpathExistence (bottom). (a) and (c): subtree of vertices rewired in the first iteration (21,178 and 11,342 resp. at termination). (b) and (d): edges evaluated (136 and 243 resp.).

on search effort (Theorem 3.3.7).

While our assumes independence between edges, it leaves for further examination more sophisticated Selector policies [26] that exploit correlations amongst edges to minimize evaluation cost. One way to do so is to train policies to imitate clairvoyant oracles in a similar fashion as [22, 10].

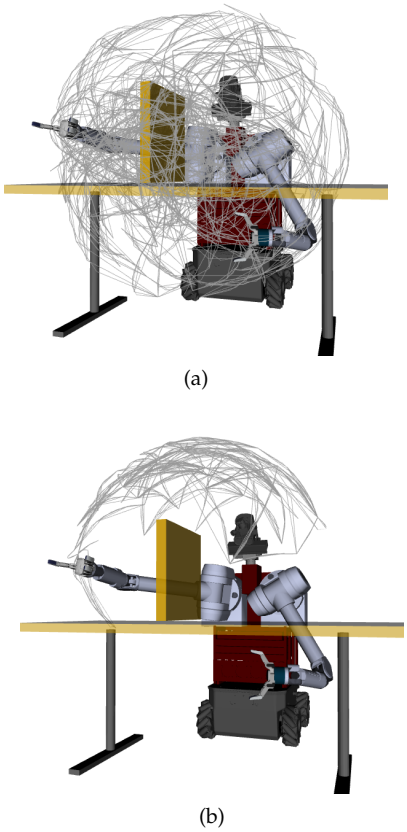
While GLS focuses on graph search, the notion of interleaving lazy search with edge evaluations can be extended to an anytime paradigm; this would let us use heuristics that exploit edge priors to guide the search through regions of high probability [98], for significant speedups. Finally, we believe GLS can interleave search efficiently over multiple resolutions of approximation in problems where multiple lazy estimates of weight functions are available, e.g., in kinodynamic planning, where different relaxations of the boundary value problem can be obtained.

4
Leveraging Experience for Graph Generation

Shortest path algorithms are agnostic to the graph they are operating on. While GLS aims to be efficient on *any* graph, since the computational cost (Chapter 3, Equation 3.2) depends on the size of the graph (number of vertices and edges), and the quality of the solution on the graph's underlying sampling distribution, constructing favorable graphs becomes central to the problem of efficient motion planning.

In this chapter we consider the challenge of sampling desirable graphs that are sparse, allowing for fast search, with vertices spread out at key locations such that a low-cost feasible path exists. We address this problem by leveraging prior experience. State-of-the-art is to train a conditional variational auto-encoder (CVAE) [62] on the prior dataset with the shortest paths as target input. While this is quite effective on many problems, we show it can fail in the face of complex obstacle configurations or mismatch between training and testing.

We present an algorithm LEGO that addresses these issues by training the CVAE with target samples that satisfy two important criteria.
Firstly, these samples belong only to *bottleneck* regions along near-optimal paths that are otherwise difficult-to-sample with a uniform sampler. Secondly, these samples are spread out across *diverse regions* to maximize the likelihood of a feasible path existing. We formally define these properties and prove performance guarantees for LEGO .



(a)

(b)

Figure 4.1: Comparison of graphs generated from (a) uniform Halton sequence sampler and (b) from a generative model trained using LEGO . The task is to plan from the shown configuration over the table and obstacle to the other side. The graph is visualized by end effector traces of the edges.

4.1 Introduction

We examine the problem of leveraging prior experience in sampling-based motion planning. Recall that in this framework, the continuous configuration space of a robot is sampled to construct a graph [76, 91] where vertices represent robot configurations and edges represent potential movements of the robot. A shortest path algorithm [53] is then run to compute a path between any two vertices on the graph. The challenge is to place a *small set of samples in key locations* such that

the algorithm can find a high quality path with small computational effort as shown in Fig. 4.1.

Low dispersion samplers such as Halton sequences [50] are quite effective in uniformly covering the space and thus bounding the solution quality [65] (Figure 4.1(a)). However, as they decrease dispersion *uniformly* in C-space, a narrow passage with δ clearance in a d-dimensional space requires $O((\frac{1}{\delta})^d)$ samples to find a path [64]. This motivates the need for biased sampling to *selectively densify* in regions with narrow passages [55, 60, 16, 58, 88]. These techniques are applicable across a wide range of domains and perform quite well in practice.

However, not all narrow passages are relevant to a given query. Biased sampling techniques, which do not have access to the likelihood of the optimal path passing through a region, can sample in more regions than necessary. Interestingly, the different environments that a robot operates in share a lot of structural similarity. We can use information extracted from planning on one such environment to decide how to sample on another; we can *learn* sampling distributions using tools such as conditional variational auto-encoder (CVAE). The best a generative model can do is to sample only along the true shortest path. Ichter et al. [62] propose a useful approximation to train a learner to sample along the *predicted* shortest path: given a training dataset of worlds, compute shortest paths, and train a model to independently predict nodes belonging to the path. However, this puts *all of the burden* on the learner. Any prediction error, due to approximation or train-test mismatch, results in failure to find a feasible path.

We argue that a sampler, instead of trying to predict the shortest path, needs to only identify key regions to focus sampling at, and let the search algorithm determine the shortest path. Essentially, we ask the following question:

> *How can we share the responsibility of finding the shortest path between the sampler and search?*

Our key insight is for the sampler to predict not the shortest path, but samples that possess two characteristics: (a) samples in bottleneck regions; these are regions containing near-optimal paths, but are difficult for a uniform sampler to reach, and (b) samples that exhibit diversity; train-test mismatch is common and to be robust to it we need to sample nodes belonging to a diverse set of alternate paths. The search algorithm can then operate on a sparse graph containing useful but diverse samples to compute the shortest path.

We present an algorithmic framework, **L**everaging **E**xperience with **G**raph **O**racles (LEGO) summarized in Figure 4.2, for training a CVAE on a prior database of worlds to learn a generative model that can be used for graph construction. During training (Figure 4.2a), LEGO

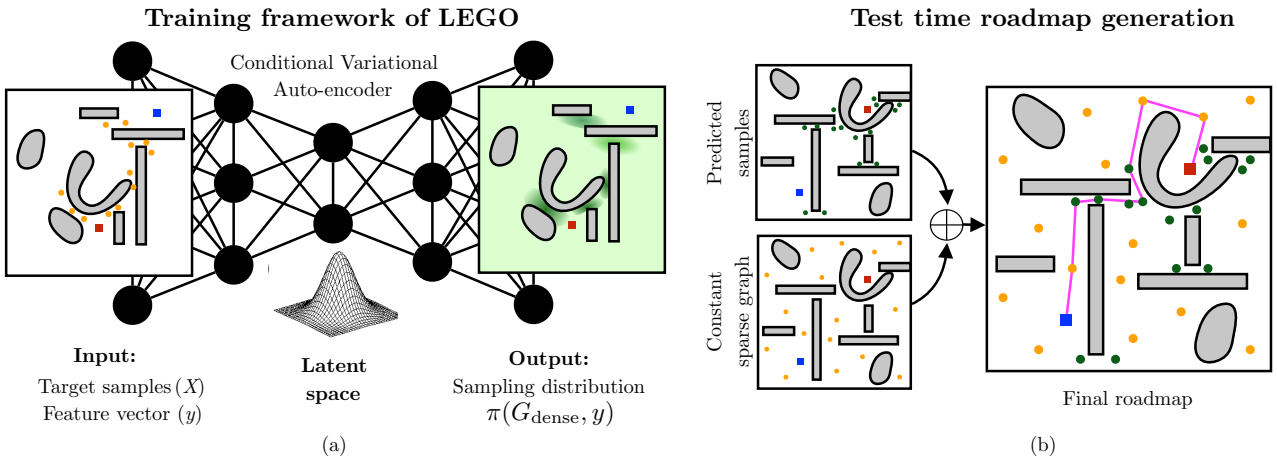


Figure 4.2: The LEGO framework for training a CVAE to predict a graph. (a) The training process for learning a generative sampling distribution using a CVAE. The input is a pair of candidate samples and feature vector. (b) At test time, the model is sampled to get vertices which are then composed with a constant sparse graph to get a final graph.

processes a uniform dense graph to identify a sparse subset of vertices. These vertices are a *diverse* set of *bottleneck* nodes through which a near-optimal path must pass. These are then fed into a CVAE [79] to learn a generative model. At test time (Figure 4.2b), the model is sampled to get a set of vertices which is additionally composed with a sparse uniform graph to get a final graph. This graph is then used by the search algorithm to find the shortest path.

4.2 *Problem Formulation*

Given a database of prior worlds, the overall goal is to learn a policy that predicts a graph which in turn is used by a search algorithm to efficiently compute a high quality feasible path. To make this chapter self contained, we repeat some of the notation mentioned previously in Chapters 2,3. Let $\mathcal{X} \in \mathbb{R}^n$ denote an n-dimensional configuration space. Let $\mathcal{X}_{\text{obs}}$ be the portion in collision and $\mathcal{X}_{\text{free}} = \mathcal{X} \backslash \mathcal{X}_{\text{obs}}$ denote the free space. A path $\xi : [0,1] \to \mathcal{X}$ is said to be collision-free if $\xi(t) \in \mathcal{X}_{\text{free}}$ for all $t \in [0,1]$. The cost of a path is defined by the cost functional $c : \xi \to \mathbb{R}_{\geq 0}$. Additionally, we define a *motion planning problem* $\Lambda = \{x_s, x_t, \mathcal{X}_{\text{free}}\}$ as a tuple of start state $x_s \in \mathcal{X}_{\text{free}}$, goal state $x_t \in \mathcal{X}_{\text{free}}$ and free space $\mathcal{X}_{\text{free}}$. Given a problem, a path ξ is said to be *feasible* if it is collision-free, $\xi(0) = x_s$ and $\xi(1) = x_t$. Let Ξ denote the set of all feasible paths. We wish to solve the *optimal* motion planning problem by finding a feasible path ξ^* that minimizes the cost functional $c(.)$.

We now embed the problem on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that each vertex $v \in \mathcal{V}$ corresponds to a state $x \in \mathcal{X}$. Let the connection strategy to denote if two vertices should have an edge[1] be defined by an indicator function $\texttt{Link} : \mathcal{X} \times \mathcal{X} \to \{0,1\}$. The weight of an edge $w(u,v)$ is

[1] Note this does not involve collision checking. We consider undirected graphs for simplicity. However, it easily extends to directed graphs.

the cost of traversing the edge, and $w(\xi)$ the cost of traversing a path where ξ is now a sequence of vertices on the graph.

Let $|\mathcal{G}|$ denote the cardinality of the graph, i.e. the size of $|\mathcal{V}|$. We introduce a graph operation with the notation $\mathcal{G} \xleftarrow{+} X$ to compactly denote insertion of a new set of vertices X, i.e. $\mathcal{V} \leftarrow \mathcal{V} \cup X$, and edges, $\mathcal{E} \leftarrow \mathcal{E} \cup \{(u, v) \mid u \in X,\ v \in \mathcal{G},\ \texttt{Link}\,(u, v) = 1\}$.

A graph search algorithm ALG is given a graph $\mathcal{G}$ and a planning problem Λ. First, it adds the start-goal pair to the graph, i.e $\mathcal{G}' = \mathcal{G} \xleftarrow{+} \{x_s, x_t\}$. It then collision checks edges against $\mathcal{X}_{\text{free}}$ till it finds and returns the shortest feasible path ξ^*. The cost of such a path can hence be found by evaluating $w(\text{ALG}(\mathcal{G}, \Lambda))$. If ALG is unable to find any feasible path, it returns $\emptyset$ which corresponds to $w_{\max}$.

Definition 4.2.1 (Dense Graph)**.** *We assume we have a* dense graph $\mathcal{G}_{\text{dense}} = (\mathcal{V}_{\text{dense}}, \mathcal{E}_{\text{dense}})$ *that is sufficiently dense s.t. for any plausible planning problem, it contains a sufficiently low-cost feasible path.*

Henceforth, we care about competing with $\mathcal{G}_{\text{dense}}$. We reiterate that searching this graph, $\text{ALG}(\mathcal{G}_{\text{dense}}, \Lambda)$, is too computationally expensive to perform online.

We wish to learn a mapping from features extracted from the problem to a sparse subgraph of $\mathcal{G}_{\text{dense}}$. Let $y \in \mathbb{R}^m$ be a feature representation of the planning problem. Let $\pi(\mathcal{G}_{\text{dense}}, y)$ be a *subgraph predictor oracle* that maps the feature vector to a subgraph $\mathcal{G} \subset \mathcal{G}_{\text{dense}}$, $|\mathcal{G}| \leq |\mathcal{G}_{\text{dense}}|$. We wish to solve the following optimization problem:

Problem 1 (Optimal Subgraph Prediction)**.** *Given a joint distribution $P(\Lambda, y)$ of problems and features, and a dense graph $\mathcal{G}_{\text{dense}}$, compute a subgraph predictor oracle π that minimizes the ratio of the costs of the shortest feasible paths in the subgraph and the dense graph:*

$$\pi^* = \underset{\pi \in \Pi}{\arg\min}\, \mathbb{E}_{(\Lambda, y) \sim P(\Lambda, y)} \left[\frac{w(\text{ALG}(\pi(\mathcal{G}_{\text{dense}}, y), \Lambda))}{w(\text{ALG}(\mathcal{G}_{\text{dense}}, \Lambda))} \right] \tag{4.1}$$

4.3 Framework for Predicting Graphs

We now present a framework for training graph predicting oracles as illustrated in Figure 4.2(a). This is a generalization of the approach presented in [62]. The framework applies three main approximations. First, instead of predicting a subgraph $\mathcal{G} \subset \mathcal{G}_{\text{dense}}$, we learn a mapping that directly predicts states $x \in \mathcal{X}$ in the continuous space[2]. Secondly, instead of solving a structured prediction problem, we learn an i.i.d sampler that will be invoked repeatedly to get a set of vertices. These vertices are then connected according to an underlying connection rule, such as k-NN, to create a graph. Thirdly, we compose the sampled graph with a *constant sparse graph* $\mathcal{G}_{\text{sparse}} \subset \mathcal{G}_{\text{dense}}$, $|\mathcal{G}_{\text{sparse}}| \leq N$.

[2] For cases where a subgraph is preferred, e.g. $\mathcal{G}_{\text{dense}}$ lies on a constraint manifold, one can design a projection operator $\mathcal{P} : \mathcal{X} \to \mathcal{V}_{\text{dense}}$

This ensures that the final predicted graph has some minimal coverage[3].

[3] Since $\mathcal{G}_{\text{dense}}$ is a Halton graph, we use the first N Halton sequences.

The core component of the framework is a Conditional Variational Auto-encoder (CVAE) [110] which is used for approximating the desired sample distribution. CVAE is an extension of a traditional variational auto-encoder [79] which is a directed graphical model with low-dimensional Gaussian latent variables. CVAE is a *conditional* graphical model which makes it relevant for our application where conditioning variables are features of the planning problem. We provide a high level description for brevity, and refer the reader to [35] for a comprehensive tutorial.

Here $x \in \mathcal{X}$ is the output random variable, $z \in \mathbb{R}^L$ is the latent random variable and $y \in \mathbb{R}^m$ is the conditioning variable. We wish to learn two *deterministic mappings* - an *encoder* and a *decoder*. An encoder maps (x, y) to a mean and variance of a Gaussian $q_\phi(z|x, y)$ in latent space, such that it is "close" to an isotropic Gaussian $\mathcal{N}(0, I)$. The decoder maps this Gaussian and y to a distribution in the output space $p_\theta(x|z, y)$. This is achieved by maximizing the following objective:

$$\mathcal{L}(x, y; \theta, \phi) = -D_{KL}\left(q_\phi(z|x, y) \,||\, \mathcal{N}(0, I)\right) + \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(x|y, z^{(l)}) \tag{4.2}$$

Note that the encoder is needed only for training. At test time, only the decoder is used to map samples from an isotropic Guassian in the latent space to samples in the output space.

We train the CVAE by passing in a dataset $\mathcal{D} = \{X_i, y_i\}_{i=1}^{D}$. y_i is the feature vector (conditioning variable) extracted from the planning problem Λ_i. X_i is the desired set of nodes extracted from the dense graph $\mathcal{G}_{\text{dense}}$ that we want our learner to predict. Hence we train the model by maximizing the following objective.

$$\mathcal{R}(\mathcal{D}; \theta, \phi) = \frac{1}{|\mathcal{D}|}\sum_{i=1}^{|\mathcal{D}|}\sum_{j=1}^{|X_i|} \mathcal{L}\left(x_j, y_i; \theta, \phi\right) \tag{4.3}$$

4.3.1 General Train and Test Procedure

To summarize, the overall training framework is as follows:

1. Load a database of planning problems Λ_i and corresponding feature vectors y_i.
2. For each Λ_i, extract relevant nodes from the dense graph $\mathcal{G}_{\text{dense}}$ by invoking $X_i = \textsc{ExtractNodes}(\Lambda_i, \mathcal{G}_{\text{dense}})$.
3. Feed dataset $\mathcal{D} = \{X_i, y_i\}_{i=1}^{D}$ as input to CVAE.
4. Train CVAE and return learned decoder $p_\theta(x|y, z)$.

At test time, given a planning problem Λ, the graph predicting oracle $\pi(\mathcal{G}_{\text{dense}}, y)$ performs the following set of steps:

1. Extract feature vector y from planning problem Λ.
2. Sample N nodes using decoder $p_\theta(x|y,z)$.
3. Connect nodes to create a graph $\mathcal{G}$. Compose sampled graph with a constant sparse graph $\mathcal{G} \leftarrow \mathcal{G} \oplus \mathcal{G}_{\text{sparse}}$.

The focus of this work is on examining variants of the the node extraction function $X = \textsc{ExtractNodes}(\Lambda, \mathcal{G}_{\text{dense}})$. While the parameters of the CVAE are certainly relevant (discussed in Appendix B), in this paper we ask the question:

What is a good input X to provide to the CVAE?

To that end, we explore the following schemes:

1. ShortestPath : Extract nodes X_{sp} belonging to the shortest path. This is the baseline approach.
2. BottleneckNode : Extract nodes X_{bn} that correspond to bottlenecks along the shortest path (Section 4.4.1).
3. DiversePathSet : Extract nodes X_{div} belonging to multiple diverse shortest paths (Section 4.4.2).
4. LEGO : Extract nodes X_{lego} that correspond to bottlenecks along multiple diverse diverse shortest paths. This is our proposed approach (Section 4.4.3).

4.3.2 Shortcomings in learning the shortest path

We examine the scheme applied in [62] of using nodes belonging to the shortest path on the dense graph as input for training the CVAE. The rationality for this scheme is that the distribution of states belonging to the shortest path might lie on a manifold that can be captured by the latent space of the CVAE. This hypothesis is validated across many high-dimensional planning domains.

We argue that the presented results should not be entirely surprising. The intrinsic difficulty of a planning problem stems from having to search in multiple potential homotopy classes to find a feasible high quality solution. This often manifests in problems involving mazes, bugtraps or narrow passages where the search has to explore and backtrack frequently. Simply increasing the dimension of the problem does not necessarily render it difficult. On the contrary, since the volume of free space increases substantially, there is often an abundance of feasible paths. The challenge, of course, is to find a manifold on which

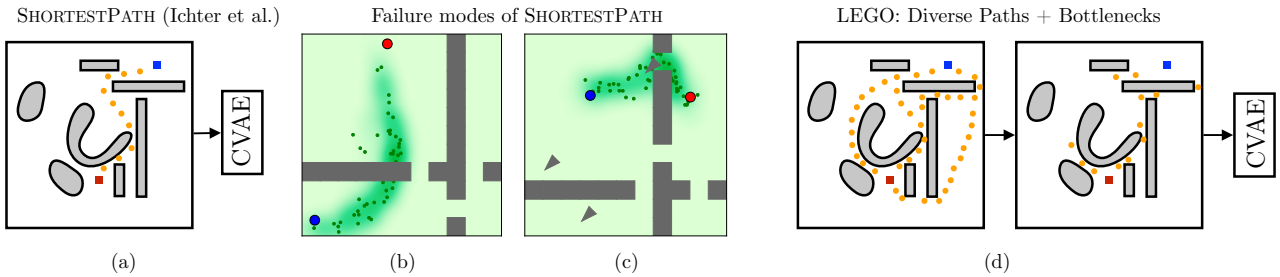


Figure 4.3: (a) The training input generated by SHORTESTPATH . Failure of SHORTESTPATH (a) to route through gaps and (b) due to unexpected obstacles. (d) The training input generated by LEGO . Diverse shortest paths are generated followed by extraction of bottleneck nodes.

such paths lie with high probability. This is where we found the CVAE to be critical - it learns to *interpolate* between the start and goal along a low dimensional manifold.

However, we are interested in more *difficult* problems where such interpolations would break down. Based on extensive evaluations of this SHORTESTPATH scheme, we were able to identify two concrete vulnerabilities:

1. *Failure to route through gaps*: Figure 4.3(b) shows the output of the CVAE when there is a gap through which the search has to route to get to the goal. The model gets stuck in a poor local minimum between linearly interpolating start-goal and routing through the gap since the network is not expressive enough to map the feature vector to such a path. This is tantamount to burdening the sampler to solve the planning problem.

2. *Presence of unexpected obstacles in test data*: Figure 4.3(c) shows the output of the CVAE when there are small, unexpected obstacles in the test data which were not present in the training data. The learned distribution samples over this obstacle as it only predicts what it thinks is the shortest path. Even if we were to have such examples in the training data, unless the feature extractor detects such obstacles, the problem remains.

4.4 *Leveraging Experience with Graph Oracles*

In this section, we present LEGO (Leveraging Experience with Graph Oracles), an algorithm to train a CVAE to predict sparse yet high quality roadmaps. We do so by tackling head-on the challenges identified in in Section 4.3. Firstly, we recognize that the learner does not have to directly predict the shortest path. Instead, we train it to predict only *bottleneck nodes* that can assist the underlying search in finding a near-optimal solution. Secondly, the roadmap must be robust to prediction errors of the learner. We safeguard against this by training the learner

Algorithm 4: BOTTLENECKNODE

Input : Planning problem Λ, Bottleneck tolerance ϵ
Dense path ξ^*_{dense}, Sparse graph $\mathcal{G}_{\text{sparse}}$
Output : Bottleneck nodes $\mathcal{V}_{\text{bn}}$

1 $\mathcal{G}_{\text{inf}} \leftarrow \mathcal{G}_{\text{sparse}} \oplus \xi^*_{\text{dense}}$ ▷ Add to sparse graph
2 $\eta \leftarrow 1$
3 **while** $w(\text{ALG}(\mathcal{G}_{\text{inf}}, \Lambda)) \leq (1+\epsilon) w(\xi^*_{\text{dense}})$ **do**
4 $\quad \eta \leftarrow \eta + \delta\eta$ ▷ Increase inflation
5 $\quad$ **for** $(u, v) \in \mathcal{E}_{\text{inf}} \setminus \mathcal{E}_{\text{sparse}}$ **do**
6 $\quad\quad w(u, v) \leftarrow \eta w(u, v)$ ▷ Inflate added edges
7 $\xi^*_{\text{inf}} \leftarrow \text{ALG}(\mathcal{G}_{\text{inf}}, \Lambda)$ ▷ Shortest inflated path
8 $\mathcal{V}_{\text{bn}} \leftarrow \xi^*_{\text{dense}} \cap \xi^*_{\text{inf}}$ ▷ Bottleneck nodes
9 **return** $\mathcal{V}_{\text{bn}}$

to predict a *diverse set of paths* with the hope that at-least one of them is feasible.

4.4.1 Bottleneck Nodes

We begin by noting that $\mathcal{G}_{\text{sparse}}$ has a uniform coverage over the entire configuration space. Hence, the learner only has to contribute a critical set of nodes that allow $\mathcal{G}_{\text{sparse}}$ to represent paths that are near-optimal with respect to the path in $\mathcal{G}_{\text{dense}}$. We call these *bottleneck nodes* as they correspond to regions that are difficult for a uniform sampler to cover. We define $X_{\text{bn}} = \text{BOTTLENECKNODE}(\Lambda, \mathcal{G}_{\text{dense}})$ as:

Definition 4.4.1 (Bottleneck Nodes)**.** *Given a dense graph $\mathcal{G}_{\text{dense}}$, find the smallest set of nodes which in conjunction with a sparse subgraph $\mathcal{G}_{\text{sparse}}$ contains a near-optimal path, i.e.*

$$\begin{aligned} \underset{\mathcal{V} \subset \mathcal{V}_{\text{dense}}}{\arg\min} \quad & |\mathcal{V}| \\ s.t. \quad & \frac{w(\text{ALG}(\mathcal{G}_{\text{sparse}} \oplus \mathcal{V}, \Lambda))}{w(\text{ALG}(\mathcal{G}_{\text{dense}}, \Lambda))} \leq 1 + \epsilon \end{aligned} \tag{4.4}$$

Here $\mathcal{G} \oplus \mathcal{V}'$ represents a merge operation, i.e. $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}'$, $\mathcal{E} \leftarrow \mathcal{E} \cup \{(u, v) \mid u \in \mathcal{V}', (u, v) \in \mathcal{E}_{\text{dense}}\}$.

The optimization Section 4.4 is combinatorially hard. We present an approximate solution in Algorithm 4. We use the optimal path ξ^*_{dense} on the dense graph and create an *inflated graph* $\mathcal{G}_{\text{inf}}(\eta)$ by composing $\mathcal{G}_{\text{sparse}} \oplus \xi^*_{\text{dense}}$ and inflating weights of newly added edges by η (Line 6). The idea is to disincentivize the search from using any of the newly added edges. This inflation factor is increased till a near-optimal path can no longer be found (Lines 3-6). At this point, the additional vertices that the shortest path on this inflated path pass through are es-

sential to achieve near-optimality. This is formalized by the following guarantee:

Proposition 1 (Bounded bottleneck edge weights)**.** *Let* $\mathcal{E}_{\text{bn}} \leftarrow \xi^*_{\text{inf}} \setminus \mathcal{E}_{\text{sparse}}$ *be the chosen bottleneck edges,* $\mathcal{E}^*_{\text{bn}}$ *be the optimal bottleneck edges and* ξ^*_{dense} *be the optimal path on* $\mathcal{G}_{\text{dense}}$*.*

$$\sum_{e_i \in \mathcal{E}_{\text{bn}}} w(e_i) \leq \sum_{e_i \in \mathcal{E}^*_{\text{bn}}} w(e_i) + \frac{(1+\epsilon) w(\xi^*_{\text{dense}})}{\eta} \tag{4.5}$$

Proof. (Sketch) Let $\mathcal{V}^*_{\text{bn}}$ be the optimal bottleneck nodes and $\mathcal{E}^*_{\text{bn}}$ be the optimal bottleneck edges. Let ξ^*_{bn} be the path returned by $\textsc{Alg}(\mathcal{G}_{\text{sparse}} \oplus \mathcal{V}^*_{\text{bn}}, \Lambda)$. From Definition 4.4.1, the following holds:

$$\sum_{e_i \in \mathcal{E}^*_{\text{bn}}} w(e_i) + \sum_{e_i \in \xi^*_{\text{bn}} \setminus \mathcal{E}^*_{\text{bn}}} w(e_i) \leq (1+\epsilon) w(\xi^*_{\text{dense}})$$
$$\sum_{e_i \in \xi^*_{\text{bn}} \setminus \mathcal{E}^*_{\text{bn}}} w(e_i) \leq (1+\epsilon) w(\xi^*_{\text{dense}})$$

Since ξ^*_{inf} is the shortest path on the inflated graph, we have:

$$\sum_{e_i \in \mathcal{E}_{\text{bn}}} w(e_i) + \eta \sum_{e_i \in \xi^*_{\text{inf}} \setminus \mathcal{E}_{\text{bn}}} w(e_i) \leq \eta \sum_{e_i \in \mathcal{E}^*_{\text{bn}}} w(e_i) + \sum_{e_i \in \xi^*_{\text{bn}} \setminus \mathcal{E}^*_{\text{bn}}} w(e_i)$$

Putting the two inequalities together we have:

$$\eta \sum_{e_i \in \xi^*_{\text{inf}} \setminus \mathcal{E}_{\text{bn}}} w(e_i) \leq \eta \sum_{e_i \in \mathcal{E}^*_{\text{bn}}} w(e_i) + \sum_{e_i \in \xi^*_{\text{bn}} \setminus \mathcal{E}^*_{\text{bn}}} w(e_i)$$
$$\sum_{e_i \in \xi^*_{\text{inf}} \setminus \mathcal{E}_{\text{bn}}} w(e_i) \leq \sum_{e_i \in \mathcal{E}^*_{\text{bn}}} w(e_i) + \frac{\sum_{e_i \in \xi^*_{\text{bn}} \setminus \mathcal{E}^*_{\text{bn}}} w(e_i)}{\eta}$$
$$\sum_{e_i \in \xi^*_{\text{inf}} \setminus \mathcal{E}_{\text{bn}}} w(e_i) \leq \sum_{e_i \in \mathcal{E}^*_{\text{bn}}} w(e_i) + \frac{(1+\epsilon) w(\xi^*_{\text{dense}})}{\eta}$$

□

Figure 4.4 illustrates the samples generated by (a) ShortestPath and (b) LEGO trained with samples from BottleneckNode ; and the successful routing through narrow passages using samples from LEGO .

4.4.2 Diverse PathSet

In this training scheme, we try to ensure the roadmap is *robust* to errors introduced by the learner. One antidote to this process is *diversity* of samples. Specifically, we want the roadmap to have enough diversity

such that if the predicted shortest path is in fact infeasible, there are low cost alternates.

We set this up as a two player game between a planner and an adversary. The role of the adversary is to invalidate as many shortest paths on $\mathcal{G}_{\text{dense}}$ as possible with a fixed budget of edges that it is allowed to invalidate. The role of the planner is to find the shortest feasible path on the invalidated graph and add this to the set of diverse paths Ξ_{div}. The function $X_{\text{div}} = \textsc{DiversePathSet}(\Lambda, \mathcal{G}_{\text{dense}})$ then returns nodes belonging Ξ_{div}. We formalize this as:

Definition 4.4.2 (Diverse PathSet)**.** *We begin with a graph $\mathcal{G}^0 = \mathcal{G}_{\text{dense}}$. At each round i of the game, the adversary chooses a set of edges to invalidate:*

$$\mathcal{E}_i^* = \underset{\mathcal{E}_i \subset \mathcal{E}, |\mathcal{E}_i| \leq \ell}{\arg\max}\, w(\textsc{Alg}(\mathcal{G}^{i-1} \ominus \mathcal{E}_i, \Lambda)) \tag{4.6}$$

and the graph is updated $\mathcal{G}^i = \mathcal{G}^{i-1} \ominus \mathcal{E}_i^*$. The planner choose the shortest path $\xi_i = \textsc{Alg}(\mathcal{G}^i, \Lambda))$ which is added to the set of diverse paths Ξ_{div}.

The optimization problem (4.6) is similar to a set cover problem (NP-Hard [38]) where the goal is to select edges to cover as many paths as possible. If we knew the exact set of paths to cover, it is well known that a greedy algorithm will choose a near-optimal set of edges [38]. We have the inverse problem - we do not know how many consecutive shortest paths can be covered with a budget of ℓ edges.

Algorithm 5 describes the procedure. We greedily choose a set of edges to invalidate as many consecutive shortest paths till we exhaust our budget (Lines 8-13). We then apply greedy set cover (Line 14). If it leads to a better solution, we continue repeating the process. At termination, we ensure:

Proposition 2 (Near-optimal Invalidated EdgeSet)**.** *Let Ξ_{inv} be the contiguous set of shortest paths invalidated by Algorithm 5 using a budget of ℓ. Let ℓ^* be the size of the optimal set of edges that could have invalidated Ξ_{inv}.*

$$\ell \leq (1 + \log |\Xi_{\text{inv}}|)\ell^* \tag{4.7}$$

Proof. (Sketch) We briefly explain the equivalence to a set cover problem. Each path in Ξ_{inv} corresponds to an element that has to be covered. Each edge $e \in \mathcal{E}_{\text{dense}}$ corresponds to a set of paths in Ξ_{inv}, where each path in the set contains the edge. Invalidating the edge invalidates all paths in the set.

Line 14 invokes a greedy set cover algorithm which at every iteration chooses the edge which covers the largest number of uncovered paths. Let ℓ_{greedy} be the number of edges selected by the greedy algorithm, and ℓ^* be the optimal. From [38], we have the following near-optimality guarantee:

$$\ell_{\text{greedy}} \leq (1 + \log |\Xi_{\text{inv}}|)\ell^*$$

Algorithm 5: DIVERSEPATHSET

Input : Planning problem Λ, Pathset size k,
Dense graph $\mathcal{G}_{\text{dense}}$, Sparse graph $\mathcal{G}_{\text{sparse}}$
Output : Diverse pathset Ξ_{div}

1 $\mathcal{G} \leftarrow \mathcal{G}_{\text{dense}}$
2 $\Xi_{\text{div}} \leftarrow \emptyset$
3 **for** $i = 1, \cdots, k$ **do**
4 $\quad \Xi \leftarrow \text{ALG}^L(\mathcal{G}, \Lambda)$ ▷ L-shortest paths
5 $\quad \mathcal{E}_i \leftarrow \emptyset, \Xi_{\text{inv}} \leftarrow \emptyset$
6 $\quad$ **while** $|\mathcal{E}_i| < \ell$ **do**
7 $\quad\quad \Xi \leftarrow \{\xi \mid \xi \in \Xi, \xi \cap \mathcal{E}_i = \emptyset\}$
8 $\quad\quad$ **for** $j = |\mathcal{E}_i|, \cdots, \ell$ **do**
9 $\quad\quad\quad e_j \leftarrow \arg\max_{e \in \mathcal{E}} \min_{\xi \in \Xi, e \notin \xi} w(\xi)$ ▷ Greedy
10 $\quad\quad\quad \mathcal{E}_i \leftarrow \mathcal{E}_i \cup \{e_j\}$ ▷ Add edge to set
11 $\quad\quad\quad \Xi_j \leftarrow \left\{\xi \in \Xi \;\middle|\; e_j \in \xi\right\}$
12 $\quad\quad\quad \Xi_{\text{inv}} \leftarrow \Xi_{\text{inv}} \cup \Xi_j$ ▷ Invalidate paths
13 $\quad\quad\quad \Xi \leftarrow \Xi \setminus \Xi_j$
14 $\quad\quad \mathcal{E}_{\text{sc}} \leftarrow \text{SETCOVER}(\Xi_{\text{inv}})$ ▷ Greedy cover
15 $\quad\quad$ **if** $|\mathcal{E}_{\text{sc}}| \leq |\mathcal{E}_i|$ **then**
16 $\quad\quad\quad \mathcal{E}_i \leftarrow \mathcal{E}_{\text{sc}}$ ▷ Use better cover
17 $\quad \mathcal{G} \leftarrow \mathcal{G} \ominus \mathcal{E}_i$ ▷ Remove edges
18 $\quad \xi_i \leftarrow \text{ALG}(\mathcal{G}, \Lambda)$ ▷ New shortest path
19 $\quad \Xi_{\text{div}} \leftarrow \Xi_{\text{div}} \cup \xi_i$ ▷ Add to diverse pathset
20 **return** Ξ_{div};

If $\ell_{\text{greedy}} \leq \ell$, i.e. we have budget remaining, we continue adding edges that can only invalidate more paths in Lines 8-13. This continues till the budget is exhausted.

□

Figure 4.4 illustrates the samples generated by (a) SHORTESTPATH and (b) LEGO trained with samples from DIVERSEPATHSET ; and the robustness to unexpected obstacles exhibited by LEGO .

4.4.3 *Combining Diversity with Bottleneck Extraction*

We present LEGO in Algorithm 6 which combines the characteristics of BOTTLENECKNODE and DIVERSEPATHSET to extract a set of diverse bottleneck nodes. We first find a set of diverse paths on the dense graph (Line 1). We then iterate over each path, and adversarially invalidate edges of the sparse graph to ensure it does contain a feasible shorter path (Line 4-8). The bottleneck nodes for this path are extracted and added to the set of nodes to be returned (Line 9).

Algorithm 6: LEGO

Input : Planning problem Λ, Bottleneck tolerance ϵ, Pathset size k, Dense graph $\mathcal{G}_{\text{dense}}$, Sparse graph $\mathcal{G}_{\text{sparse}}$

Output : LEGO nodes $\mathcal{V}_{\text{lego}}$

$\Xi_{\text{div}} \leftarrow \textsc{DiversePathSet}(\Lambda, k, \mathcal{G}_{\text{dense}}, \mathcal{G}_{\text{sparse}})$

$\mathcal{V}_{\text{lego}} \leftarrow \emptyset$

while $\Xi_{\text{div}} \neq \emptyset$ **do**

  $\xi^* \leftarrow \arg\min_{\xi \in \Xi_{\text{div}}} w(\xi)$ ▷ Pick diverse path

  $\Xi_{\text{sparse}} \leftarrow \textsc{Alg}^L(\mathcal{G}_{\text{sparse}}, \Lambda)$

  $\Xi_{\text{sparse}} \leftarrow \{\xi \mid \xi \in \Xi_{\text{sparse}}, w(\xi) \leq (1+\epsilon) w(\xi^*)\}$

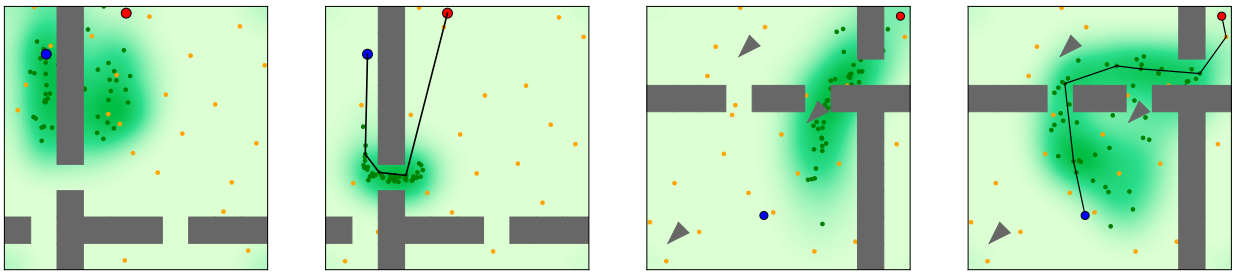
  $\mathcal{E}_{\text{inv}} \leftarrow \textsc{SetCover}(\Xi_{\text{sparse}})$ ▷ Edges to remove

  $\mathcal{G}_{\text{sparse}} \leftarrow \mathcal{G}_{\text{sparse}} \ominus \mathcal{E}_{\text{inv}}$ ▷ Invalidate paths

  $\mathcal{V}_{\text{bn}} \leftarrow \textsc{BottleneckNode}(\Lambda, \epsilon, \xi^*, \mathcal{G}_{\text{sparse}})$

  $\mathcal{V}_{\text{lego}} \leftarrow \mathcal{V}_{\text{lego}} \cup \mathcal{V}_{\text{bn}}$ ▷ Add to LEGO nodes

return $\mathcal{V}_{\text{lego}}$;



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

Figure 4.4: Comparison of samples (distribution illustrated as heatmap) generated by ShortestPath (a, c) and LEGO (b, d) in different environments. In the first environment(left), LEGO (b) is trained using BottleneckNode samples. In the second environment (right), LEGO (d) is trained using DiversePathSet samples. In both instances, ShortestPath fails to find a solution.

4.5 *Experiments and Analysis*

In this section we evaluate the performance of LEGO on various problem domains and compare it against other samplers. We consider samplers that do not assume offline computation or learning such as Medial-Axis PRM (MAPRM) [125, 55], Randomized Bridge Sampler (RBB) [57], Workspace Importance Sampler (WIS) [87], a Gaussian sampler, Gaussian [15], and a uniform Halton sequence sampler, Halton [50]. Additionally, we also compare our framework against the state-of-the-art learned sampler ShortestPath [63] upon which our work is based.

Evaluation Procedure For a given sampler and a planning problem, we invoke the sampler to generate a fixed number of samples. We then

evaluate the performance of the samplers on three metrics: a) sampling time b) success rate in solving shortest path problem and c) the quality of the solution obtained, on the graph constructed with the generated samples.

Problem Domains To evaluate the samplers, we consider a spectrum of problem domains. The $\mathbb{R}^2$ problems have random rectilinear walls with random narrow passages (Fig. 4.6(a)). These passages can be small, medium or large in width. The n-link arms are a set of n line-segments fixed to a base moving in an uniform obstacle field (Fig. 4.6(b)). The n-link snakes are arms with a free base moving through random rectilinear walls with passages (Fig. 4.6(c)). Finally, the manipulator problem has a 7DoF robot arm [113] manipulating a stick in an environment with varying clutter (Fig. 4.6(d)). Two variants are considered - constrained ($\mathbb{R}^7$), when the stick is welded to the hand, and unconstrained, when the stick can slide along the hand ($\mathbb{R}^8$).

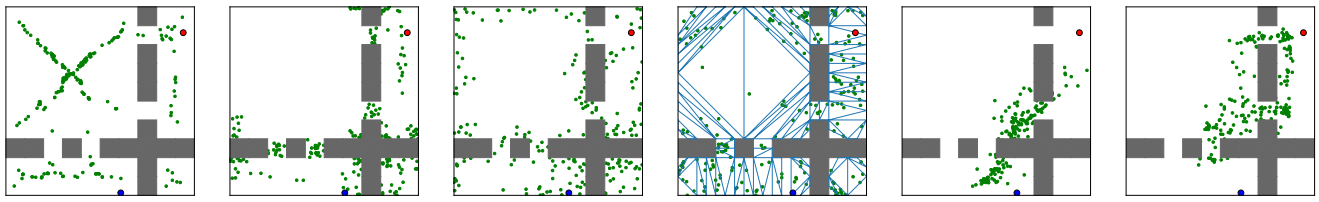


Figure 4.5: Comparison of samples (green) generated by all baseline algorithms on a 2D problem, planning from start (blue) to goal (red).

Experiment Details For the learned samplers SHORTESTPATH and LEGO , we use 4000 training worlds and 100 test worlds. Dense graph is an $r-$disc Halton graph [65]: 2000 vertices in $\mathbb{R}^2$ to $30,000$ vertices in $\mathbb{R}^8$. The CVAE was implemented in TensorFlow [37] with 2 dense layers of 512 units each. Input to the CVAE is a vector encoding source and target locations and an occupancy grid. Training time over 4000 examples ranged from 20 minutes in $\mathbb{R}^2$ to 60 minutes in $\mathbb{R}^8$ problems. At test time, we time-out samplers after 5 sec.

4.5.1 *Performance Analysis*

Sampling time Table 4.1 reports the average time each sampler takes for 200 samples across 100 test instances. SHORTESTPATH and LEGO are the fastest. MAPRM and RBB both rely on heavy computation with multiple collision checking steps. WIS , by tetrahedralizing the workspace and identifying narrow passages, is relatively faster but slower than the learners. Unfortunately, some of the baselines time-out

on manipulator planning problem due to expense of collision checking.

	Non-Learned Samplers					Learned Samplers	
	Halton	**MAPRM**	**RBB**	**Gaussian**	**WIS**	**ShortestPath**	**LEGO**
Point Robot (2D)	0.0036	0.53	0.22	0.02	0.25	0.006	0.006
N-link Arm (3D)	0.0058	–	23.96	1.95	0.36	0.016	0.016
N-link Arm (7D)	0.0071	–	37.24	3.77	1.12	0.017	0.017
Snake Robot (5D)	0.0069	39.56	142.21	3.43	0.54	0.013	0.013
Snake Robot (9D)	0.0074	40.01	180.43	8.71	2.11	0.017	0.017
Manipulator (7D)	0.0072	–	–	3.24	–	0.018	0.018
Manipulator (8D)	0.0078	–	–	3.33	–	0.018	0.018

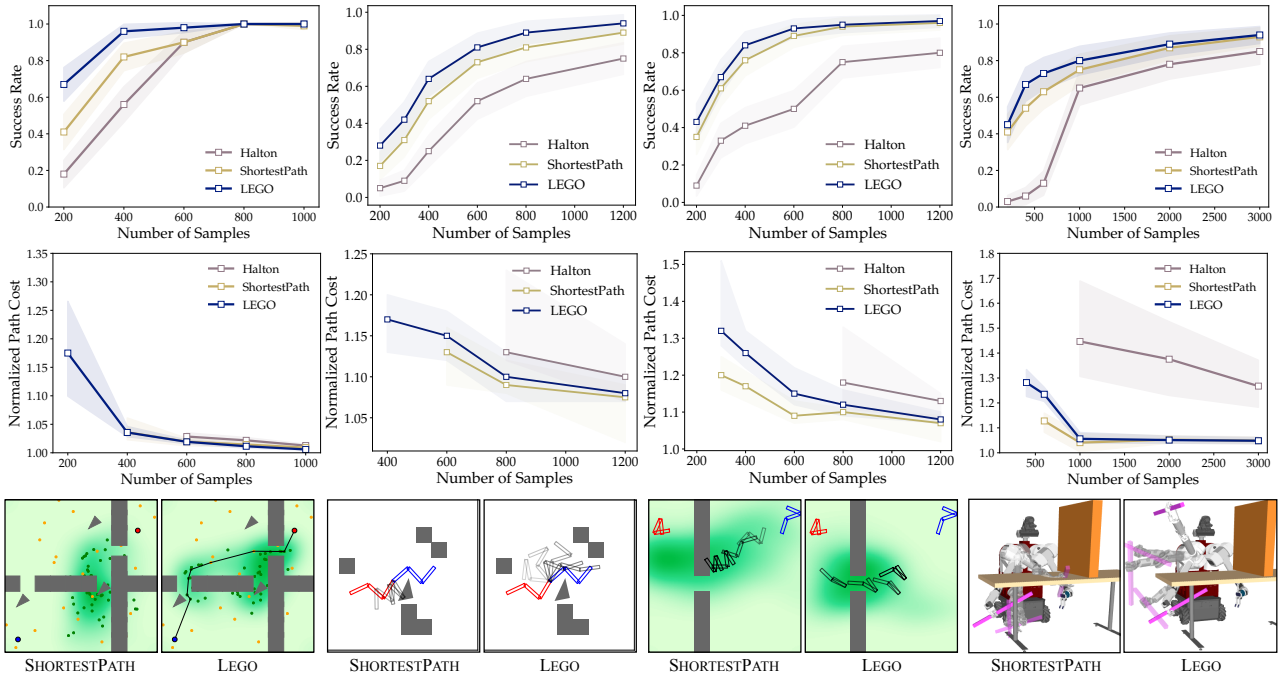
Table 4.1: Average time (sec.) by sampling algorithms to generate 200 samples over 100 planning problems

Success Rate Table 4.2 reports the success rates (95% confidence intervals) on 100 test instances when sampling 500 vertices. Success rate is the fraction of problems for which the search found a feasible solution. LEGO has the highest success rate. The baselines are competitive in $\mathbb{R}^2$, but suffer for higher dimensional problems.

	Non-Learned Samplers					Learned Samplers	
	Halton	**MAPRM**	**RBB**	**Gaussian**	**WIS**	**ShortestPath**	**LEGO**
				2D Point Robot Planning			
Large (Easy)	0.73 ± 0.08	0.73 ± 0.08	0.74 ± 0.09	0.65 ± 0.09	0.78 ± 0.08	0.86 ± 0.07	$\mathbf{0.97 \pm 0.03}$
Medium	0.48 ± 0.08	0.63 ± 0.09	0.61 ± 0.09	0.48 ± 0.10	0.63 ± 0.09	0.69 ± 0.09	$\mathbf{0.89 \pm 0.06}$
Small (Hard)	0.36 ± 0.09	0.53 ± 0.09	0.48 ± 0.08	0.32 ± 0.09	0.52 ± 0.09	0.59 ± 0.09	$\mathbf{0.83 \pm 0.07}$
				N-Link Arm			
3D	0.39 ± 0.09	–	0.54 ± 0.09	0.46 ± 0.10	0.52 ± 0.10	0.61 ± 0.09	$\mathbf{0.74 \pm 0.08}$
7D	0.29 ± 0.09	–	0.46 ± 0.09	0.41 ± 0.09	0.46 ± 0.09	0.57 ± 0.10	$\mathbf{0.71 \pm 0.08}$
				N-Link Snake Robot			
5D	0.41 ± 0.09	0.42 ± 0.09	0.48 ± 0.10	0.41 ± 0.09	0.50 ± 0.10	0.77 ± 0.08	$\mathbf{0.84 \pm 0.07}$
9D	0.49 ± 0.09	0.45 ± 0.09	0.52 ± 0.10	0.51 ± 0.10	0.53 ± 0.09	0.82 ± 0.07	$\mathbf{0.86 \pm 0.07}$
				Manipulator Arm Planning			
Unconstrained	0.24 ± 0.09	–	–	–	–	0.81 ± 0.08	$\mathbf{0.82 \pm 0.07}$
Constrained	0.09 ± 0.05	–	–	–	–	0.58 ± 0.09	$\mathbf{0.70 \pm 0.09}$

Table 4.2: Success Rates of different algorithms on 100 trials over different datasets (reported with a 95% C.I.)

Normalized Path Cost This is the ratio of cost of the computed solution w.r.t. the cost of the solution on the dense graph. Fig. 4.6 shows the normalized cost for HALTON , SHORTESTPATH and LEGO - these were the only baselines that consistently had bounded 95% confidence intervals (i.e. when success rate is $\geq$ 60%). SHORTESTPATH has the lowest cost, however LEGO is within 10% bound of the optimal.



(a) Point Robot (2D Large) (b) N-Link Arm (3D) (c) Snake Robot (5D) (d) Manipulator Constrained (7D)

Figure 4.6: Comparisons of SHORTESTPATH against LEGO on different problem domains. In each problem domain (column), success rate (top), normalized path lengths (middle) and solutions determined on the roadmaps constructed using samples generated by the two samplers (bottom) are shown.

4.5.2 *Observations*

We report on some key observations from Table 4.2 and Figure 4.6.

Observation 1. LEGO *consistently outperforms all baselines*

As shown in Table 4.2, LEGO has the best success rate (for 500 samples) on all datasets. The second row in Figure 4.6 shows that LEGO is within 10% bound of the optimal path.

Observation 2. LEGO *places samples only in regions where the optimal path may pass.*

Figure 4.5 shows samples generated by various baseline algorithms on a 2D problem. The heuristic baselines use various strategies to identify important regions - MAPRM finds medial axes, RBB finds bridge points, GAUSSIAN samples around obstacles, WIS divides up space non-uniformly and samples accordingly. However, these methods places samples everywhere irrespective of the query. SHORTESTPATH takes the query into account but fails to find the gaps. LEGO does a combination of both – it finds the right gaps.

Observation 3. LEGO *has a higher performance gain on harder problems (narrow passages) as it focuses on bottlenecks.*

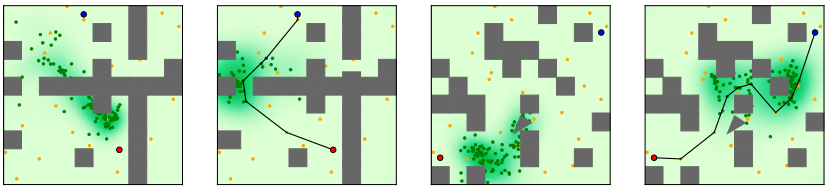
Table 4.2 shows how success rates vary in 2D problems with small / medium / large gaps. As the gaps gets narrower, LEGO out-

performs more dominantly. The BottleneckNode component in LEGO seeks the bottleneck regions (Figure 4.4(b)).

For manipulator planning $\mathbb{R}^8$ problems, when stick is unconstrained, LEGO and ShortestPath are almost identical. We attribute this to such problems being easier, i.e. the shortest path simply slides the stick out of the way and plans to the goal. When the stick is constrained, LEGO does far better. Figure 4.6(d) shows that LEGO is able to sample around the table while ShortestPath cannot find this path.

Observation 4. LEGO *is robust to a certain degree of train-test mismatch as it encourages diversity.*

Figure 4.7 shows the success rate of learners on a 2D test environment that has been corrupted. Environment 1 is less corrupted than environment 2. Figure 4.8(a) shows that on environment 1, LEGO is still the best sampler. ShortestPath (Figure 4.8(c)) ignores the corruption in the environment and fails. LEGO (Figure 4.8(d)) still finds the correct bottleneck. Figure 4.8(b) shows that all learners are worse than Halton . ShortestPath (Figure 4.8(e)) densifies around a particular constrained region while LEGO (Figure 4.8(e)) still finds a path due to the DiversePathSet component sampling in multiple bottleneck regions.

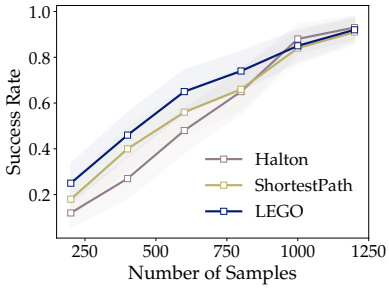


(a) (b) (c) (d)

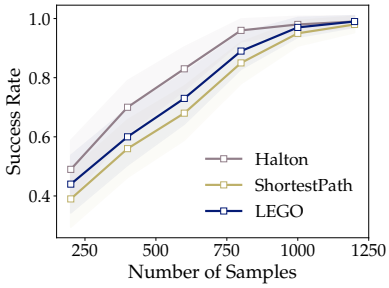
4.6 Discussion

We present a framework for training a generative model to predict roadmaps for sampling-based motion planning. We build upon state-of-the-art methods that train a CVAE using the shortest path as input. We identify failure modes such as complex obstacle configurations and train-test mismatch. Our algorithm LEGO addresses these issues by training the CVAE using *diverse bottleneck nodes* as input. We formally define these terms and provide provable algorithms to extract such nodes. Our results indicate that the predicted roadmaps outperform competitive baselines on a range of problems.

Using priors in planning is a double edged sword. While one can get astounding speed ups by focusing search on a tiny portion of C-space [62], any problem not covered in the dataset can lead to catas-



(a)



(b)

Figure 4.7: Success rate on (a) less corrupted env. Figure 4.8(a,b) and (b) more corrupted env. Figure 4.8(c,d).

Figure 4.8: Corrupted environments: (a,b) mixture of walls and random squares and (c,d) only squares. Output of ShortestPath and LEGO on the former (a,b) and the latter (c,d) respectively.

trophic failures. This is symptomatic of the fundamental problem of *over-fitting* in machine learning. While one could ensure the training data covers all possible environments [122], an algorithmic solution is to explore regularization techniques for planning. We argue DIVERSEPATHSET can be viewed as a form of regularization.

We can also include a more informed conditioning vector that captures the state of the search, e.g., the length of the current shortest path. This is similar to Informed RRT* [42]. Finally, we wish to scale to problems with varying workspace where a global planner guides the sampler to focus on relevant parts of the workspace as in [59, 129, 19].

5
Incremental Lazy Motion Planning

Lazy search algorithms such as GLS efficiently balance the costs of vertex expansion and collision checks to minimize the total planning time. However, such search-based algorithms are limited by graph approximation, and the computed path is only resolution-optimal. *Sampling*-based algorithms such as BIT* generate a series of increasingly dense graphs and employ a shortest path algorithm at each approximation. While asymptotically optimal, as graphs get denser, as observed in Chapter 3, search becomes prohibitively expensive resulting in slow convergence. In this chapter, we present Incremental Generalized Lazy Search (IGLS) to address the problem of efficient optimal motion planning. Our key insight is that by combining the efficiency of lazy search with batch-informed sampling, we can improve the convergence rate of anytime asymptotically optimal sampling-based algorithms. IGLS represents a family of lazy asymptotically-optimal algorithms generalizing existing algorithms such as BIT* and AIT*. While the previous chapters approximated optimal motion planning as search for the shortest path on a discrete graph, this chapter aims to compute the optimal solution in the continuous space, albeit asymptotically.

5.1 Introduction

Sampling-based algorithms have shown tremendous success in solving complex high-dimensional robot motion planning algorithms while guaranteeing asymptotic optimality [73]. While these algorithms compute a feasible solution quickly, improving the convergence rate to the optimal solution, especially in high-dimensional cluttered environments, has been a central topic of research [67, 54]. A popular approach in this class of algorithms is to sample increasingly dense graphs to approximate the robot's configuration space and employ a shortest path algorithm to compute the resolution-optimal solution [44, 41]. As the graphs get denser, the algorithm improves the solution in an anytime-fashion. These algorithms have been shown to be proba-

bilistically complete and almost-surely asymptotically optimal.

The underlying shortest path search in these algorithms typically involves two atomic operations that define its computational effort: vertex expansions and collision checking (Chapter 3, Section 3.2). In high-dimensional problems such as robot motion planning, collision checking is generally the bottleneck operation [54]. However, as the algorithm samples increasingly dense graphs, the cost of vertex expansions increases as well. Since the search effort is the sum of time spent on the two operations, we require striking a balance between the two operations to minimize computational effort.

As studied in Chapter 3, GLS introduces an EVENT-based toggle that interleaves lazy search with collision checks, and thus provides a framework that balances graph operations with edge evaluations to minimize planning time. Since GLS operates on a fixed resolution graph, it can only guarantee resolution-optimal paths.

In this chapter, we extend GLS to propose Incremental Generalized Lazy Search (IGLS), a framework of lazy sampling-based motion planners. While the sampling-based planning framework allows to asymptotically improve the graph abstraction of the problem, GLS provides a framework to minimize computation cost of search. We effectively extend the efficiency gains of GLS by balancing vertex expansions and collision checks in each iteration, to minimize the *total planning time* in converging to the optimal solution.

5.2 *Problem Formulation*

In this chapter we are concerned with the optimal motion planning problem introduced in Section 2.1. We revisit some mathematical notation. Let $\mathcal{X} \subseteq \mathbb{R}^n$ be the statespace of the planning problem and $\mathcal{X}_{\text{obs}} \subset \mathcal{X}$ denote the subspace occupied by obstacles. The free space is denoted by $\mathcal{X}_{\text{free}} = \mathcal{X} \backslash \mathcal{X}_{\text{obs}}$. Given source and target states $(v_s,\ v_t) \in \mathcal{X}_{\text{free}}$, a path $\xi : [0,1] \to \mathcal{X}$ is represented as a sequence of states such that $\xi(0) = v_s$ and $\xi(1) = v_t$. Let Ξ be the set of all non-trivial paths. A motion is considered valid if it lies completely within $\mathcal{X}_{\text{free}}$ i.e. $\forall t \in [0,1],\ \xi(t) \in \mathcal{X}_{\text{free}}$. Given a cost function $c : \xi \to \mathbb{R}_{>0}$, an optimal path ξ^* is a feasible path minimizing the objective function:

$$\begin{aligned} \xi^* = \underset{\xi \in \Xi}{\arg\min}\ c(\xi) \quad s.t. \\ \xi(0) = v_s,\ \xi(1) = v_t,\ \forall t \in [0,1],\ \xi(t) \in \mathcal{X}_{\text{free}} \end{aligned} \tag{5.1}$$

5.3 *Incremental Generalized Lazy Search*

We propose Incremental Generalized Lazy Search (IGLS), an informed asymptotically-optimal anytime search algorithm to solve the optimal motion planning problem described in Section 5.2. The general idea is to iteratively construct graphs from samples in the configuration space and employ lazy search to compute the shortest path on the graphs while minimizing computational effort.

5.3.1 *Notation and Algorithmic Details*

Our algorithm constructs a lazy shortest path search tree over available samples and evaluates edges in the tree to compute a collision-free shortest path. The set of vertices in the search tree is denoted by $\mathcal{V}$ and of those samples yet unconnected is denoted by $\mathcal{S}$. Every sample x is associated with a parent $p(x)$ which is valid if $x \in \mathcal{V}$. Let $g : \mathcal{X} \to \mathbb{R}_{\geq 0}$ denote the cost-to-come from the source v_s to a sample x via vertices in the search tree i.e. $g(x) = \sum_{e \in \xi_x} c(e)$. The cost-to-come in finite only for vertices in the search tree. Let $h : \mathcal{X} \to \mathbb{R}_{\geq 0}$ denote an admissible heuristic i.e. a lower bound on the cost-to-go, from a sample in the configuration space to the target v_t. Therefore, each sample x in the search tree is associated with $f : \mathcal{X} \to \mathbb{R}_{\geq 0}$ which represents the cost of a path from source to target through x.

In addition to the parent, each sample x also keeps track of its parents in the search trees from previous batches, $\mathcal{N}(x)$. The connection radius is a function of the number of samples in the graph, and generally reduces with iterations. Keeping track of previous parents helps take advantage of long edges from previous batches. Evaluated edges are also tracked as the set $\mathcal{E}_{\text{eval}}$.

The algorithm maintains two priority queues to efficiently process vertices: $\mathcal{Q}_{\text{extend}}$ stores leaf vertices in the search tree for expansion. $\mathcal{Q}_{\text{rewire}}$ stores vertices that need to be rewired. $\mathcal{Q}_{\text{rewire}}$ is populated when an edge in the lazy search tree is evaluated to be in collision and the subsequent subtree needs to be repaired for consistency. The algorithm uses Events and Selectors introduced in the GLS (Chapter3) framework to efficiently toggle between search and evaluation and reduce the computational effort in search.

5.3.2 *The Algorithm*

Algorithm 7 describes the overview of IGLS. The algorithm takes as input the source-target pair (v_s, v_t) and the collision checking module ϕ. Additionally, it borrows parameters Event and Selector from GLS framework. The BatchSize specifies the number of samples to generate in each batch of IGLS. The algorithm begins by populating

Algorithm 7: IGLS

Input : source v_s, target v_t, collision checker ϕ

Parameter: Event, Selector, BatchSize

Output : $\{\xi_1, \xi_2, \ldots, \xi^*\}$ *s.t.* $\forall i,\ c(\xi_{i+1}) < c(\xi_i)$

1 $\mathcal{S} \leftarrow \{v_t\},\ \mathcal{V} \leftarrow \{v_s\},\ \mathcal{Q}_{\text{extend}} \leftarrow \emptyset$
2 **repeat**
3 $\mathcal{Q}_{\text{extend}} \xleftarrow{+} \mathcal{V}$ ▷ Initialize queue
4 ComputeConnectionRadius($|\mathcal{V}| + |\mathcal{S}|$)
5 **repeat**
6 Search(Event) ▷ Lazy Search
7 **if** $\mathcal{Q}_{\text{extend}}$ *is empty* **then**
8 **break**
9 Evaluate(Selector) ▷ Expensive
10 **until** ξ_{v_t} *s.t.* $\forall e \in \xi_{v_t},\ e \in \mathcal{E}_{\text{eval}},\ \phi(e) = 1$;
11 Prune($\mathcal{S},\ \mathcal{V},\ g(v_t)$)
12 $\mathcal{S} \xleftarrow{+}$ Sample(BatchSize, $g(v_t)$)
13 $\mathcal{Q}_{\text{extend}} \leftarrow \emptyset$
14 **until** *forever*;

the unconnected samples set $\mathcal{S}$ with the target v_t and the connected vertices set $\mathcal{V}$ with the source v_s (line 1). The algorithm interleaves sampling increasingly dense graphs with search for the shortest path.

The algorithm proceeds by populating $\mathcal{Q}_{\text{extend}}$ with the vertices in the search tree, $\mathcal{V}$ (Algorithm 7, Line 3). $\mathcal{Q}_{\text{extend}}$ processes vertices in increasing $f(\cdot)$ values. The total number of samples in the edge-implicit graph defines the connection radius (Algorithm 7, Line 4). The connection radius is chosen to ensure asymptotic optimality [70]. With an updated connection radius, the edge-implicit graph is completely defined allowing for a shortest path algorithm to compute the optimal path on the graph (Lines 5-10). If the algorithm improves the current solution cost, it prunes samples outside the updated Informed Set [43]. The algorithm then samples a new batch of states within the Informed Set to improve the approximation of the statespace. The datastructures are reinitialized before the next iteration. This continues to asymptotic optimality.

Lazy Search GLS is the search algorithm of choice (Algorithm 8). The difference from GLS (Appendix A, Algorithm 14) is in the way nearest neighbors are queried (highlighted in red). In addition to computing

Algorithm 8: SEARCH

while $\mathcal{Q}_{\text{extend}}$ *is not empty* **do**
  if $\mathcal{Q}_{\text{extend}}$.FRONT() *triggers* EVENT **then**
    return
  $u \leftarrow \mathcal{Q}_{\text{extend}}$.POP()
  $\mathcal{N}_{\mathcal{G}} \leftarrow \{v \mid v \in \mathcal{S} \cup \mathcal{V}, \|v - u\|_2 \leq r\}$
  $\mathcal{N}_{\text{eval}} \leftarrow \{v \mid (u, v) \in \mathcal{E}_{\text{eval}}\}$
  forall $v \in \mathcal{N}_{\mathcal{G}} \cup \mathcal{N}_{\text{eval}}$ **do**
    if $(u, v) \in \mathcal{E}_{\text{eval}}$ *and* $\phi(u, v) = 0$ **then**
      continue
    if $g(u) + c(u, v) < g(v)$ **then**
      $p(v) \leftarrow u$
      $g(v) \leftarrow g(u) + c(u, v)$
    if $v \in \mathcal{S}$ **then**
      $\mathcal{S} \xleftarrow{-} \{v\}$, $\mathcal{V} \xleftarrow{+} \{v\}$
      $\mathcal{Q}_{\text{extend}}$.PUSH($v$)
    else
      $\mathcal{Q}_{\text{extend}}$.UPDATEKEY($v$)

the nearest neighbors using the r-disk, since the connection radius is constantly changing, we also consider evaluated edges from previous iterations that are no longer within the current r-disk. This allows us to reuse computation from previous expensive edge evaluations. The lazy search continues until the EVENT triggers. If the EVENT is chosen to CONSTANTDEPTH with depth unity, we retrieve the behavior of BIT* [40]. Similarly, with SHORTESTPATH as the EVENT, we retrieve AIT* [115].

Edge Evaluation The algorithm toggles from lazy search to edge evaluation (Algorithm 9) with the EVENT trigger. The SELECTOR chooses an edge in the search tree to evaluate. If the edge is collision-free, lazy search continues until EVENT triggers again or until termination. If the edge is in collision, the subtree at the edge is repaired using LPA* [82]. The algorithm tracks the evaluated edges.

Repair The subtree rooted at the edge found to be in collision is repaired by processing the vertices using $\mathcal{Q}_{\text{rewire}}$. The queue processes vertices in the increasing order of $f(\cdot)$ values. As in the SEARCH() procedure, the difference from GLS (Chapter A, Algorithm 10) is in con-

Algorithm 9: EVALUATE

$v' \leftarrow \mathcal{Q}_{\text{extend}}.\text{FRONT}()$
$e(u,v) \leftarrow \text{SELECTOR}(\xi_{v_s,v'})$
$\mathcal{E}_{\text{eval}} \xleftarrow{+} \{e\}$
if $\phi(e) = 0$ **then**
  $\mathcal{T}_{\text{repair}} \leftarrow \mathcal{T}_{\text{sub}}(v)$
  REPAIR()

sidering the nearest neighbors to rewire to (Algorithm 10, Lines6, 17). The cached edge evaluations provide additional neighbors the vertices can consider in addition to the available neighbors defined by the current r-disk. At the end of this procedure, the search tree is consistent with the edge found to be in collision.

Algorithm 10: REPAIR

forall $v \in \mathcal{T}_{\text{repair}}$ **do**
  if $v \in \mathcal{Q}_{\text{extend}}$ **then**
    $\mathcal{Q}_{\text{extend}}.\text{REMOVE}(v)$
  $p(v) \leftarrow \texttt{NIL}$, $g(v) \leftarrow \infty$
  $\mathcal{V}_{\text{parents}} \leftarrow \{u \in \mathcal{V} \mid \|v-u\|_2 \leq r\} \cup \mathcal{N}(v)$
  forall $\{u \mid u \in \{\mathcal{N}_{\mathcal{G}} \cup \mathcal{N}_{\text{eval}}\} - \{\mathcal{T}_{\text{repair}} \cup v_t\}\}$ **do**
    if $g(u) + c(u,v) < g(v)$ **then**
      $p(v) \leftarrow u$
      $g(v) \leftarrow g(u) + c(u,v)$
  $\mathcal{Q}_{\text{rewire}}.\text{PUSH}(v)$
while $\mathcal{Q}_{\text{rewire}}$ *is not empty* **do**
  $u \leftarrow \mathcal{Q}_{\text{rewire}}.\text{POP}()$
  if $p(u) = \texttt{NIL}$ **then**
    $\mathcal{V} \xleftarrow{-} \{u\}$, $\mathcal{S} \xleftarrow{+} \{u\}$
    continue
  if u *does* ***not*** *trigger* EVENT **then**
    forall $\{v \mid v \in \mathcal{N}_{\mathcal{G}} \cup \mathcal{N}_{\text{eval}},\ v \in \mathcal{Q}_{\text{rewire}}\}$ **do**
      if $g(u) + c(u,v) < g(v)$ **then**
        $p(v) \leftarrow u$
        $g(v) \leftarrow g(u) + c(u,v)$
        $\mathcal{Q}_{\text{rewire}}.\text{UPDATEKEY}(v)$
  $\mathcal{Q}_{\text{extend}}.\text{PUSH}(u)$
$\mathcal{T}_{\text{repair}}.\text{CLEAR}()$
return

5.4 *Discussion*

Asymptotically optimal algorithms interleave *sampling* and *search* to compute the optimal solution in the continuous space. This separation of the two operations allows us to independently optimize the two operations to improve the convergence rate of the algorithm. Irrespective of the sampling strategy, each iteration of these algorithms involves invoking a search for the shortest path. In this chapter, we focused on extending the efficiency that GLS offers in minimizing the computational effort in solving the SSSP problem. By choosing appropriate EVENT and SELECTOR definitions, IGLS allows to minimize the planning time in each iteration of the algorithm thereby improving the convergence rate.

This leaves open the question of improving the sampling strategy for each iteration. While in Chapter 4 we investigated construction of sparse graphs with adequate coverage in narrow passages, this does not yet guarantee the *optimal* solution, especially in the absence of prior information. In the remainder of this thesis, we focus on improving the convergence rate by adapting the sampling scheme with the planning procedure.

6
Guided Incremental Local Densification

Sampling-based motion planners rely on incremental densification to discover progressively shorter paths. After computing feasible path ξ between start x_s and goal x_t, the *Informed Set* (IS) prunes the configuration space $\mathcal{X}$ by conservatively eliminating points that cannot yield shorter paths. Densification via sampling from this Informed Set retains asymptotic optimality of sampling from the entire configuration space. For path length $c(\xi)$ and Euclidean heuristic h, $IS = \{x | x \in \mathcal{X}, h(x_s, x) + h(x, x_t) \leq c(\xi)\}$.

Relying on the heuristic can render the IS especially conservative in high dimensions or complex environments. Furthermore, the IS only shrinks when shorter paths are discovered. Thus, the computational effort from each iteration of densification and planning is wasted if it fails to yield a shorter path, despite improving the cost-to-come for vertices in the search tree. Our key insight is that even in such a failure, shorter paths to *vertices in the search tree* (rather than just the goal) can immediately improve the planner's sampling strategy. Guided Incremental Local Densification (GuILD) leverages this information to sample from *Local Subsets* of the IS. tasks in $\mathbb{R}^7$.

6.1 Introduction

Sampling-based algorithms have shown tremendous success in solving complex high-dimensional robot motion planning problems. These algorithms achieve asymptotic-optimality by incrementally densifying the robot's configuration space to continually improve the shortest path in an anytime manner [73, 71, 5, 44, 117, 115]. While such algorithms often compute an initial feasible path quickly, their convergence is slow because they need to sample a huge number of configurations before a shorter path is found.

While the configuration space can be densified uniformly to find shorter paths, the current best path ξ between the start x_s and target x_t defines an *Informed Set* $\mathcal{X}_{\text{inf}} \subseteq \mathcal{X}$ [42] that contains only states that

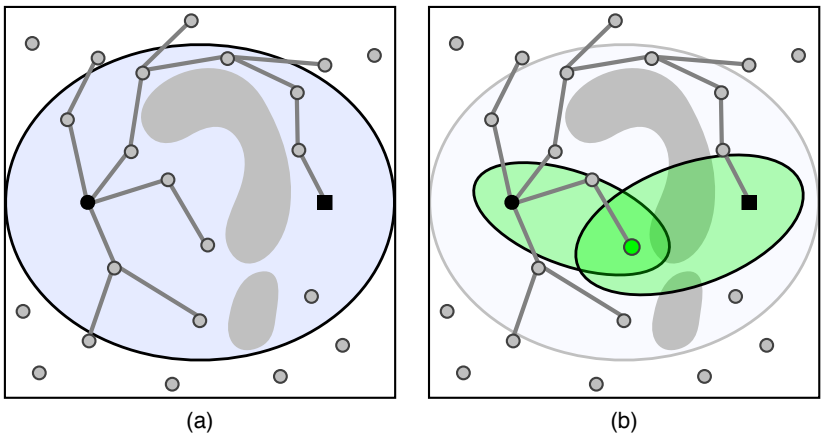


Figure 6.1: Comparison of (a) Informed Set and (b) *Local Subsets* induced by GuILD. To further focus sampling within the Informed Set, GuILD chooses a beacon (green) and decomposes the original problem into two smaller problems using information from the search tree.

can yield shorter paths:

$$\mathcal{X}_{\text{inf}} = \{x \mid x \in \mathcal{X}, h(x_s, x) + h(x, x_t) \leq c(\xi)\}, \tag{6.1}$$

where $c(\xi)$ is the cost of the current path and h is an admissible heuristic. For problems minimizing path length in $\mathbb{R}^n$, the Informed Set with the Euclidean distance heuristic is an n-dimensional prolate hyperspheroid $\mathcal{E}_{\text{IS}}$, parameterized by foci x_s and x_t and transverse axis diameter $c(\xi)$ (i.e., a generalized ellipse as seen in Figure 6.1a). Sampling from $\mathcal{E}_{\text{IS}}$ preserves the asymptotic optimality guarantee of sampling from the entire configuration space $\mathcal{X}$.

In practice, however, the Informed Set often provides limited sample efficiency improvement. Since the measure of the Informed Set $\lambda(\mathcal{E}_{\text{IS}})$ is a function of the current solution cost, a high cost solution may yield a large Informed Set with comparable (or greater) measure to the full state space $\lambda(\mathcal{X})$. Paradoxically, to reap the greatest benefits of sampling from the Informed Set, the planner must already have a path of sufficiently small cost. This is a particular challenge for planning problems in high dimensions or cluttered environments. Furthermore, the computational effort from each iteration of densification and planning is wasted if it fails to yield a shorter path.

Our key insight is that even in such a failure, we can open the black box of the planning algorithm to immediately improve the planner's sampling strategy. With **Gu**ided **I**ncremental **L**ocal **D**ensification (GuILD), shorter paths to *any vertex in the search tree* can guide further sampling. GuILD introduces the idea of a beacon, a vertex in the search tree that decomposes the original sampling/planning problem into two smaller subproblems (Figure 6.1b). Much like the Informed Set between the start and target, the beacon induces *Local Subsets* with (i) start and beacon as foci and (ii) beacon and target as foci. GuILD

leverages improvements to the search tree to adapt the Local Subsets and converge to the optimal path with fewer samples (Figure 6.2).

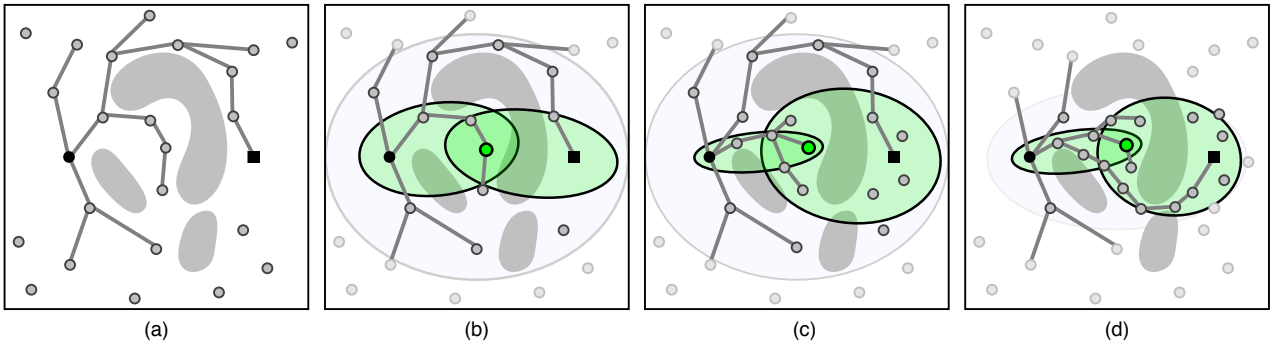


Figure 6.2: **(a)** Initial solution. **(b)** GuILD selects a beacon and induces Local Subsets (green), which do not cover the narrow passage. **(c)** The planner does not find a shorter path to the goal, so the Informed Set is unchanged. However, GuILD leverages the improved cost-to-come in the search tree to update the Local Subsets. The start-beacon set shrinks to further focus sampling, and the remaining slack between the beacon's and goal's cost-to-comes is used to *expand* the beacon-target set. The Local Subsets now cover the narrow passage, focusing sampling within the Informed Set to quickly converge to **(d)** the optimal solution.

We make the following contributions:

- We introduce GuILD, an incremental densification framework that effectively leverages search tree information to focus sampling.
- We compare theoretical properties of the Local Subsets that GuILD samples to the Informed Set.
- We propose several BeaconSelector strategies for GuILD, including an adversarial bandit algorithm.
- We show experimentally that regardless of the BeaconSelector, GuILD outperforms the state-of-the-art Informed Set densification baseline across a range of planning domains. In particular, GuILD yields modest improvements in simpler planning domains and excels in domains with difficult-to-sample homotopy classes.

6.2 *Problem Formulation*

In this section, we formally introduce the problem of sampling-based optimal motion planning (optimal SBMP). Let $\mathcal{X} \subseteq \mathbb{R}^n$ be the statespace of the planning problem, where $\mathcal{X}_{\text{obs}} \subset \mathcal{X}$ is the subspace occupied by obstacles and the free space is $\mathcal{X}_{\text{free}} = \mathcal{X} \backslash \mathcal{X}_{\text{obs}}$. Given source and target states $x_s, x_t \in \mathcal{X}_{\text{free}}$, a path $\xi : [0,1] \to \mathcal{X}$ is represented as a sequence of states such that $\xi(0) = x_s$ and $\xi(1) = x_t$. Let $\Xi(x_s, x_t)$ be the set of all such paths. Given a cost function $c : \xi \to \mathbb{R}_{>0}$, an optimal collision-free path is defined as:

$$\xi^* = \underset{\xi \in \Xi(x_s, x_t)}{\arg\min} \; c(\xi) \quad s.t. \quad \forall t \in [0,1], \; \xi(t) \in \mathcal{X}_{\text{free}}.$$

Optimal SBMP algorithms progressively improve a roadmap approximation of the state space to plan asymptotically-optimal paths

(Algorithm 11). In each iteration, states are sampled from $\mathcal{X}_{\text{free}}$ to grow the vertices of an edge-implicit[1] graph $\mathcal{G}$ (Line 4). Then, a shortest path algorithm computes the *resolution-optimal* path on $\mathcal{G}$, internally using an admissible heuristic h to focus the search (Line 5). If densification produces a lower-cost path, the best solution cost is updated and that path is emitted as ξ_i (Lines 6-8). As the discrete graph $\mathcal{G}$ more closely approximates the continuous state space, the sequence of resolution-optimal paths $\{\xi_1, \xi_2, \cdots\}$ approaches the optimal path ξ^*.

[1] The radius of connectivity of the implicit graph is chosen to ensure asymptotic optimality [73].

6.2.1 *Informed Incremental Densification*

Given a candidate state $x \in \mathcal{X}_{\text{free}}$ and the admissible search heuristic h, the cost of all paths between x_s and x_t that pass through x is lower bounded by $h(x_s, x) + h(x, x_t)$. Defined by Equation 6.1, the Informed Set (IS) excludes states for which this lower bound exceeds the best solution cost $c(\xi)$. Unlike sampling the entire state space $\mathcal{X}_{\text{free}}$, sampling from the IS automatically excludes states that cannot be part of a lower-cost path [42].

Euclidean distance, an admissible heuristic for path length, admits a concise geometric interpretation of the IS: an n-dimensional prolate hyperspheroid $\mathcal{E}_{\text{IS}} = \mathcal{E}(x_s, x_t, c(\xi))$ with foci x_s, x_t and transverse axis diameter $c(\xi)$.

$$\mathcal{E}_{\text{IS}} = \{x \mid x \in \mathcal{X}, \|x_s - x\|_2 + \|x - x_t\|_2 \leq c(\xi)\} \tag{6.2}$$

Prolate hyperspheroids can be sampled analytically, rather than via rejection sampling [42].

Sampling from the IS is a sufficient condition to converge to ξ^*. The improved efficiency can be characterized by comparing the measure $\lambda(\mathcal{E}_{\text{IS}})$ to the measure of the full state space $\lambda(\mathcal{X})$. When $\lambda(\mathcal{E}_{\text{IS}}) < \lambda(\mathcal{X})$, the IS can yield significant improvements on sample efficiency. However, when the initial solution has high path length (e.g., due to a cluttered environment), $\lambda(\mathcal{E}_{\text{IS}})$ may instead be closer to—or even larger than—$\lambda(\mathcal{X})$. This issue is exacerbated because the IS only shrinks when shorter paths are found. As a result, each iteration of densification and search that fails to find a shorter path does not affect the state sampling distribution. In the next section, we introduce a new strategy that leverages this previously-wasted computational effort.

6.3 *Guided Incremental Local Densification*

We present GuILD, an incremental densification framework that leverages partial search information to focus sampling within the IS. If an iteration of densification and planning has failed to improve the solution cost, what information is there for GuILD to take advantage

Algorithm 11: Informed Optimal SBMP

Input : start v_s, goal v_t

Output : $\{\xi_1, \xi_2, \cdots\}$ s.t. $c(\xi_{i+1}) < c(\xi_i)$

1 Initialize best solution cost: $c(\xi) \leftarrow \infty$

2 Initialize edge-implicit graph: $\mathcal{G} \leftarrow \{v_s, v_t\}$

3 **repeat**

4 Densify: $\mathcal{G} \xleftarrow{+}$ `Sample`$(v_s,\ v_t,\ c(\xi))$

5 Compute shortest path: $\hat{\xi} \leftarrow$ `Search`$(v_s,\ v_t,\ \mathcal{G})$

6 **if** $c(\hat{\xi}) = g(v_t) < c(\xi)$ **then**

7 Update best solution cost: $c(\xi) \leftarrow g(v_t)$

8 Emit current solution $\hat{\xi}$

9 **until** *forever*;

of?

Our key insight is to open the black box of the underlying search algorithm. Although the iteration may not have found a shorter path to the goal, new shorter paths to *other vertices in the search tree* can immediately improve the sampling strategy (Algorithm 11, Line 4).

6.3.1 *Guiding Densification with Search Tree Information*

During search (Algorithm 11, Line 5), the algorithm internally expands vertices in $\mathcal{G}$ to construct a search tree $\mathcal{T}$. Each vertex $v \in \mathcal{T}$ is associated with a parent vertex, a cost-to-come $g(v)$ via that parent, and a cost-to-go heuristic estimate $h(v, v_t)$. When new states are sampled and added to $\mathcal{G}$, they may also be added to $\mathcal{T}$. Other vertices may then discover that their cost-to-come would be reduced by updating their parent to this new vertex [73, 44]. While the IS only shrinks when these changes propagate all the way to v_t, GuILD leverages *any* cost-to-come improvement to adaptively guide densification.

GuILD introduces the idea of a *beacon*: a vertex in the search tree that decomposes the original sampling/planning problem into two smaller subproblems (Figure 6.3). We define the *Local Subsets* (LS) induced by beacon b to be the union of two prolate hyperspheroids, with foci (v_s, b) and foci (b, v_t). The start-beacon set has transverse axis diameter $g(b)$, only including points that can improve the cost-to-come from v_s to b. Given the current shortest subpath to b and its cost-to-come, the beacon-target set only includes points that can extend that

subpath and improve the solution cost by setting the transverse axis diameter to $c(\xi) - g(b)$.

GuILD adapts $\mathcal{E}_{\text{LS}}$ after each iteration of densification and planning. When $g(b)$ is reduced, the start-beacon set shrinks and the beacon-target set *expands*. Surprisingly, this expansion is actually desirable: if the best solution cost remains the same and the beacon's cost-to-come is reduced, there is a larger path length budget that can be expended between the beacon and target that could still yield a shorter path overall (Figure 6.2).

We summarize the GuILD framework in Algorithm 12, which replaces the highlighted line in Algorithm 11. The BeaconSelector (Algorithm 12, Line 4) is a function that chooses a beacon to guide local densification. The beacon must have been expanded by the underlying search algorithm to preserve the sampling guarantees that we prove in the next section.[2] GuILD samples uniformly from $\mathcal{E}_{\text{LS}}$ (Line 6) using the same analytic strategy proposed for $\mathcal{E}_{\text{IS}}$ [41].

[2] GuILD must sample from the IS with nonzero probability to retain asymptotic optimality. This is achieved by including v_{s} in the beacon set.

In Algorithm 13, we present some candidate beacon selectors that we evaluate in Section 6.4. The `InformedSet` beacon selector recovers the behavior of sampling from the IS by choosing v_{s} as the beacon. Uniform-GuILDuniformly samples from the beacon set $\mathcal{B}$. The Greedy-GuILDbeacon selector aims to select the beacon with the maximum possible improvement in path length, while minimizing the measure of the set that needs to be sampled. It takes the ratio of these two quantities:

$$w(b) = \frac{c(\xi) - h(v_{\text{s}}, b) - h(b, v_{\text{t}})}{\lambda(\mathcal{E}_{\text{LS}})}. \tag{6.3}$$

Finally, the Bandit-GuILDbeacon selector implements the EXP3 adversarial bandit algorithm [6]. The bandit reward function is the fractional improvement in the path length

$$r(b) = \frac{c(\xi_{i-1}) - c(\xi_i)}{c(\xi_{i-1})}.$$

An adversarial bandit algorithm is necessary because this reward function is nonstationary.

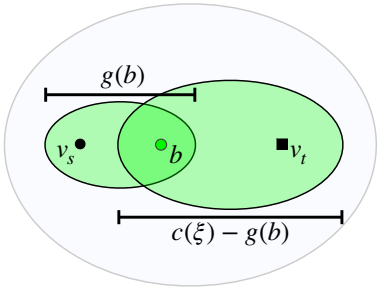


Figure 6.3: Local Subsets $\mathcal{E}_{\text{LS}}$ (green) are defined by a beacon b, as well as the cost-to-come on the search tree $g(b)$ and the current best solution cost $c(\xi)$.

6.3.2 *Properties of Local Subsets*

First, we guarantee that GuILD will never draw samples outside the IS by showing that both sets in $\mathcal{E}_{\text{LS}}$ are subsets of the IS (Equation 1). Then, we show that the measure of $\mathcal{E}_{\text{LS}}$ is upper-bounded by that of the IS (Equation 2), so choosing the appropriate beacon will allow GuILD to sample the space more efficiently.

Algorithm 12: GUILD

Input : beacon set $\mathcal{B}$, search tree $\mathcal{T}$, graph $\mathcal{G}$

Output : densified graph $\mathcal{G}$

if *a solution does not yet exist* **then**
  $\mathcal{G} \xleftarrow{+}$ `UniformSample(`$\mathcal{X}_{\text{free}}$`)`
else
  $b \leftarrow$ BEACONSELECTOR($\mathcal{B}$)
  $\mathcal{E}_{\text{LS}} \leftarrow \mathcal{E}(v_{\text{s}}, b, g(b)) \cup \mathcal{E}(b, v_{\text{t}}, c(\xi) - g(b))$
  $\mathcal{G} \xleftarrow{+}$ `UniformSample(`$\mathcal{E}_{\text{LS}}$`)`
return $\mathcal{S}$

Theorem 1. *Given the current best solution cost $c(\xi_i)$, a beacon $b \in \mathcal{T}$ previously expanded with cost-to-come $g(b)$, we have that $\mathcal{E}(v_{\text{s}}, b, g(b))$, $\mathcal{E}(b, v_{\text{t}}, c(\xi_i) - g(b)) \subseteq \mathcal{E}_{\text{IS}}$.*

Proof. Let $x \in \mathcal{E}(v_{\text{s}}, b, g(b))$. By construction of the prolate hyperspheroid and the triangle inequality:

$$\begin{aligned} h(v_{\text{s}}, x) + h(x, v) &\leq g(b) \\ h(v_{\text{s}}, x) + h(x, b) + h(b, v_{\text{t}}) &\leq g(b) + h(b, v_{\text{t}}) \\ h(v_{\text{s}}, x) + h(x, v_{\text{t}}) &\leq g(b) + h(b, v_{\text{t}}) \end{aligned}$$

Since b was previously expanded,

$$h(v_{\text{s}}, x) + h(x, v_{\text{t}}) \leq g(b) + h(b, v_{\text{t}}) \leq c(\xi_i)$$

Therefore, we have $\mathcal{E}(v_{\text{s}}, b, g(b)) \subseteq \mathcal{E}_{\text{IS}}$.

Similarly, let $x \in \mathcal{E}(v_{\text{s}}, b, c(\xi_i) - g(b))$. We have,

$$\begin{aligned} h(b, x) + h(x, v_{\text{t}}) &\leq c(\xi_i) - g(b) \\ g(b) + h(b, x) + h(x, v_{\text{t}}) &\leq c(\xi_i) \\ h(v_{\text{s}}, x) + h(x, v_{\text{t}}) &\leq c(\xi_i) \end{aligned}$$

Therefore, we have $\mathcal{E}(v_{\text{s}}, b, c(\xi_i) - g(b)) \subseteq \mathcal{E}_{\text{IS}}$. □

Theorem 2. *Given the current best solution cost $c(\xi_i)$, a beacon $b \in \mathcal{T}$ previously expanded with cost-to-come $g(b)$, we have $\lambda(\mathcal{E}_{\text{LS}}) \leq \lambda(\mathcal{E}_{\text{IS}})$*

Proof. A prolate hyperspheroid $\mathcal{E} \in \mathbb{R}^n$ with transverse diameter a and distance between the foci f has measure $\lambda(\mathcal{E}) = \mathcal{L} a \left(a^2 - f^2\right)^{\frac{n-1}{2}}$ where $\mathcal{L} = \frac{\pi^{\frac{n}{2}}}{2^n \Gamma(\frac{n}{2}+1)}$ is only dimension-dependent. Let a_s, f_s and a_t, f_t denote the parameters corresponding to $\mathcal{E}(v_{\text{s}}, b, g(b))$ and $\mathcal{E}(b, v_{\text{t}}, c(\xi_i) - g(b))$

Algorithm 13: Candidate BeaconSelectors

Input : beacon set $\mathcal{B}$

Function `InformedSet`
  return v_s;
Function `Uniform`
  return $b \sim \mathcal{U}(\mathcal{B})$;
Function `Greedy`
  return $\arg\max_{b \in \mathcal{B}} w(b)$ ▷ Equation 6.3
Function `Bandit`
  return $\text{EXP3}(\mathcal{B})$

respectively. We have the following:

$$\begin{aligned} a_s + a_t &= a_{\text{IS}} \quad \text{(by definition)} \\ f_s + f_t &\geq f_{\text{IS}} \quad \text{(triangle inequality)} \end{aligned}$$

Consider the measure of the IS:

$$\begin{aligned} \lambda(\mathcal{E}_{\text{IS}}) &= \mathcal{L} a_{\text{IS}} \left(a_{\text{IS}}^2 - f_{\text{IS}}^2 \right)^{\frac{n-1}{2}} \\ &\geq \mathcal{L}(a_s + a_t) \left((a_s + a_t)^2 - (f_s + f_t)^2 \right)^{\frac{n-1}{2}} \\ &\geq \mathcal{L}(a_s + a_t) \left(a_s^2 + a_t^2 - f_s^2 - f_t^2 \right)^{\frac{n-1}{2}} \\ &\geq \mathcal{L}(a_s + a_t) \left((a_s^2 - f_s^2) + (a_t^2 - f_t^2) \right)^{\frac{n-1}{2}} \\ &\geq \mathcal{L} a_s (a_s^2 - f_s^2)^{\frac{n-1}{2}} + \mathcal{L} a_t (a_t^2 - f_t^2)^{\frac{n-1}{2}} \end{aligned}$$

Therefore, we have $\lambda(\mathcal{E}_{\text{IS}}) \geq \lambda(\mathcal{E}_{\text{LS}})$. □

6.4 *Experiments and Analysis*

We evaluate GuILD on an array of planning problems to characterize the proposed beacon selectors (Figure 6.4).

In the $\mathbb{R}^2$ environments, the task is to plan from the bottom left corner to the top right corner (Forest, TwoWall), or the bottom right corner (Trap). A forest of obstacles is randomly placed throughout each environment. The easier Forestenvironment only has these random obstacles; as a result, there are many different homotopy classes that will produce near-optimal paths. TwoWall and Trap introduce large obstacles with narrow passages that must be crossed to produce near-optimal paths. Discovering these passages typically requires the

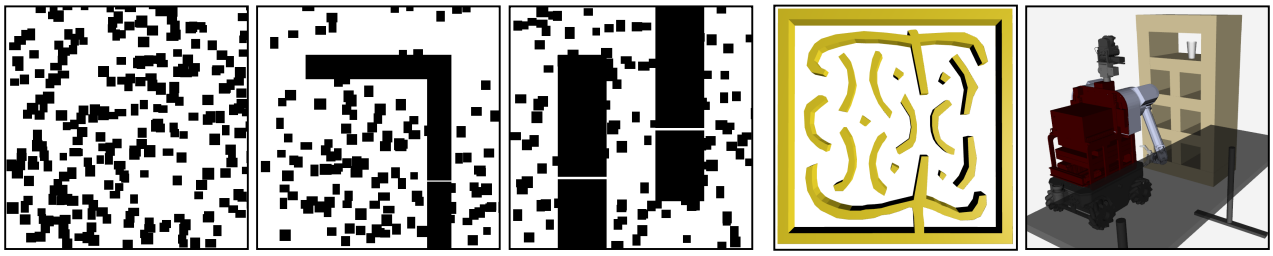


Figure 6.4: Evaluation environments.

space to be sampled very densely. At the lower sampling resolutions resulting from fewer initial graph samples, there are many suboptimal homotopy classes that are more easily sampled and discovered. Therefore, initial paths will have high cost and $\lambda(\mathcal{E}_{\text{IS}})$ will be much larger than $\lambda(\mathcal{X})$. These more challenging scenarios highlight the limitations of IS densification.

In SE2Maze, a benchmark environment from OMPL [119], the task is to navigate through a maze. In HERB Bookshelf, a 7-DOF manipulator [112] is tasked with moving its end-effector to pick an object from a shelf.

6.4.1 *Evaluation Metrics and Hypotheses*

The first metric we consider is the *Sample Efficiency* of incremental densification: how many samples must be drawn before the optimal SBMP algorithm converges to the cost of the optimal path? We determine this minimum cost $c(\xi^*)$ by running with a large timeout (1 min. for $\mathbb{R}^2$ problems, 5 min. for higher-dimensional planning problems).

H 4. GuILD *will require fewer samples to converge to the minimum solution cost than the Informed Set.*

Next, to understand the performance of the optimal SBMP algorithm over time, we then consider the *Convergence Percentage* and *Normalized Path Cost* as a function of samples. Convergence Percentage is the fraction of trials where the planner had converged to the optimal path cost, with that number of samples. Normalized Path Cost divides the cost of the current best solution by the optimal path cost $\frac{c(\xi)}{c(\xi^*)}$.

H 5. *For a fixed sample budget,* GuILD *will have a higher Convergence Percentage than the Informed Set.*

H 6. *For a fixed sample budget,* GuILD *will have lower Normalized Path Cost than the Informed Set.*

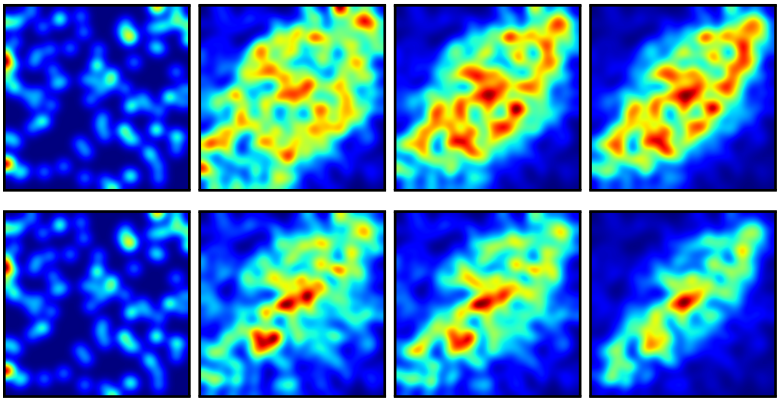


Figure 6.5: Snapshots of the sample heatmap for IS (top) and UNIFORM-GUILD (bottom) on the FOREST environment. Regions that have been sampled more densely are more red. UNIFORM-GUILD densifies around the optimal path more quickly than IS.

	IS (Baseline)	UNIFORM-GUILD	GREEDY-GUILD	BANDIT-GUILD
FOREST	(2414, 2814)	(1714, 1914)	(1414, 1614)	(1410, 1690)
TWOWALL	(2614, 3414)	(1814, 2414)	(1614, 2294)	(1830, 2310)
TRAP	(3930, 6110)	(1930, 2590)	(3410, 3990)	(2210, 2410)
SE2MAZE	(2025, 2525)	(1525, 1825)	(1525, 1725)	(1625, 1825)
HERB BOOKSHELF	(4525, 5605)	(1325, 1725)	(1225, 1605)	(1225, 1805)

Table 6.1: Median Sample Efficiency for converging to the optimal path cost, with nonparametric 95% confidence interval. The best performing BEACONSELECTOR on each planning problem is highlighted.

6.4.2 *Results*

To test these hypotheses, we implement the BEACONSELECTORS described in Section 6.3. We construct the set of beacons $\mathcal{B}$ by sampling states from a low-discrepancy sequence [51]. We run 100 random trials for each pair of algorithm and environment.

First, we render a heatmap to visualize snapshots of the sampling distribution over time on the simple FORESTenvironment (Figure 6.5). Qualitatively, UNIFORM-GUILDfocuses sampling around the eventual optimal path much more quickly than IS, which we expect to yield improved Sample Efficiency and faster convergence.

Table 6.1 reports a nonparametric 95% confidence interval on the median Sample Efficiency for each instantiation of GUILD. We find that regardless of the beacon selector, GUILD focuses sampling more efficiently than sampling from the IS, supporting **H 4**. In general, all three GUILD selectors achieve comparable sample efficiency, suggesting that the key to their success is sampling from Local Subsets. However, BANDIT-GUILD most consistently ranks among the best performing BEACONSELECTORS, while UNIFORM-GUILD and GREEDY-GUILD were each marginally less efficient on one environment.

To understand convergence across the random trials, we plot the Convergence Percentage as a function of the number of samples (Figure 6.6a-e). To support **H 5**, we would expect the IS baseline curve to remain below the GUILD curves. We find this to be the case: on

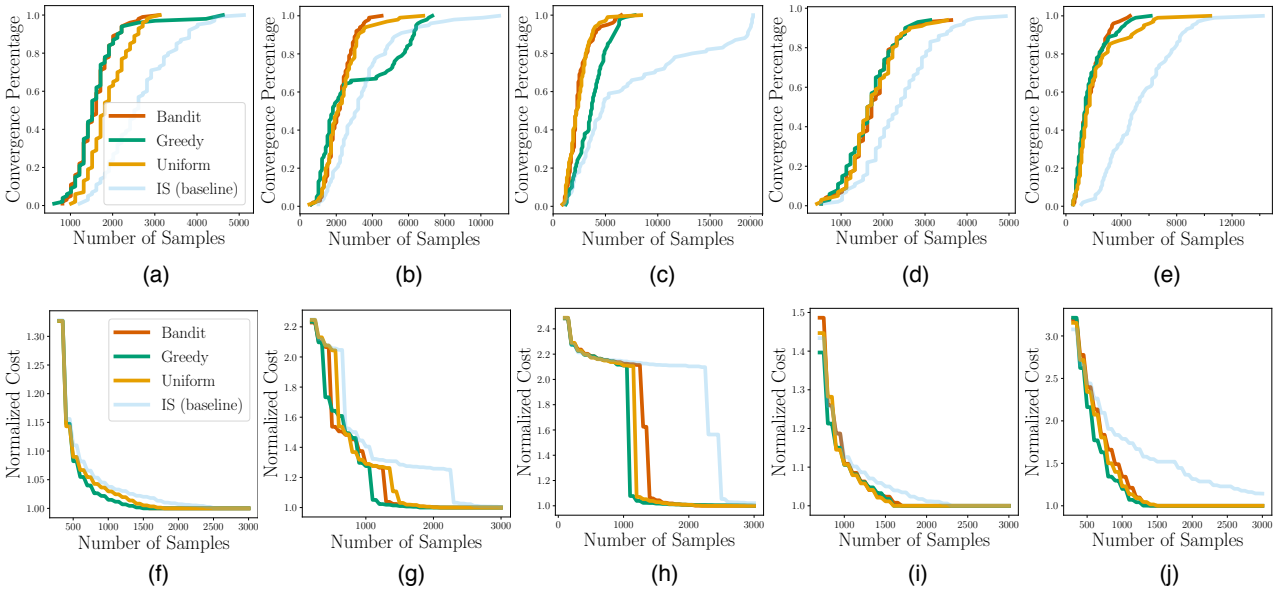


Figure 6.6: (top) Convergence Percentage across 100 trials. For most environments, sampling from the IS results in a worse Convergence Percentage than GuILD for a given sample budget. (bottom) Normalized Path Cost, median across 100 trials. Sampling with GuILD reaches a near-optimal path length more quickly than sampling from the IS, with significant improvement in all but the easiest planning domain.

most environments, sampling from the IS results in a lower Convergence Percentage for a fixed sample budget. However, on a third of the TwoWall trials, the Greedy-GuILD heuristic causes over-sampling in regions that ultimately do not yield the optimal path.

While **H 4** shows that GuILD ultimately converges faster to the optimal cost, we would also like to characterize the rate of convergence. For each planning problem, we plot the median Normalized Path Cost across the trials to understand how path length is reduced over time (Figure 6.6f-k). Sharp drops in path length (e.g., Figure 6.6h) highlight the discrepancy between high-cost homotopies that are easy to sample (navigating around large obstacles) and near-optimal homotopies that are difficult to sample (crossing narrow passages through obstacles). Steady decreases in path length (e.g., Figure 6.6f) show planning problems where sampling is not stuck in high-cost homotopies and can discover low-cost homotopies more easily.

H 6 is supported across all environments: sampling with GuILD consistently yields a lower cost than sampling with IS. In planning problems with low-cost homotopies, GuILD automatically focuses sampling to deliver paths through these narrow passages. With the Trap environment, GuILD instances identify the narrow passage around 1100-1200 samples and quickly optimize within homotopies crossing the passage. By contrast, the IS baseline requires nearly double the samples to find the passage. The TwoWall environment shows a similar trend, although with additional intermediate-cost homotopies that only traverse one of the two narrow passages (Figure 6.6h).

The SE2Maze and HERB Bookshelf demonstrate the anytime performance of GuILD on environments with less distinct homotopy costs. As a result, all sampling schemes steadily improve path cost, similar to Forest. However, in HERB Bookshelf, GuILD instances have a much steeper improvement, suggesting that GuILD may yield more significant sample efficiency improvements in higher dimensions.

6.5 Discussion

GuILD is a new framework for incremental densification with a simple insight: even when the planner fails to discover a shorter path, the search tree still contains valuable information that can immediately improve the densification strategy. GuILD uses the search tree to select a *beacon*, a vertex in the search tree that decomposes the original sampling/planning problem into two smaller subproblems. Improving the cost-to-come for any beacon allows GuILD to shrink and expand the Local Subsets that it samples from. Similar to the Informed Set that GuILD builds upon, Local Subsets can be easily (and efficiently) incorporated into any sampling-based optimal motion planning algorithm.

We demonstrate that even simple beacon selectors, such as Uniform-GuILD, can dramatically accelerate the convergence rate relative to the Informed Set densification baseline. We also propose a Bandit-GuILDselector using EXP3, an adversarial bandit algorithm that consistently ranks among the best beacon selectors across all the planning problems we considered. In particular, GuILD excels in domains with difficult-to-sample homotopy classes and high-dimensional planning problems.

In this work, we have primarily considered either heuristic beacon selectors (Greedy-GuILD) or beacon selectors that learn from experience online (Bandit-GuILD). Better heuristics for selecting beacons may exist, including ones that build on prior work in identifying and sampling bottleneck points. We believe that experience in the form of large planning problem datasets may also provide valuable information that will guide beacon selection to sample even more efficiently.

7
Conclusion

This thesis addressed the problem of efficient robot motion planning in three phases: graph construction, efficient lazy graph search, and incremental densification. We propose a collection of frameworks and algorithms to address challenges with each of these phases, to operationalize efficient robot motion planning in cluttered environments. Figure 7.1 shows how the developed algorithms can be unified into a framework for optimal motion planning. In Section 7.1, we summarize the proposed algorithms and the underlying key insights in specifically addressing the challenges introduced in Chapter 1. In Section 7.2 we outline potential avenues to build upon the results presented in this thesis. Finally, in Section 7.3, we close with concluding remarks.

7.1 Summary and Contributions

To address the problem of efficient optimal robot motion planning, this thesis begins by adopting the sampling-based motion planning paradigm of approximating the continuous robot planning with a discrete graph and searching for the shortest path. Subsequently, under this paradigm, the thesis examines three principle challenges 1: (a) *search*: what is the ideal strategy to efficiently search for the shortest path on a given graph approximation, (b) *sampling*: can the graph approximation be conditioned on the planning query to enable faster planning times, and finally (c) *optimality*: how can the shortest path on the graph be efficiently improved to obtain the optimal solution. We propose a collection of frameworks and algorithms to address each of these challenges. Figure 7.1 shows how the developed algorithms can be unified in a framework for optimal motion planning.

C1: Compute the shortest path while minimizing planning time

We opened this thesis proposing *Generalized Lazy Search* (GLS) in Chapter 3 to efficiently search for the shortest path on a graph. In contrast to

conventional lazy search algorithms, our key insight was that *interleaving* lazy search with edge evaluations balances the computational costs of search effort and collision checks to minimize overall planning time. We show that our framework captures several existing lazy search algorithms 3.1 and is flexible to deriving new efficient algorithms with the choice of EVENT and SELECTOR (Algorithms 2, 3).

With the definitions of SUBPATHEXISTENCE and FAILFAST, we show that our insight extends to the more practical setting where we have priors on edge validity that are learned from experience. We show that using priors, we can adapt the behavior of the search algorithm to minimize the expected planning time, approximating the omniscient oracle as shown in Figure 3.2. Given any graph approximation, as shown in each row of Figure 7.1, GLS (right column) provides a mechanism for efficient shortest path search.

C2: Sample sparse graphs with local densification in narrow regions

While GLS minimizes the computational cost of search, the graph itself factors into the planning time GLS can achieve. In particular, with a uniform sampler, very dense graphs may be needed to compute a solution, but are expensive to search on. This motivated us to consider the problem of generating desirable graphs that are sparse but with adequate coverage in narrow bottleneck regions in the statespace.

To this end, in Chapter 4, we proposed (to) *Leverage Experience with Graph Oracles* (LEGO) to sample such graphs. We note that environments that a robot typically operates in share structural similarity. Our framework leverages experience in such environments to learn a generative model that can sample diverse bottleneck regions in the statespace. We formally define diverse bottleneck regions and provide provable algorithms to extract states in such regions to train the generative model. The states sampled from such a generative model are combined with uniformly sampled states to construct a sparse graph with adequate coverage in narrow passages. Figure 7.1 (top row, left column) shows such a graph with LEGO samples illustrated in yellow. Any search algorithm can be employed on the graph to determine the shortest path. While we use a CVAE framework as the generative model, LEGO focuses on *identifying* diverse bottleneck regions in the statespace. This leaves for investigation other generative models that can better capture the sampling distribution [77].

C3: Efficiently compute the optimal solution from graph approximations

The graph abstraction is a discrete approximation of the continuous statespace, and therfore, the shortest path on a graph is only guaran-

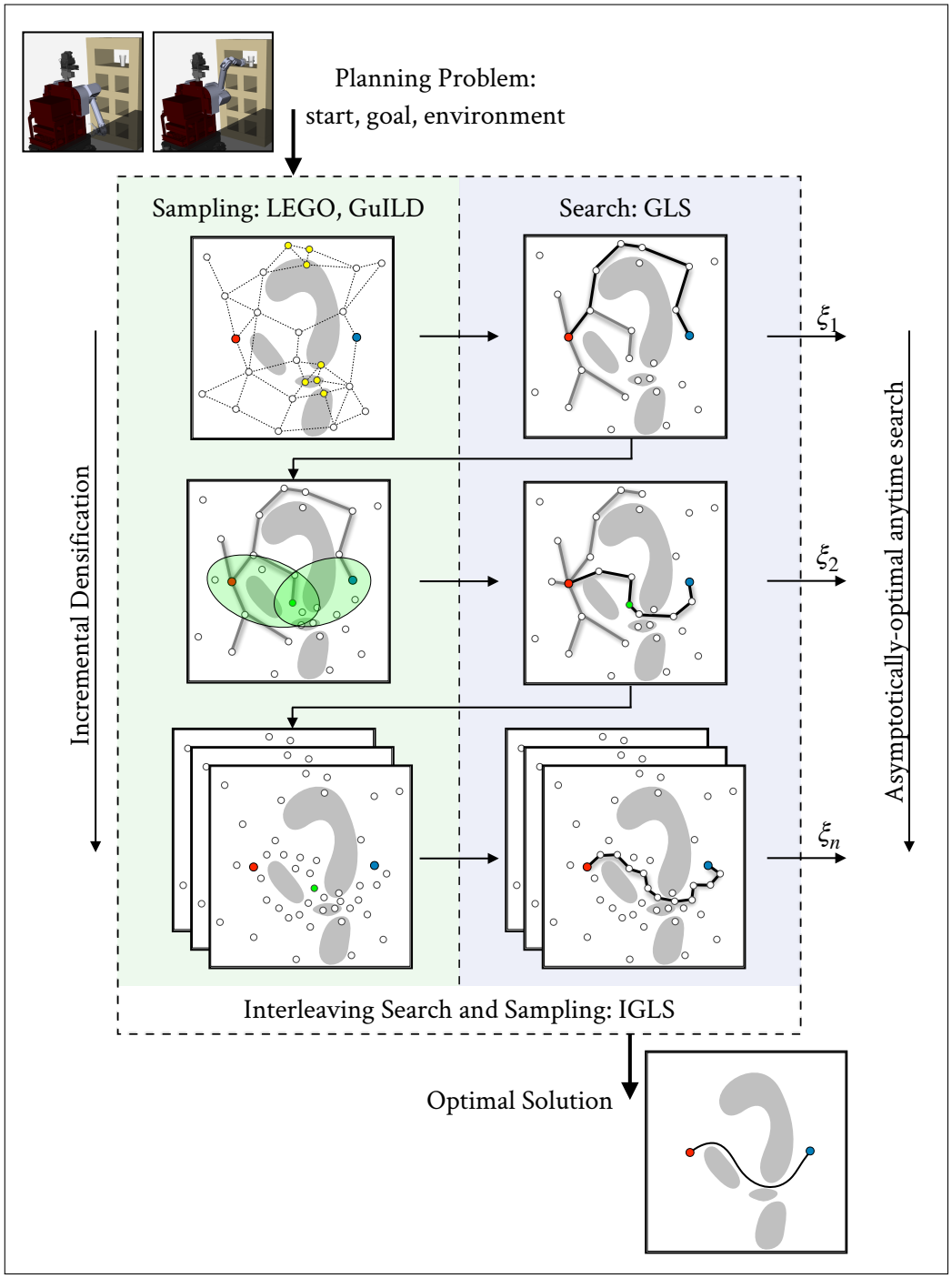


Figure 7.1: Role of the proposed algorithms in optimal motion planning

teed to be *resolution-optimal*. This leads us to finally address the problem of computing the optimal path in the continuous space. Existing optimal planning algorithms generate a series of increasingly dense graphs and employ a shortest path algorithm at each approximation. As graphs get denser, search becomes prohibitively expensive resulting in slow convergence. However, this separation of sampling and search in optimal motion planning motivated us to focus on improving the two operations independently for a faster convergence rate.

In Chapter 5 we proposed Incremental GLS (IGLS) which builds on the efficiency gains of GLS with incremental densification (Figure 7.1). We show the efficacy of IGLS in reducing the total planning time to convergence, by efficiently searching for the shortest path in each iteration of sampling. Similar to GLS, different EVENT and SELECTOR definitions lend different search behaviors to IGLS. While generalizing existing optimal sampling-based planners, IGLS is flexible in additionally leveraging prior information such as edge probabilities to reduce search time, and thereby improve the convergence rate to optimality.

Sampling-based optimal motion planners such as IGLS, although efficient in search, are typically slow in convergence as they need to sample configurations in the statespace that can help improve the initial solution. To this end, finally in Chapter 6, we proposed Guided Incremental Local Densification (GUILD) to leverage the search tree constructed by the planner in each iteration to inform sampling (Figure 7.1, middle row, left column). Our key insight is that even when incremental densification fails to compute a better solution, the improvement in the *shortest path tree* provides information to the planner to further focus subsequent sampling. At the core of our algorithm, is the observation that the search tree can provide additional information to the planner on where to sample states. While we look at generating local informed sets, this insight can extend to other similar approaches to focus sampling [68].

Figure 7.1 illustrates the role of all the proposed algorithms in the unified framework for optimal motion planning.

7.2 *Future Directions*

While this thesis takes a step towards efficient optimal robot motion planning, the fundamental problem of enabling robots to be near-optimal and yet efficient real-time planners is rich with open problems. In the remainder of this section, we discuss some of the extensions and promising avenues to pursue towards such robot capabilities.

Interleaving Lazy Search and Execution

Interleaving finite-horizon lazy search with edge evaluation naturally extends to the paradigm of interleaving planning and execution. In such a paradigm, the robot plans a subpath estimated to lead to the goal with the partial information it currently has access to, moves along the subpath while processing new sensory information to update the world and continually replan. This suggests a lazy equivalent to algorithms such as LRTA* [85] and RTAA* [80]. Completeness and correctness follow from the results presented therein. A possible extension to GLS in this real-time domain is to employ greediness[1] in edge evaluation: Given a subpath, we currently evaluate a single edge along this subpath (Algorithm 1). However, we can choose to evaluate more than one edge, hence performing an *exploitative* action. This introduces a second parameter that indicates how many edges to evaluate along the subpath. We show that our current formulation using unit greediness always outperforms any other greediness value in the number of edge evaluations. If we relax the algorithm to produce suboptimal paths, greediness may be of use in early termination allowing the robot to move sooner. We briefly study the superiority of no greediness in the Appendix A for the case that optimal paths are required.

[1] LRTA* [85] studies this parameter as *movement* denoting the number of edges the robot should move along the most promising subpath

Laziness beyond Collision Evaluation

In Chapter 3 we noted that a more dominant heuristic allows GLS reduce search effort as well as edge evaluations. While simple heuristics such as the Euclidean are easy to compute, they are often quite uninformative especially in high dimensional statespaces and complex environments. Dominant domain-specific heuristics are typically expensive to compute. In problems where multiple heuristic functions are available, e.g., in kinodynamic planning, where different relaxations of the boundary value problem can be obtained, we believe GLS can interleave search efficiently over multiple resolutions of approximation. This extends *laziness* over heuristic computation. The notion of layers of approximations essentially makes the edge weight nonbinary.[2] While GLS can resolve such a complexity with additional book-keeping, this opens the question on how to order computation across multiple levels of approximations, and across edges to potentially trade-off optimality for improved planning times.

[2] unlike in the current exposition, where an edge was either collision-free or in collision

Experience for efficient sampling strategies

Sampling distributions from local information We assume LEGO to take the entire environment description as the conditional. This is however not practical in long-horizon planning problems such as navigation in

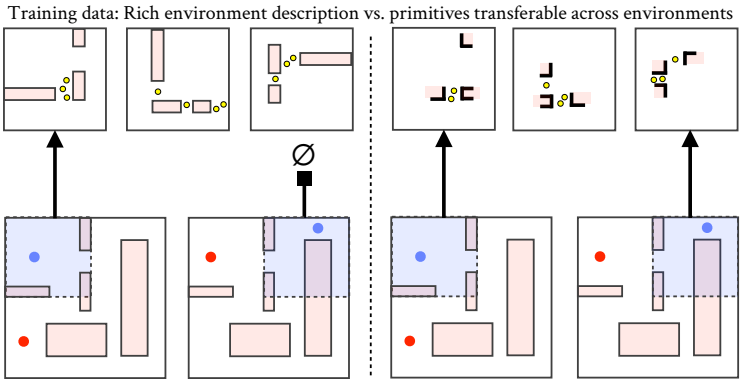


Figure 7.2: Training samplers conditioned on primitive yet transerrable descriptions (top right) as opposed to richer environment descriptions (top left) can scale to varying local structure that the robot senses as it executes motions. Blue illustrates the robot's current position (circle) and its sensing horizon. Red indicates the goal location. On the right, the sampler is trained on planar primitives extracted from the environment.

large warehouses, where the complete environment description can be expensive to process or even irrelevant. Rather than choosing samples conditioned on the entire environment, we can extend LEGO to varying workspaces and use attention or a global planner to scan the environment and guide where to sample [59, 129, 19]. As is typically the case, when the robot is constrained by a finite sensing-horizon, the algorithm can only rely on the local structure of the environment to generate samples. As the local structure can be highly arbitrary, this motivates work that identifies fundamental primitives in the workspace that the sampler can be conditioned upon [18]. As shown in Figure 7.2, training generative models on primitives such as pairs of planes [18] which are transferrable can prove to be useful in domains where the robot can only sense the local structure.

Using priors in planning is a double edged sword. While one can get astounding speed ups by focusing search on a tiny portion of C-space [62], any problem not covered in the dataset can lead to catastrophic failures: the planner may *overfit* to the prior. While one could ensure the training data covers all possible environments [122], an algorithmic solution is to explore regularization techniques for planning. LEGO's DiversePathSet can be viewed as a form of regularization for sampling. To alleviate issues with over-fitting, in addition to considering low-dimensional primitives (Figure 7.2), we can also include a more informed yet *environment-independent* conditioning vector that captures the state of the search, e.g., the length of the current shortest path. This is similar to Informed RRT* [42].

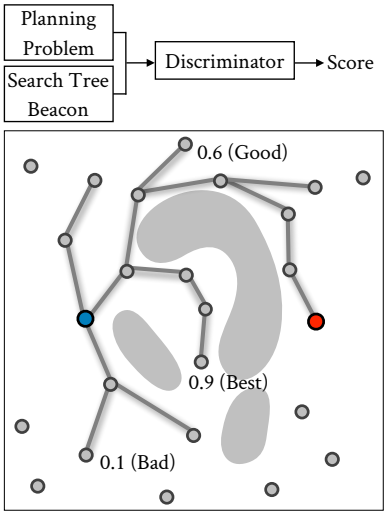


Figure 7.3: Learning to discrimate between candidate beacons

Preprocessing to improve sampling efficiency GuILD reduces the problem of determining the shortest path from a distribution of paths in the continuous space to *selection* of a discrete number of beacons and subsequent sampling. While we propose simple yet effective approaches to selecting beacons, one can train a discriminator (Figure 7.3) to determine the goodness of a beacon and approximate oracular behavior.

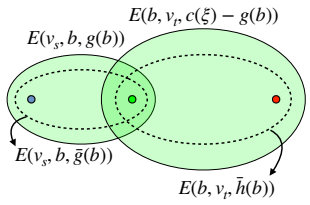


Figure 7.4: Improved upper bounds over the beacon's cost-to-come $\bar{g}(\cdot)$ and cost-to-go $\bar{h}(\cdot)$ can further focus sampling

Constraining the beacons to a fixed set of samples (Chapter 6, Section 6.3) opens avenues for further preprocessing. The measure of local informed sets depends on the cost-to-come and cost-to-go values. This suggests that improved heuristics can further focus sampling. Algorithms such as ALT [45] and REACH for A* [47] determine upper bounds to shortest paths between two vertices. Such upper bounds can further constrain the measure of local informed sets (Figure 7.4). While we look at considering two subproblems of the original planning problem by constructing the local informed sets, the idea can be extended to a heirarchical approach in long-horizon planning domains.

7.3 Concluding Remarks

This thesis began with the vision to make feasible, efficient and real-time robot motion planning. Realizing this vision required a careful consideration of *efficiency* in (a) representing the planning problem (Chapter 4) - *sampling efficiency*, (b) computing the solution within that choice of representation (Chapter 3) - *search efficiency* and finally (c) refining the representation to compute the optimal solution (Chapters 5,6) - *search and sampling efficiency*.

We close this thesis with the optimism that the insights, algorithms and frameworks proposed in this thesis contribute to extensions (Chapter 7) that continue to work towards this vision.

A
Appendix: GLS Details

In this chapter we provide a detailed pseudo-code of GLS as well as the proofs for the theorems stated in Sections 3.3 and 3.4.

Algorithm Implementation

As introduced in Section 3.3, GLS maintains a lazy shortest-path tree $\mathcal{T}_{\text{lazy}}$ over the graph $\mathcal{G}$. Every vertex v in $\mathcal{T}_{\text{lazy}}$ is associated with a node τ which encodes auxilliary search information. For the sake of simplicity, for the remainder of this exposition we will consider $\mathcal{T}_{\text{lazy}}$ to consist of nodes (as against just vertices of the graph). We define the node entry $\tau \in \mathcal{T}_{\text{lazy}}$ as $\tau = (v, p, w, c)$, where $v[\tau] = v$ is the vertex associated with τ, $p[\tau] = p$ is τ's parent in $\mathcal{T}_{\text{lazy}}$ which can be backtracked to compute a subpath ξ_τ from v_{s} to v. The node τ also stores the length of the path to the node in $\mathcal{T}_{\text{lazy}}$, $w[\tau] = w[\xi_\tau]$. Finally, c denotes a vector of τ's child nodes in $\mathcal{T}_{\text{lazy}}$. The child nodes are tracked to allow efficient repair of the search tree if necessary. For ease in exposition, we will assume that when a parent is assigned to a node, the parent is updated to track the new child. Hence for the remaining discussion, we consider the node as $\tau = (v, p, w)$. The node can additionally store information for Event and Selector such as depth of a subpath to a node in the $\mathcal{T}_{\text{lazy}}$ (for ConstantDepth) or existence prior over subpaths (for SubpathExistence).

The algorithm maintains two priority queues to efficiently process nodes, each of which is ordered according to $w(\xi[\tau])$. Specifically, we will make use of the following queues:

- $\mathcal{Q}_{\text{extend}}$ stores leaf nodes in $\mathcal{T}_{\text{lazy}}$ to extend the search tree.
- $\mathcal{Q}_{\text{rewire}}$ stores nodes that require rewiring. It is used to update the structure of $\mathcal{T}_{\text{lazy}}$ when an edge is evaluated to be in collision.

We start in Alg. 14 which details the main loop used by GLS. It takes as input the graph $\mathcal{G}$, the source and target vertices v_{s}, v_{t}, the world ϕ, Event and Selector.

Algorithm 14: GLS($\mathcal{G}$, v_s, v_t, ϕ, EVENT, SELECTOR)

1 $\mathcal{E}_{\text{eval}} \leftarrow \emptyset, \mathcal{V}_{\text{rwr}} \leftarrow \emptyset$
2 $\tau_{v_s} = (v_s, \text{NIL}, 0)$; $\mathcal{T}_{\text{lazy}}$.insert($\tau_{v_s}$)
3 **forall** $v \in \mathcal{V}$, $v \neq v_s$ **do**
4 | $\tau_v = (v, \text{NIL}, \infty)$ ▷ Initialization
5 $\mathcal{Q}_{\text{extend}}$, $\mathcal{Q}_{\text{rewire}} \leftarrow \emptyset$
6 $\mathcal{Q}_{\text{extend}}$.push($\tau_{v_s}$)
7 **repeat**
8 | **repeat**
9 | | EXTENDTREE()
10 | | **if** $\mathcal{Q}_{\text{extend}}$ *is empty* **then**
11 | | | **return** *failure*
12 | **until** EVENT($\mathcal{Q}_{\text{extend}}$.front()) is true;
13 | $\tau_{\text{top}} \leftarrow \mathcal{Q}_{\text{extend}}$.front()
14 | $e \leftarrow$ SELECTOR($\xi_{\tau_{\text{top}}}$)
15 | EVALUATEEDGE(e) ▷ Populates $\mathcal{E}_{\text{eval}}$, $\mathcal{Q}_{\text{rewire}}$
16 | REWIRETREE(EVENT, $\mathcal{T}_{\text{lazy}}$) ▷ Populates $\mathcal{V}_{\text{rwr}}$, $\mathcal{Q}_{\text{extend}}$
17 **until** $v[\tau_{\text{top}}] = v_t$ ***and*** $\forall e \in \xi_{\tau_{\text{top}}}, e \in \mathcal{E}_{\text{eval}}, \phi(e) = 1$;
18 **return** $\xi^* \leftarrow \xi_{\tau_{\text{top}}}$

Main Algorithm We start in Alg. 14 by adding a node corresponding to the source into the search tree (line 2) and initializing all other nodes (lines 3-5). We continue by initializing all the priority queues used by the algorithm and inserting the source node into $\mathcal{Q}_{\text{extend}}$. The algorithm then iteratively extends the search tree until the EVENT is triggered (line 8-12 and Alg. 15). The search is halted and the SELECTOR is invoked on the subpath to the top node in $\mathcal{Q}_{\text{extend}}$ (which triggers the EVENT). The algorithm evaluates the edge (lines 13-15 and Alg. 16). If a subpath to the target node is evaluated to be collision-free (line 17), the algorithm terminates. However if an edge is found to be in collision, the subtree following the edge in collision is repaired in line 16 (and Alg. 17) before resuming search (lines 9-12).

Extending the Lazy Search Tree The queue $\mathcal{Q}_{\text{extend}}$ contains leaf nodes in the search tree $\mathcal{T}_{\text{lazy}}$. In Alg. 15, the top node τ_u in $\mathcal{Q}_{\text{extend}}$ with minimal key is popped (line 1). For each of u's successors v in $\mathcal{G}$, if the length of the subpath to reach τ_v through τ_u is cheaper than the current length $w[\tau_v]$, the node entry for τ_v is updated using τ_u (lines 3-7) and inserted into $\mathcal{T}_{\text{lazy}}$ and $\mathcal{Q}_{\text{extend}}$. This algorithm is run in a loop until the top node in $\mathcal{Q}_{\text{extend}}$ triggers the EVENT (lines 8-12, Alg. 14).

Algorithm 15: EXTENDTREE(EVENT, $\mathcal{T}_{\text{lazy}}$)

```
1 τ_u ← Q_extend.pop()
2 forall v ∈ V s.t. (u, v) ∈ E do
3 |   if w[τ_u] + w(u, v) > w[τ_v] then
4 |   |   continue
5 |   τ_v ← (v, τ_u, w[τ_u] + w(u, v))
6 |   T_lazy.insert(τ_v)
7 |   Q_extend.push(τ_v)
8 return
```

Algorithm 16: EVALUATEEDGE(e)

```
1 if φ(e) = o then                                   ▷ Expensive
2 |   E.remove(e)
3 |   T_rewire ← T_subtree(v)
4 E_eval ← E_eval ∪ e
5 return
```

Edge Evaluation Given a subpath $\xi[\tau_{\text{top}}]$ from v_s to the top node in $\mathcal{Q}_{\text{extend}}$, the SELECTOR returns an unevaluated edge e along the subpath. In Alg. 16, we query $\phi(e)$. This is an expensive operation and contributes to the computational cost of solving the SSSP (see Section. 3.2). If the edge is in collision (line 1), the corresponding edge is removed from the graph and the subtree of the target vertex is collected. This will later be used in Alg. 17) to rewire all nodes in the subtree in a systematic manner. Note that in practice, every edge evaluation is accompanied by updating auxilliary parameters that govern the behavior of EVENT and SELECTOR (such as depth of nodes for CONSTANTDEPTH or probabilities over edges for SUBPATHEXISTENCE).

Rewiring the Lazy Search Tree In Alg. 16, when an edge (u, v) is found to be in collision, the subtree in $\mathcal{T}_{\text{lazy}}$, $\mathcal{T}_{\text{subtree}}(v)$ rooted at τ_v, is to be rewired as in Alg. 17. The subtree can be obtained by traversing along the child nodes beginning at v. Initially every node in the subtree is updated to have an infinite key. Since these nodes' entries are expected to change (which could also affect whether a node can trigger the EVENT), they are removed from $\mathcal{Q}_{\text{extend}}$ and $\mathcal{T}_{\text{lazy}}$ (lines 1-5). For each of these nodes, the best *valid* parent in $\mathcal{T}_{\text{lazy}}$ is determined (lines 6-9). Nodes belonging to $\mathcal{Q}_{\text{extend}}$ are extended in line 17 of Alg. 14 and hence are considered as invalid parents (line 7). The node is updated using the new parent's node entries and pushed into $\mathcal{Q}_{\text{rewire}}$ (line 10).

Once $\mathcal{Q}_{\text{rewire}}$ is populated with all the nodes in the subtree, the al-

Algorithm 17: REWIRETREE($\mathcal{T}_{\text{lazy}}$)

1 **forall** $\tau \in \mathcal{T}_{\text{rewire}}$ **do**
2 $\quad \mathcal{T}_{\text{lazy}}$.remove($\tau$)
3 $\quad$ **if** $\tau \in \mathcal{Q}_{\text{extend}}$ **then**
4 $\quad\quad \mathcal{Q}_{\text{extend}}$.remove($\tau$)
5 $\quad \tau = (v[\tau], \text{NIL}, \infty)$
6 $\quad \mathcal{S}_{\text{parents}} \leftarrow \{\tau' \in \mathcal{T}_{\text{lazy}}$ s.t. $(v[\tau'],\ v[\tau]) \in \mathcal{E}\}$
7 $\quad$ **forall** $\tau' \in \mathcal{S}_{\text{parents}} - \{\mathcal{T}_{\text{rewire}} \cup \mathcal{Q}_{\text{extend}} \cup \tau_{v_t}\}$ **do**
8 $\quad\quad$ **if** $w[\tau] > w[\tau'] + w(v[\tau'],\ v[\tau])$ **then**
9 $\quad\quad\quad \tau \leftarrow (v[\tau], \tau', w[\tau'] + w(v[\tau'],\ v[\tau]))$
10 $\quad \mathcal{Q}_{\text{rewire}}$.push($\tau$)
11 **while** $\mathcal{Q}_{\text{rewire}}$ *is not empty* **do**
12 $\quad \tau \leftarrow \mathcal{Q}_{\text{rewire}}$.pop()
13 $\quad$ **if** $p[\tau] = NIL$ **then**
14 $\quad\quad$ **continue**
15 $\quad \mathcal{T}_{\text{lazy}}$.insert($\tau$)
16 $\quad$ **if** τ *triggers* EVENT **then**
17 $\quad\quad$ **continue**
18 $\quad \mathcal{Q}_{\text{extend}}$.push($\tau$)
19 $\quad$ **forall** $v \in \mathcal{V}$ *s.t.* $(v[\tau],\ v) \in \mathcal{E},\ \tau_v \in \mathcal{Q}_{\text{rewire}}$ **do**
20 $\quad\quad$ **if** $w[\tau] + w(u[\tau],\ v) < w[\tau_v]$ **then**
21 $\quad\quad\quad \tau_v \leftarrow (v, \tau, w[\tau] + w(v[\tau],\ v))$
22 $\quad\quad\quad \mathcal{Q}_{\text{rewire}}$.update_node($\tau_v$)
23 $\mathcal{T}_{\text{rewire}}$.clear()
24 **return**

gorithm iteratively pops the node with the minimal key from $\mathcal{Q}_{\text{rewire}}$ (lines 11-12). If a valid parent has been determined for the node, it is inserted into $\mathcal{T}_{\text{lazy}}$ (line 15). Otherwise the node is left as initialized in line 5. Essentially, this implies that nodes can be inserted and also removed from $\mathcal{T}_{\text{lazy}}$ during rewiring. If a node has been successfully rewired and can be extended further without triggering the EVENT, it is now a potential valid best parent to its successors in $\mathcal{G}$. Lines 19-22 verify if the node is indeed a better parent for each of its successors and updates them accordingly. Since $\mathcal{T}_{\text{lazy}}$ is restructured during this operation, this algorithm is accompanied by updating auxilliary parameters that govern the behavior of EVENT and SELECTOR (such as depth of nodes for CONSTANTDEPTH or probabilities over edges for SUBPATHEXISTENCE).

Theory

Proofs for Section 3.3

Theorem 3.3.1 (Completeness)**.** *Let* EVENT *be a function that on halting ensures there is at least one unevaluated edge on the current shortest path or that the goal is reached. Let* SELECTOR *be a function that evaluates at least one unevaluated edge (if it exists).* GLS *instantiated using* EVENT *and* SELECTOR *on a finite graph $\mathcal{G}$ is complete.*

Proof. In each iteration, EXTENDTREE(EVENT) ensures there is at least one unevaluated edge on the estimated shortest path (unless the goal has been reached). The SELECTOR chooses at atleast one unevaluated edge which is then evaluated. Since there are a finite number of edges ($\mathcal{G}$ is finite), the algorithm will eventually terminate. □

Theorem 3.3.2 (Correctness)**.** *If the heuristic $h(v, v_t)$ is admissible, then* GLS *terminates with the shortest feasible path.*

Proof. Let ξ^* be the shortest feasible path with respect to $w(\cdot)$ and world ϕ. Suppose GLS terminates with a path ξ' such that $w(\xi') > w(\xi^*)$. For any vertex $v^* \in \xi^*$, its f-value $f(v^*) = w(\xi_{v_s,v^*}) + h(v^*, v_t)$, where ξ_{v_s,v^*} is the subpath from the start to vertex v^*. As the heuristic function is admissible, we have that $f(v^*) \leq w(\xi^*)$.

Recall that in each iteration, the shortest subpath computation returns a vertex v_{ret} with the smallest f-value among all the leaves of the tree $\mathcal{T}_{\text{lazy}}$. If the algorithm terminated with $v_{\text{ret}} \notin \xi^*$, it means that all other leaf vertices in $\mathcal{T}_{\text{lazy}}$ satisfied $f(v_{\text{leaf}}) > w(\xi^*)$. This contradicts the fact that there will always exist a leaf vertex along ξ^* in $\mathcal{T}_{\text{lazy}}$ [52]. Therefore, at termination we must have that $f(v) = w(\xi^*)$ and GLS returns the shortest feasible path ξ^*. □

Proposition 3.3.1 (Correctness)**.** *Consider* GLS *with any* SELECTOR *and any* EVENT*. If the heuristic $h(v, v_t)$ is consistent, and the* SELECTOR *evaluates an edge along a path $\xi_{v_s,v}$ such that all edges along $\xi_{v_s,v}$ have been evaluated to be feasible, then $\xi_{v_s,v}$ is the shortest feasible path to v, i.e., $w(\xi_{v_s,v}) = w^*(v)$.*

Proof. Let (v_0, v_1) be the last edge evaluated on the path $\xi_{v_s,v}$, where $\xi_{v_s,v} = (v_s, \ldots, v_0, v_1, \ldots v)$. Assume there exists a shorter feasible path $\xi'_{v_s,v} = (v_s, \ldots, v', \ldots v)$ such that $w(\xi') < w(\xi)$. Let v' be the vertex along ξ' that has yet to be expanded in the search tree. Let $\xi_{v_s,v'}$ be the prefix that has been expanded, and $\xi_{v',v}$ that is unexpanded. We have

the following inequality:

$$\begin{aligned} & w(\xi_{v_s,v'}) + h(v', v_t) \\ \leq & w(\xi_{v_s,v'}) + h(v', v) + h(v, v_t) \qquad \text{(consistency)} \\ \leq & w(\xi_{v_s,v'}) + w(\xi_{v',v}) + h(v, v_t) \\ \leq & w(\xi') + h(v, v_t) \\ < & w(\xi_{v_s,v}) + h(v, v_t) \end{aligned} \tag{A.1}$$

This means that v' would be expanded before v and edges along ξ' would be evaluated before (v_0, v_1) is evaluated.

□

Theorem 3.3.3 (Complexity)**.** *Let α be the maximum depth of a lazy subtree $\mathcal{T}_{\text{lazy}}$ for a given* Event*. The total running time of the algorithm is bounded by $O(nd^{\alpha} \cdot \log(n) + m)$, where n and m are the number of vertices and edges in $\mathcal{G}$, and d is the maximal degree of a vertex.*

Proof. Let $\text{Anc}(v)$ be the set of all unexpanded vertices that are α edges from v and lie on a path between v_s and v. Note that $|\text{Anc}(v)| = O(d^{\alpha})$. Furthermore, the number of edges connecting vertices in $\text{Anc}(v)$ is bounded by $O(d^{\alpha})$.

We wish to bound the number of times the lazy tree $\mathcal{T}_{\text{lazy}}$ associated with v will be updated through the algorithm's execution. We charge each update to the event that the algorithm evaluates one of the edges connecting vertices in $\text{Anc}(v)$ to v.

Each such update involves updating queues of nodes. The total number of nodes is bounded by n and the cost of updating the queue is logarithmic in its size. Finally, note that each edge is evaluated at most once.

Thus, the algorithm's running time can be bounded by

$$\underbrace{O(n)}_{\text{\# of nodes}} \cdot \underbrace{O(d^{\alpha})}_{\text{\# of node updates}} \cdot \underbrace{O(\log n)}_{\text{cost of node update}} + \underbrace{O(m)}_{\text{\# of edges}},$$

which concludes the proof. □

Theorem 3.3.4. *Consider two instantiations of* GLS *with heuristics h_1 and h_2. Both use the selector* Forward*. Both use the same* Event*. Let $\mathcal{E}_{\text{eval},1}$ and $\mathcal{E}_{\text{eval},2}$ be the edges evaluated respectively. If h_1 strictly dominates h_2, then it evaluates a subset of edges, i.e. $\mathcal{E}_{\text{eval},1} \subseteq \mathcal{E}_{\text{eval},2}$.*

Proof. Assume the statement is not true, i.e., $\mathcal{E}_{\text{eval},1} \setminus \mathcal{E}_{\text{eval},2} \neq \emptyset$. Let $(v_0, v_1) \in \mathcal{E}_{\text{eval},1} \setminus \mathcal{E}_{\text{eval},2}$ be the edge such that v_0's cost-to-come is minimal. Let u be the parent of v_0 on the shortest path from v_s to v_0. Note that $(u, v_0) \in \mathcal{E}_{\text{eval},1} \cap \mathcal{E}_{\text{eval},2}$, i.e., this edge has been evaluated to be valid by both algorithms. Let $\mathcal{T}_i$ denote the search tree of the

algorithms with heuristic $h_i,\ i \in \{1,\ 2\}$. Finally, let $w^*(v)$ denotes the length of the shortest feasible path from v_s to vertex v. Since h_1 strictly dominates h_2, we have for all v, $h_1(v, v_t) > h_2(v, v_t)$.

Since edge (v_0, v_1) was evaluated by GLS with heuristic h_1, it had to belong to a subpath that triggered EVENT. Let this subpath be ξ_τ which corresponds to a leaf vertex τ. This implies

$$w(\sigma_\tau) + h_1(\tau, v_t) \leq w^*(v_t) \tag{A.2}$$

The edge (u, v_0) was evaluated by GLS with heuristic h_2. The vertex τ exists in the priority queue after this evaluation, similar to the case with the previous algorithm. Consider the f-value of vertex τ

$$w(\sigma_\tau) + h_2(\tau, v_t) < w(\sigma_\tau) + h_1(\tau, v_t) \leq w^*(v_t) \tag{A.3}$$

Hence τ will be expanded by GLS with heuristic h_2 before v_t is expanded. This will result in the EVENT being triggered with σ_τ, and evaluation of edge (v_0, v_1). □

Note that the proof of Theorem 3.3.4 assumes that h_1 *strictly* dominates h_2. If h_1 weakly dominates h_2 (namely, if $\forall v, h_1(v) \geq h_2(v)$), additional constraints on tie-breaking between vertices are essential for the proof to hold, similar to the dominance of A* with a weakly dominant heuristic [101, 56].

Theorem 3.3.5. *Consider two instantiations of* GLS. *Let* EVENT1, EVENT2 *be the two events such that* EVENT1 *is delayed w.r.t* EVENT2. *Both use the* FORWARD SELECTOR. *Let $\mathcal{E}_{\text{eval},1}$ and $\mathcal{E}_{\text{eval},2}$ be the edges evaluated respectively. Then the delayed event evaluates a subset of edges, i.e. $\mathcal{E}_{\text{eval},1} \subseteq \mathcal{E}_{\text{eval},2}$.*

Proof. Assume that $\mathcal{E}_{\text{eval},1} \setminus \mathcal{E}_{\text{eval},2} \neq \emptyset$ and let $(v_0, v_1) \in \mathcal{E}_{\text{eval},1} \setminus \mathcal{E}_{\text{eval},2}$ be the edge such that v_0's cost-to-come is minimal. Let u be the parent of v_0 on the shortest path from v_s to v_0. Note that $(u, v_0) \in \mathcal{E}_{\text{eval},1} \cap \mathcal{E}_{\text{eval},2}$, i.e., this edge has been evaluated to be valid by both the algorithms. Let $\mathcal{T}_1, \mathcal{T}_2$ denote the search trees of the two algorithms. Finally, let $w^*(v)$ denotes the length of the shortest feasible path from v_s to vertex v.

Since edge (v_0, v_1) belonged to a subpath σ_1 that triggered EVENT1, there is a leaf vertex τ_1 in $\mathcal{T}_1$ that was expanded by the lazy search. This implies

$$w^*(u) + w(\sigma_1) + h(\tau_1, v_t) \leq w^*(v_t) \tag{A.4}$$

The edge (u, v_0) was evaluated by GLS with $\mathcal{E}_{\text{eval},2}$. Since EVENT1 is delayed with respect to EVENT2 there exists a leaf vertex τ_2 in $\mathcal{T}_2$ such that the path σ_2 from v_0 to τ_2 is a prefix of σ_1, i.e. $\sigma_2 \subseteq \sigma_1$. Let

$\sigma_{2,1}$ be the postfix. Then we have

$$\begin{aligned} & w^*(u) + w(\sigma_2) + h(\tau_2, v_t) \\ \leq & w^*(u) + w(\sigma_2) + w(\sigma_{2,1}) + h(\tau_1, v_t) \qquad \text{(consistency of } h\text{)} \\ \leq & w^*(u) + w(\sigma_1) + h(\tau_1, v_t) \\ \leq & w^*(v_t) \end{aligned} \tag{A.5}$$

This implies that τ_2 will be expanded by GLS with EVENT2 before v_t is expanded, which in turn implies (v_0, v_1) will be evaluated.

□

Theorem 3.3.6. *Let Ξ be the set of paths that are shorter than ξ^* and infeasible.*

Let EVENT1, EVENT2 *be sufficiently delayed such that they trigger only when the shortest subpath $\xi_{\text{sub}} \in \mathcal{T}_{\text{lazy}}$ is a prefix to a path $\xi \in \Xi$. Given* FORWARD, *they both evaluate optimal number of edges $\mathcal{E}^*_{\text{eval}}$. Let $\mathcal{V}_{\text{rwr},1}$ and $\mathcal{V}_{\text{rwr},2}$ be the vertex rewirings respectively.*

If EVENT1 *is delayed w.r.t* EVENT2, *the vertex rewiring of the delayed event is more, i.e. $|\mathcal{V}_{\text{rwr},1}| \geq |\mathcal{V}_{\text{rwr},2}|$.*

Proof. Consider the path ξ^i from the set Σ that needs to be eliminated. Let ξ^i_1 be the subpath prefix that triggers EVENT1. Let ξ^i_2 be the subpath prefix that triggers EVENT2. Since EVENT1 is delayed w.r.t EVENT2, one subpath encompasses the other, i.e., $\xi^i_2 \subseteq \xi^i_1$. When the selector invalidates the path, the difference in the rewiring is at least the additional portion of the path that is expanded, i.e. $\left|\mathcal{V}^i_{\text{rwr},1}\right| - \left|\mathcal{V}^i_{\text{rwr},2}\right| \geq |\xi^i_1 \setminus \xi^i_2|$. The total difference in rewiring is

$$|\mathcal{V}_{\text{rwr},1}| - |\mathcal{V}_{\text{rwr},2}| \geq \sum_{i=1}^{K} \left|\mathcal{V}^i_{\text{rwr},1}\right| - \left|\mathcal{V}^i_{\text{rwr},2}\right| \geq \sum_{i=1}^{K} \left|\xi^i_1 \setminus \xi^i_2\right| \geq 0 \tag{A.6}$$

□

Theorem 3.3.7. GLS *evaluates the same number of edges $|\mathcal{E}_{\text{eval}}|$ as* LAZYSP, *i.e., is edge optimal, while having a smaller number of vertices rewired $|\mathcal{V}_{\text{rwr}}|$ under the following setting:*

1. *Heuristic: Distance on the unevaluated graph $h_G(v, v_t)$*
2. EVENT*:* HEURISTICPROGRESS
3. SELECTOR*:* FORWARD

Proof. We are going to prove this via induction over iterations of LAZYSP and GLS. Each iteration cycles through the algorithm by invoking EVALUATEEDGES (SELECTOR), EXTENDTREE (EVENT) and SELECTSHORTESTSUBPATH ().

At iteration i, let $\mathcal{E}^i_{\text{eval,LSP}}$ and $\mathcal{V}^i_{\text{rwr,LSP}}$ be the edges evaluated and vertex rewired respectively by LazySP. Let ξ^i be the candidate shortest path.

Let $\mathcal{E}^i_{\text{eval,GLS}}$ and $\mathcal{V}^i_{\text{rwr,GLS}}$ be the edges evaluated and vertices rewired, respectively, by GLS at iteration i. Recall that distance on the graph $h_{\mathcal{G}}(v, v_t)$ is the heuristic used by the search. Let v^i be the leaf vertex corresponding to a current shortest subpath from the start ξ_{v_s,v^i}. This implies v^i corresponds to a vertex with the smallest f-value $w(\xi_{v_s,v^i}) + h_{\mathcal{G}}(v^i, v_t)$.

We also introduce the lazy edge status function $\phi_{\text{lazy}}(\xi, \mathcal{E}_{\text{eval}})$ which determines if a path ξ is valid depending on edges evaluated thus far in $\mathcal{E}_{\text{eval}}$.

Following are the conditions for the induction:

A Both algorithms have the same set of evaluated edges $\mathcal{E}^i_{\text{eval,GLS}} = \mathcal{E}^i_{\text{eval,LSP}}$.

B Both algorithms share the same subpath $\xi_{v_s,v^i} \subseteq \xi^i$.

For $i = 1$, $\mathcal{E}^1_{\text{eval,GLS}} = \mathcal{E}^1_{\text{eval,LSP}}$ because no edges have been evaluated. Hence (A) is true. Since $h_{\mathcal{G}}(v, v_t)$ is the distance on the unevaluated graph, the leaf vertex v^i considered by GLS lies on ξ^1, i.e. $\xi_{v_s,v^i} \subseteq \xi^1$. Hence (B) is true.

Assuming the conditions hold true for *ith* iteration, we will show these conditions hold for $i + 1$.

In the ith iteration, since both LazySP and GLS use Forward, share the same subpath (B) and have the same evaluation status (A) - they both evaluate the same edge e. Both algorithms increase their evaluated set $\mathcal{E}^{i+1}_{\text{eval,LSP}} = \mathcal{E}^{i+1}_{\text{eval,GLS}} \leftarrow \mathcal{E}^i_{\text{eval,GLS}} \cup e$. Hence (A) holds.

If e is valid, neither algorithms rewire vertices. However, if an edge is in collision, LazySP rewires at least the remainder of the path ξ^i. GLS does not have to rewire the remainder of the subpath ξ_{v_s,v^i} as it was never expanded beyond an edge during the search. Hence GLS can only result in smaller rewires, i.e. $|\mathcal{V}^{i+1}_{\text{rwr,GLS}}| \leq |\mathcal{V}^{i+1}_{\text{rwr,LSP}}| - |\xi_{v^i,v_t}|$.

We will now show that $\xi_{v_s,v^{i+1}} \subseteq \xi^{i+1}$.

LazySP finds the next candidate shortest path ξ^{i+1} by solving the following search problem

$$\begin{aligned} \xi^{i+1} \leftarrow \arg\min_{\xi} \; & w(\xi) \\ \text{s.t.} \; & \phi_{\text{lazy}}(\xi, \mathcal{E}^{i+1}_{\text{eval,LSP}}) = 1 \end{aligned} \tag{A.7}$$

GLS invokes the ExtendTree (Event) which proceeds till HeuristicProgress toggles off the search. The search stops at vertex v^{i+1}

which satisfies the following:

$$\begin{aligned} v^{i+1} \leftarrow \arg\min_{v} \; & w(\xi_{v_s,v}) + h_{\mathcal{G}}(v, v_t) \\ \text{s.t.} \; & h_{\mathcal{G}}(v, v_t) < h_{\min} \end{aligned} \tag{A.8}$$

Note that $\phi_{\text{lazy}}(\xi_{v_s,v}, \mathcal{E}^{i+1}_{\text{eval,GLS}}) = 1$, i.e. the subpath from the start to any vertex is valid according to the lazy estimate.

By definition, the heuristic $h_{\mathcal{G}}(v, v_t) = w(\xi_{v,v_t})$ is the weight of the shortest path on the unevaluated graph ξ_{v,v_t}. The heuristic progress threshold $h_{\min}$ is by definition the minimum heuristic value of the child vertex of any evaluated edge, i.e. $h_{\min} = \min_{(u',v') \in \mathcal{E}^{i+1}_{\text{eval,GLS}}} h_{\mathcal{G}}(v', v_t)$. For a leaf vertex v to trigger HeuristicProgress, $h_{\mathcal{G}}(v, v_t) < h_{\min} = \min_{(u',v') \in \mathcal{E}^{i+1}_{\text{eval,GLS}}} h_{\mathcal{G}}(v', v_t)$. As the heuristic is consistent, this implies that none of the edges $(u', v') \in \xi_{v,v_t}$ belong to $\mathcal{E}^{i+1}_{\text{eval,GLS}}$ i.e. they have not been evaluated yet. This means that the subpath to goal from such a leaf vertex is valid according to the lazy estimate, i.e. $\phi_{\text{lazy}}(\xi_{v,v_t}, \mathcal{E}^{i+1}_{\text{eval,GLS}}) = 1$.

Hence (A.8) can be re-written as

$$\begin{aligned} v^{i+1} \leftarrow \arg\min_{v} \; & w(\xi_{v_s,v}) + w(\xi_{v,v_t}) \\ \text{s.t.} \; & \phi_{\text{lazy}}(\xi_{v_s,v}, \mathcal{E}^{i+1}_{\text{eval,GLS}}) = 1 \\ & \phi_{\text{lazy}}(\xi_{v,v_t}, \mathcal{E}^{i+1}_{\text{eval,GLS}}) = 1 \end{aligned} \tag{A.9}$$

Since $\mathcal{E}^{i+1}_{\text{eval,GLS}} = \mathcal{E}^{i+1}_{\text{eval,LSP}}$, (A.7) and (A.9) are the same optimization. Hence $\xi_{v_s,v^{i+1}} \subseteq \xi^{i+1}$ and (B) holds. As a result, the induction holds.

This process continues till both algorithms discover the shortest feasible path ξ^* at the end of iteration N. Both evaluate the same number of edges $\mathcal{E}^{N+1}_{\text{eval,LSP}} = \mathcal{E}^{N+1}_{\text{eval,GLS}}$. But GLS saves on more vertices being rewired than LazySP, i.e. $|\mathcal{V}^N_{\text{rwr,GLS}}| \leq |\mathcal{V}^N_{\text{rwr,LSP}}| - \sum_{i=1}^{N} |\xi_{v^i,v_t}|$. □

Corollary 3.3.3. *There exists a graph $\mathcal{G}$ for which the number of vertex rewires $|\mathcal{V}_{\text{rwr}}|$ for* LazySP *over* GLS *is linear over logarithmic.*

Consider LazySP with Forward and GLS (HeuristicProgress, Forward). We construct the graph explicitly.

Scenario. Consider the graph in Fig. A.1. It has a set of l vertices connected to the start. The upper half of the l vertices are connected to vertex A. The lower half is connected to B. Each of A and B is connected to a chain of N vertices going to the goal. $|\mathcal{V}| = N + l + 3$, $|\mathcal{E}| = 3N + 2l - 1$.

The graph is such that *only one* of l edges connected to the start is valid. The remaining edges, not connected to start, are all valid.

The weights of the graph are such that the shortest path alternates between the top and bottom halves of the graph. One such set of

Proof.

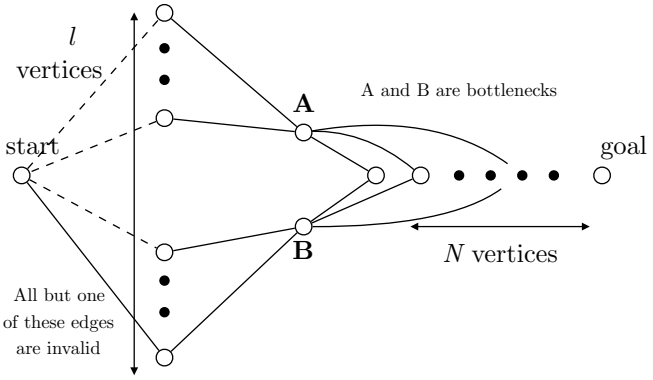


Figure A.1: An example graph structure where LazySP terminates with a provably higher computation cost than GLS.

weights is illustrated in the figure. Assume that all $l-1$ shortest paths are invalid and the last one is valid. Finally assume that A and B alternate being the optimal parent to the N vertices.

LazySP *computation* .

For LazySP, the only computation is vertex rewiring. The graph is such that successive shortest paths alternate between the upper and lower halves. The shortest paths in the upper half pass through A and lower half pass through B. Hence every edge that is invalidated, causes all N vertices to rewire to either vertex A or B. This is the optimistic thrashing scenario explained in LazyPRM*[54]. Since l edges have to be invalidated, the number of rewires is $O(Nl)$

GLS *computation* There are two computation steps to account for - heuristic computation and vertex rewiring.

The heuristic computation is a Djikstra operation.

$$\begin{aligned} &O(|\mathcal{V}|\log|\mathcal{V}|+|\mathcal{E}|) \\ &O((N+l)\log(N+l)+3N+2l-1) \\ &O((N+l)\log(N+l)) \end{aligned} \tag{A.10}$$

With HeuristicProgress the search never proceeds beyond the first set of edges, therefore the amount of vertex rewiring is 0.

Hence the complexity of LazySP is $O(Nl)$ while GLS is $O((N+l)\log(N+l))$. The ratio is linear over logarithmic growth.

□

Proofs for Section 3.4

Theorem 3.4.1. *Let* Alg *be a shortest path algorithm such that* $\textsc{Alg}(\mathcal{G}, v_s, v_t, \phi)$ *returns shortest path* ξ^**, edges evaluated* $\mathcal{E}_{\text{eval}}$ *and vertices rewired* $\mathcal{V}_{\text{rwr}}$*. For*

all such ALG*, the number of edges evaluated* $|\mathcal{E}_{\text{eval}}|$ *is lower bounded by*

$$\begin{aligned} \min_{\mathcal{E}_{\text{eval}}} \quad & |\mathcal{E}_{\text{eval}}| \\ s.t. \quad & \forall e \in \mathcal{E}_{\text{eval}},\ \phi(e) = 0 \\ & \forall \xi_i \in \{\xi \mid w(\xi) \leq w(\xi^*)\},\ \xi_i \cap \mathcal{E}_{\text{eval}} \neq \emptyset \end{aligned} \tag{3.4}$$

Proof. Let Ξ be the set of paths that are shorter than ξ^* (and infeasible), i.e. $\Xi = \{\xi_i \mid w(\xi_i) \leq w(\xi^*)\}$. Let $\mathcal{E}_{\text{inv}} = \{e \in \mathcal{E} \mid \phi(e) = 0\}$ be the set of invalid edges. Each edge $e \in \mathcal{E}_{\text{inv}}$ *covers* a path $\xi_i \in \Xi$ if $e \in \xi_i$. We define *cover* as a subset of edges $\mathcal{E}_{\text{eval}} \subseteq \mathcal{E}_{\text{inv}}$ that covers all paths in Ξ, i.e. $\xi_i \cap \mathcal{E}_{\text{eval}} \neq \emptyset$. If we select a min cover, i.e. $\min |\mathcal{E}_{\text{eval}}|$ then all paths shorter than ξ^* but infeasible will be eliminated. Hence this is the minimal set of edges that need to be eliminated by ALG. □

Theorem 3.4.2. *Given* SUBPATHEXISTENCE *(*δ*), any* SELECTOR *and distance on the unevaluated graph* $h_G(v, v_t)$ *as heuristic, the expected planning time of* GLS *can be bounded above by:*

$$K\left(c_e \frac{1}{(1-\delta)} + c_r b L(\delta)\right) \tag{3.7}$$

where K is the number of paths shorter than ξ^* *but infeasible, b is the maximum branching factor, and* $L(\delta)$ *is the maximum length of an unevaluated subpath before the event* SUBPATHEXISTENCE *(*δ*) is triggered.*

Proof. We first describe the GLS algorithm with SUBPATHEXISTENCE (δ). The algorithm searches till the probability of the current shortest subpath drops below δ. It toggles to edge evaluation which will either eliminate the subpath or check an edge such that the probability rises above δ. The search continues forward. This repeats till the shortest path has been found.

We begin by bounding the number of edge evaluations from above. Let ξ^* be the shortest feasible path. Recall that there are K paths shorter than ξ^* that are infeasible and that the algorithm has to eliminate. Since we are showing an upper bound, we can relax the condition that the paths have overlapping edges since they will only reduce edge evaluations (eliminating one implies the other is eliminated).

Consider one of K paths that we have to eliminate. If we pick an edge from the subpath, with probability $1 - \delta$ the path may be found to be invalid. A selector either invalidates a subpath with probability $1 - \delta$ or results in a wasted edge evaluation. This process is repeated till a path is eventually eliminated. The expected number of edge

evaluated to eliminate the path is:

$$\begin{aligned}\mathbb{E}_{\mathbf{p}}\left[\mathcal{E}_{\text{eval}}\right] &\leq (1-\delta) + 2\delta(1-\delta) + 3\delta^2(1-\delta) + \dots \\ &\leq (1-\delta)\left(1 + 2\delta + 3\delta^2 + \dots\right) \\ &\leq (1-\delta)\frac{1}{(1-\delta)^2} \\ &\leq \frac{1}{(1-\delta)}\end{aligned} \tag{A.11}$$

Hence the total expected cost of edge evaluation is bounded by $c_e K \frac{1}{(1-\delta)}$. Note as $\delta \to 1$, this term goes to ∞. This is backed by the intuition that triggering the event often results in increased edge evaluation.

We will now upper bound the number of vertex rewiring. Recall that the search is using as heuristic the distance on the unevaluated graph $h_{\mathcal{G}}(v, v_t)$. Hence when one of the K subpaths is eliminated, only the vertices of that subpath, and their immediate neighbors (generated when vertices are expanded) are rewired. Since we are deriving an upper bound, we will ignore overlap between subpaths (which can only help).

Recall that $L(\delta)$ is the length of the maximum unevaluated subpath before an event is triggered. Therefore, when a subpath is eliminated, the maximum vertex rewires that can occur is $\mathcal{V}_{\text{rwr}} = bL(\delta)$ where b is the maximum branching factor. A selector either invalidates a subpath with probability $1-\delta$ and results in rewiring or the process continues without any penalty. The expected number of vertices rewired before the path is eliminated can be upper bounded:

$$\begin{aligned}\mathbb{E}_{\mathbf{p}}\left[\mathcal{V}_{\text{rwr}}\right] &\leq (1-\delta)bL(\delta) + \delta(1-\delta)bL(\delta) + \dots \\ &\leq (1-\delta)bL(\delta)\left(1 + \delta + \delta^2 + \dots\right) \\ &\leq (1-\delta)bL(\delta)\frac{1}{(1-\delta)} \\ &\leq bL(\delta)\end{aligned} \tag{A.12}$$

Hence the total expected cost of vertex rewiring is upper bounded by $c_r K b L(\delta)$. Note as $\delta \to 0$, this term goes to ∞. This is backed by the intuition that triggering the event less often results in increased vertex rewiring. □

Corollary 3.4.1. *Given* SUBPATHEXISTENCE *(δ), any* SELECTOR, *a Bernoulli prior* $\mathbf{p}(e)$ *and distance on the unevaluated graph* $h_{\mathcal{G}}(v, v_t)$ *as heuristic, the expected planning time of* GLS *can be bounded above by:*

$$K\left(c_e \frac{1}{(1-\delta)} + c_r \frac{b\log(\delta)}{\log(p_{max})}\right) \tag{3.8}$$

where K is the number of paths shorter than ξ^* but infeasible, b is the maximum branching factor, and $p_{max} = \max_e \mathbf{p}(e)$ is the maximum value of an edge prior.

Proof. Consider one of K paths that we have to eliminate. Let $p_{\max}$ be the maximum probability of an edge being valid. Then the maximum length of any subpath $L(\delta)$ is

$$\begin{aligned} p_{\max}^{L(\delta)} &\geq \delta \\ L(\delta) &\leq \frac{\log(\delta)}{\log(p_{\max})} \end{aligned} \tag{A.13}$$

Using (A.13) in Theorem 3.4.2 we have

$$\begin{aligned} K\left(c_e \frac{1}{(1-\delta)} + c_r b L(\delta)\right) \\ K\left(c_e \frac{1}{(1-\delta)} + c_r \frac{b\log(\delta)}{\log(p_{\max})}\right) \end{aligned} \tag{A.14}$$

□

Theorem 3.4.3. *Given a Bernoulli prior $\mathbf{p}(e)$, there exists a critical threshold $\delta \in (0,1)$ that minimizes the upper bound on the expected computational cost*

$$\delta \approx \begin{cases} 1, & c_e \ll c_r \\ \frac{1}{\left(\frac{c_e}{bc_r}\log(\frac{1}{p_{max}})+2\right)} & c_e \gg c_r \end{cases} \tag{3.9}$$

Proof. The total expected planning time can be bounded as:

$$\begin{aligned} \mathbb{E}_{\mathbf{p}}\left[C(\mathcal{E}_{\text{eval}}, \mathcal{V}_{\text{rwr}})\right] &= c_e \left|\mathcal{E}_{\text{eval}}\right| + c_r \left|\mathcal{V}_{\text{rwr}}\right| \\ &= c_e K \frac{1}{(1-\delta)} + c_r K \frac{b\log(\delta)}{\log(p_{\max})} \\ &= K\left(c_e \frac{1}{(1-\delta)} + c_r \frac{b\log(\delta)}{\log(p_{\max})}\right) \end{aligned} \tag{A.15}$$

We will now show that there exists a critical point δ that minimizes this. Solving for that critical point, we have:

$$\begin{aligned} \frac{\partial}{\partial\delta}\left(K\left(c_e \frac{1}{(1-\delta)} + c_r \frac{b\log(\delta)}{\log(p_{\max})}\right)\right) &= 0 \\ \frac{c_e}{(1-\delta)^2} + \frac{c_r}{\log(p_{\max})}\frac{b}{\delta} &= 0 \\ (1-\delta)^2 - \frac{c_e}{bc_r}\log\left(\frac{1}{p_{\max}}\right)\delta &= 0 \\ \delta^2 - \left(\frac{c_e}{bc_r}\log(\frac{1}{p_{\max}}) + 2\right)\delta + 1 &= 0 \end{aligned} \tag{A.16}$$

Let $\eta = \left(\frac{c_e}{bc_r}\log(\frac{1}{p_{\max}}) + 2\right)$. The critical point is:

$$\delta = \frac{\eta - \sqrt{\eta^2 - 4}}{2} \tag{A.17}$$

When $c_e \ll c_r$, η is close to 2 and $\delta \to 1$.
When $c_e \gg c_r$, we have $\eta \gg 2$. Hence, the critical point is:

$$\begin{aligned}
\delta &= \frac{\eta - \sqrt{\eta^2 - 4}}{2} \\
&= \frac{\eta\left(1 - \sqrt{1 - \frac{4}{\eta^2}}\right)}{2} \\
&\approx \frac{\eta\left(1 - \left(1 - \frac{4}{2\eta^2}\right)\right)}{2} \quad \text{(First order truncation of Binomial Series)} \\
&= \frac{1}{\eta} \\
&= \frac{1}{\left(\frac{c_e}{bc_r}\log(\frac{1}{p_{\max}}) + 2\right)}
\end{aligned} \tag{A.18}$$

The critical point is inversely proportional to the ratio $\frac{c_e}{c_r}$.

□

Theorem 3.4.4. *Given a path ξ,* FAILFAST *is within a factor of 4 of the optimal expected number of edges from ξ that must be evaluated to invalidate ξ.*

Proof. We map this problem to that of Bayesian search where the objective is to sequentially search for an item (invalid edge) in a set of boxes (unevaluated edges) while minimizing the cost of search. At every iteration, an agent pays a cost of 1 and selects the edge with the most likely posterior probability of failing $P(\phi(e) = 0|\text{history})$. If the edge is invalid, the game stops, else the agent continues. We adopt Theorem 2.2 from [36] that states the bound of the expected cost accrued by this greedy agent is within a factor of 4 from the optimal strategy. □

Theorem 3.4.5. *Given a Bernoulli prior $\mathbf{p}(e)$ and a path ξ,* FAILFAST *minimizes the expected number of edges from ξ that must be evaluated to invalidate ξ.*

Proof. Given a path ξ, and a sequence of edges $S = \{e_1, e_2, \ldots, e_n\}$ belonging to the path, and the corresponding priors of the edges being valid $(p_1, p_2, \ldots, p_n)$, let the expected number of edge evaluations to invalidate the ξ be $\mathcal{E}_{\text{eval}}(S)$ which is given by

$$\begin{aligned}
\mathbb{E}_{\mathbf{p}}\left[\mathcal{E}_{\text{eval}}(S)\right] &= (1 - p_1) + 2p_1(1 - p_2) + \ldots \\
&= \sum_{l=1}^{n} \left(\prod_{m=1}^{l-1} p_m\right)(1 - p_l)\, l
\end{aligned} \tag{A.19}$$

Without loss of generality, let $p_i > p_{i+1}$ for a given i. Consider the alternate sequence of evaluations $S' = \{e_1, e_2, \ldots, e_{i+1}, e_i \ldots, e_n\}$ where

the positions of the edges e_i, e_{i+1} are swapped. Consider the difference:

$$\begin{aligned}
&\mathbb{E}_{\mathbf{p}}\left[\mathcal{E}_{\text{eval}}(S)\right] - \mathbb{E}_{\mathbf{p}}\left[\mathcal{E}_{\text{eval}}(S')\right] \\
&= \ldots + \prod_{m=1}^{i-1} p_m \left[(1-p_i)i + p_i(1-p_{i+1})(i+1)\right] + \ldots \\
&- \ldots + \prod_{m=1}^{i-1} p_m \left[(1-p_{i+1})i + p_{i+1}(1-p_i)(i+1)\right] + \ldots \\
&= \prod_{m=1}^{i-1} p_m \left[-i(p_i - p_{i+1}) + (i+1)(p_i - p_{i+1})\right] \\
&= \prod_{m=1}^{i-1} p_m (p_i - p_{i+1}) \\
&> 0
\end{aligned} \tag{A.20}$$

Since each such swap results in monotonic decrease in the objective, there exists an unique fixed point, i.e., the optimal sequence S^* has $p_1 \leq p_2 \leq \ldots \leq p_n$. □

Greediness in Edge Evaluation

In this section we study the role of greediness in edge evaluation as discussed in Chapter 7 and provide accompanying theorems and proofs. We start by noting that correctness (Theorem3.3.2) and optimality can be easily extended to the case where greediness is employed. For the following analysis, we focus on ConstantDepth Event. The results can be extended to more general Events. We fix one parameter, either the Event depth α or greediness β and show under what conditions of the other parameter (β or α, respectively), GLS performs fewer edge evaluations. Theorem A.0.1 gives a general relationship between different Event depths α for a fixed greediness β. We then move on to fix α and see how varying β affects the algorithm. In Theorem A.0.2 we show that for a fixed Event depth, no greediness ($\beta = 1$) is always better (in terms of edge evaluations) when compared to any other value of β. Finally, in Theorem A.0.3 we show the somewhat counter-intuitive result that for larger greediness values ($\beta > 1$) and a fixed Event depth, there is always an example where the greater the greediness, the better.

We start by noting that if $\beta > 1$ it may be the case that the larger the depth, the better (when considering edge evaluations). In a nutshell, the greediness β may drive the algorithm to evaluate edges along paths that, at first glance, seem promising but as the algorithm evaluates edges, it becomes evident that other paths are more promising. See Fig. A.2 for an example. In the next Theorem, we show under what conditions (for $\beta > 1$) this natural behaviour does indeed hold.

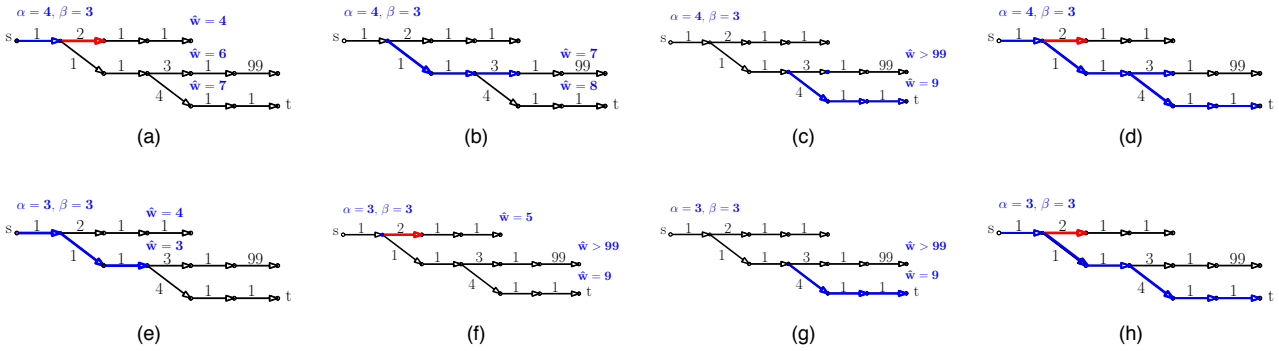


Figure A.2: Example where given a larger depth results in *more* edge evaluations. Top (a-d) and bottom (e-h) depict the flow of GLS for depth value of $\alpha = 4$ and $\alpha = 3$, respectfully and a greediness value of $\beta = 3$. Each figure depicts one iteration, blue edges are found to be collision free while red edges are found to be in collision. Value of frontier nodes are shown at each iteration and all the edges evaluated by each algorithm are shown in (d and h).

Theorem A.0.1. *Consider two instantiations of GLS with* ConstantDepth Event *and* Forward *with greediness β. Let α_1, α_2 be the depths at which the two algorithms trigger the* Event *respectively such that $\alpha_1 > \alpha_2 \geq \beta$. Let $\mathcal{E}_{\text{eval},1}$ and $\mathcal{E}_{\text{eval},2}$ be the edges evaluated respectively. Then $\mathcal{E}_{\text{eval},1} \subseteq \mathcal{E}_{\text{eval},2}$ if $\alpha_1 \geq \alpha_2 + \beta - 1$.*

Proof. Assume that $\mathcal{E}_{\text{eval},1} \setminus \mathcal{E}_{\text{eval},2} \neq \emptyset$ and let $(v_0, v) \in \mathcal{E}_{\text{eval},1} \setminus \mathcal{E}_{\text{eval},2}$ be an edge such that v_0's cost-to-come is minimal. Note that this implies that both algorithms compute the shortest path to v_0. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the respective search trees.

Consider the iteration before GLS with laziness α_1 evaluates (v_0, v) and let $(v_{\beta-1}, v_{\beta-2}, \ldots, v_1, v_0, v)$ be the sequence of vertices along the β edges lying on the shortest path from v_{start} to v. Since the edge (v_0, v) is evaluated, there exists a leaf vertex $v_i \in \mathcal{T}_1$ where $0 \leq i \leq \beta - 1$. Furthermore, the lazy path from v_i was considered, hence there is a vertex $\tilde{v} \in \mathcal{T}_1$ which is α_1 edges from v_i whose key is minimal. Namely $w(\xi_{v_s,\tilde{v}}) < w^*(v_t)$ the minimal cost to reach v_t. Note that this path $\xi_{v_s,\tilde{v}}$ contains the edge (v_0, v).

Clearly, $\mathcal{T}_2$ contains a leaf vertex associated with v_0. GLS with laziness α_2 does not expand any path from v_0 that contains the edge (v_0, v), thus all paths α_2 edges away from v_0 passing through v have lazy cost larger than $w^*(v_t)$. However, the vertex $\tilde{v}$ (which caused GLS with laziness α_1 to evaluate (v_0, v)) is $\alpha_1 - i \geq \alpha_1 - (\beta - 1) \geq \alpha_2$ edges from v_0. We know that $w(\xi_{\tilde{v}}) < w^*(v_t)$ thus (v_0, v) should have been evaluated by GLS with laziness α_2 which gives us a contradiction. For a visualization, see Figure A.3(left). □

We now move to the case where α is fixed and we compare the edge evaluation of GLS for different values of β, starting with $\beta = 1$.

Theorem A.0.2. *For every graph $\mathcal{G}$ and every $\alpha \geq \beta > 1$, we have that $E_1 \subseteq E_\beta$. Here E_x denotes the edges evaluated by* GLS *with* ConstantDepth *α and greediness x.*

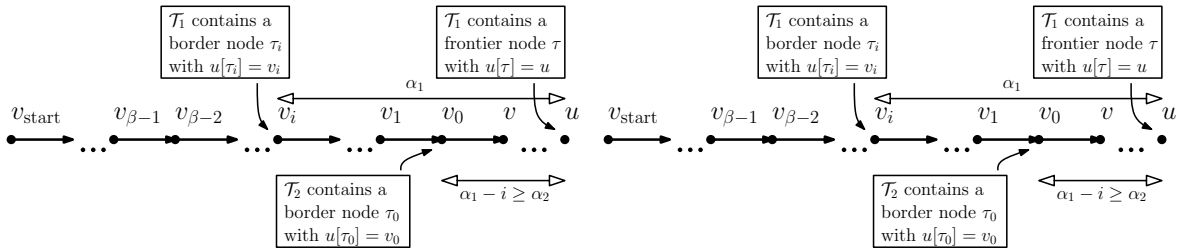


Figure A.3: Graph construction used in Theorem A.0.1 (left) and Theorem A.0.2 (right).

Proof. Assume that $E_1 \setminus E_\beta \neq \emptyset$ and let $(v_0, v) \in E_1 \setminus E_\beta$ be an edge such that v_0's cost-to-come is minimal. Note that this implies both algorithms compute the shortest path to v_0. Furthermore, let $\mathcal{T}_x$ denote the search tree of GLS with greediness x.

Consider the iteration before GLS with greediness 1 (no greediness) evaluates (v_0, v). Since the edge (v_0, v) is evaluated, the node $\tau_0 \in \mathcal{T}_1$ associated with v_0 was the border node and there exists a node $\tau \in \mathcal{T}_1$ which is α edges from τ_0 whose key is minimal. Namely, $\bar{w}(P[\tau]) < w^*$ with $w^* = w^*(v_t)$ the minimal cost to reach v_t. Note that the path $P[\tau]$ contains the edge (v_0, v).

Now, consider the search tree $\mathcal{T}_\beta$ of GLS with greediness β. Clearly, $\mathcal{T}_\beta$ contains a border node τ_0' with $u[\tau_0'] = v_0$. There exists a node τ' with $u[\tau'] = u[\tau]$. Namely, the node τ' which is exactly α edges away from τ_0' has $\bar{w}(P[\tau']) < w^*$ where $P[\tau']$ contains the edge (v_0, v). Hence the node τ' would be popped from $\mathcal{Q}_{\text{frontier}}$ before any node associated with v_t implying the edge (v_0, v) would be evaluated giving us a contradiction. For a visualization, see Figure A.3(right). □

We continue to examine the general case where α is fixed and we compare the edge evaluation of GLS for different values of β for $\beta > 1$.

Theorem A.0.3. *For every depth $\alpha < \infty$ and every greediness $\alpha \geq \beta_2 > \beta_1 > 1$, there exists a graph $\mathcal{G}$ where $E_{\beta_1} \setminus E_{\beta_2} \neq \emptyset$. Here, E_β denotes the set of edges evaluated by* GLS *with* ConstantDepth *α and greediness β.*

Proof. We construct the graph $\mathcal{G}$ explicitly. See Figures A.4 and A.5. We consider two following two cases (i) $\beta_2 \text{mod} \beta_1 \neq 0$ and (ii) $\beta_2 \text{mod} \beta_1 = 0$. For each case we provide a different graph $\mathcal{G}$ and show that $E_{\beta_1} \setminus E_{\beta_2} \neq \emptyset$. See Fig. A.4 and A.5 for depictions of each case described.

For case (i) where $\beta_2 \text{mod} \beta_1 \neq 0$, we have a path of length β_2 followed by two paths of length α. For GLS with greediness β_1, (Figure A.4 a-d), the algorithm starts by evaluating edges along the path of length β_2 (Figure A.4a). Since $\beta_2 \text{mod} \beta_1 \neq 0$, at some point it will evaluate the first edge along the upper path (which is in collision) (Figure A.4b). This path is longer than the lower one, but to see this, the

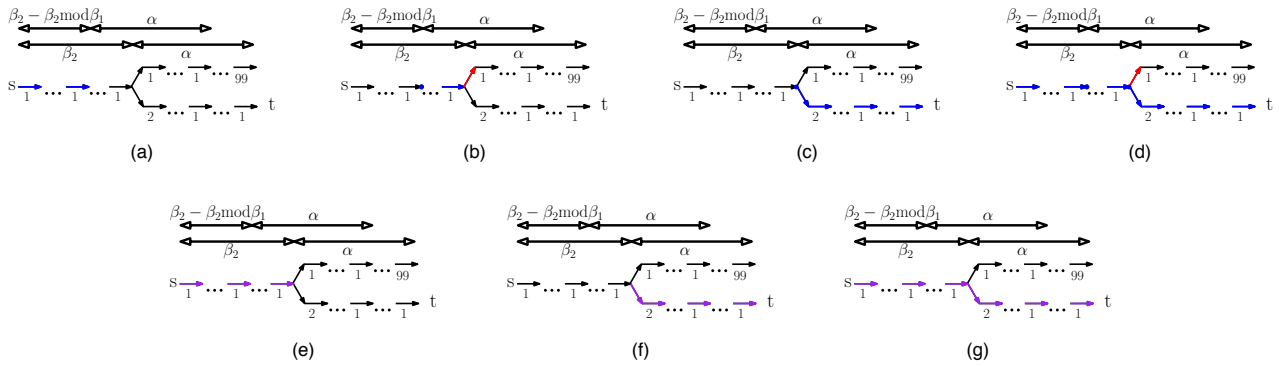


Figure A.4: Construction used in Theorem A.0.3, when $\beta_2 \text{mod} \beta_1 \neq 0$.

algorithm requires a lookahead of α edges from the end of the first path. The algorithm continues to evaluate edges along the lower path until the target is reached (Figure A.4c).

For GLS with greediness β_2 (Figures A.4 e-g), the algorithm starts by evaluating all edges along the path of length β_2 (Figure A.4e). Since it can see all edges along the upper path (which is collision free) it continues to evaluate the lower path until the target is reached (Figure A.4f). The final edges evaluated by each algorithm are depicted in Figures A.4 c,g).

For case (ii) where $\beta_2 \text{mod} \beta_1 = 0$, we have a path of length β_2 which after one edge has a shorter path of α edges. The rest of the construction is similar to case (i). Essentially, GLS with greediness β_1 and GLS with greediness β_2 behave similarly to case (i) except that both algorithms will evaluate the first edge along the path of length β_2 followed by the first (in-collision edge) of the shorter path of α edges. After this first iteration for both algorithms (Figures A.5a,e) the behaviour reduces to that of case (i). □

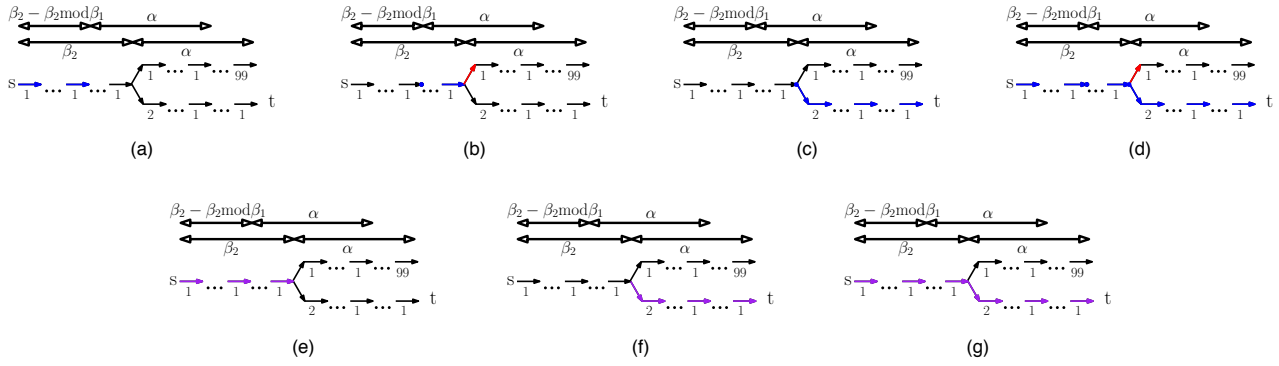


Figure A.5: Construction used in Theorem A.0.3, when $\beta_2 \text{mod} \beta_1 = 0$.

B
Appendix: LEGO Details

In this chapter we give a brief overview of the CVAE architecture, and the parameters of choice in the architecture for the training procedure in our experiments. Finally, we also describe in detail our experimental setup, the training examples chosen in each problem domain and some additional experimental results that investigate algorithm performance across sampler parameter choices.

CVAE Framework

We refer the reader to [35] for technical details and a comprehensive tutorial on CVAE. In Section B we describe the CVAE architecture implemented to train LEGO and ShortestPath algorithms. In Sections B and B, we study two parameters that determine the performance of the CVAE generative model.

Architecture

The entire CVAE module (Figure B.1) takes as input the training samples X, which in case of LEGO are the samples in bottleneck regions and along diverse paths. Additionally, the CVAE takes as input a vector of external features y, upon which the generative model is also conditioned upon. In the problems we consider, these features include information regarding the environment such as the poses of the obstacles and the start-goal pair. A standard CVAE model consists of an encoder and a decoder, often represented by neural networks trained using the input samples and the external features.

During training, the encoder network takes as input the high dimensional vector of features including the training sample and the other external features and encodes it into a low-dimensional latent variable vector. The latent variable is then fed into the decoder network along with the vector of external features as an input which outputs a sample in the configuration space. This sample output by the decoder

is used to minimize an objective function which aims to fundamentally reduce the divergence between the probability distribution of the training samples and the learned generative model to be able to closely reconstruct the training samples set. During testing, only the decoder network is used to generate the required samples. The decoder takes as input a latent variable sampled from standard normal distribution as well as the vector of external features to generate useful samples.

In our implementation of the CVAE, both encoder and decoder networks have two fully connected hidden layers with 512 units each. The specifics of the external features used in each of the planning problems considered in Chapter 4, Section 4.5 are discussed in Section B. The behavior of the generative model, in addition to the features used, also depends on certain parameters. We study the effect of these parameters and their design choices in our implementation in the following subsections.

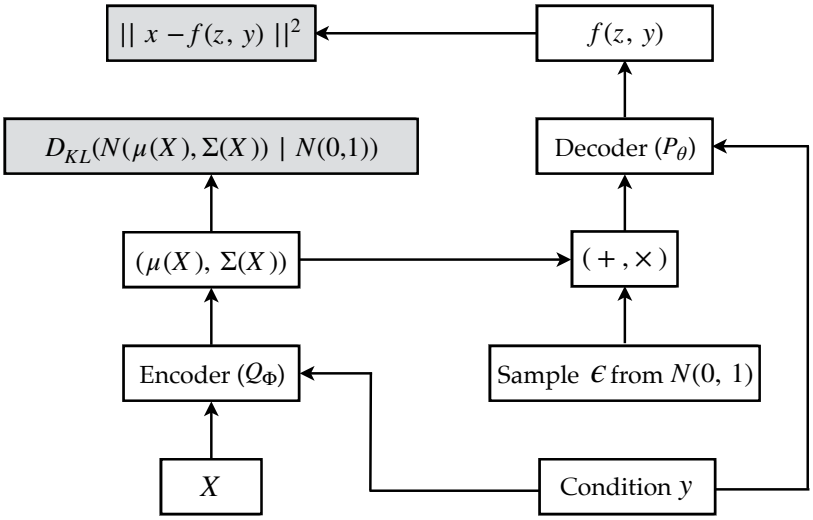
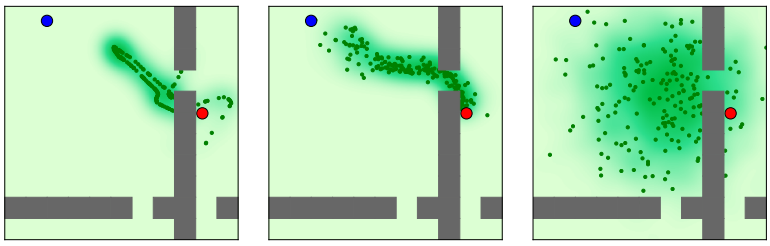


Figure B.1: A simple illustration of the CVAE framework setup for training with X and y together denoting the input to the CVAE.

Dimensionality of the Latent Variable

The latent variable captures the information available to the model through the training examples in a lower dimensional latent space. The dimensionality of the latent variable denotes how efficiently the model can capture the sources of variability required to regenerate data similar to the training examples. Theoretically, a model with larger latent dimension is at least as good as a model with lower latent dimension. However, in practice, when the latent variable dimension is high, it becomes computationally expensive for methods like stochastic gradient descent to reduce the KL divergence between the true and the

approximated distributions over the latent variables conditioned on the training examples. Figure B.2 shows the behaviors exhibited by the trained generative model for different latent variable dimensions. We choose latent variable dimension of 3 for $\mathbb{R}^2$, $\mathbb{R}^5$ problems and 5 for $\mathbb{R}^7$, $\mathbb{R}^8$ and $\mathbb{R}^9$ problems.



(a) (b) (c)

Figure B.2: Samples generated by CVAE trained with different latent variable dimensions (a) 1 (b) 3 and (c) 7.

Regularization Parameter

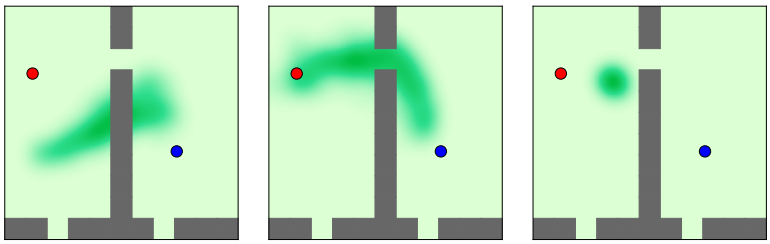
Although VAEs are generally devoid of regularization parameters, one could introduce the parameter in modifying the objective function the CVAE aims to minimize when learning the generative model. The objective function in a CVAE is given by:

$$\texttt{Reconstruction Loss} + \lambda \times \texttt{KL Divergence} \tag{B.1}$$

The reconstruction loss ensures that the training data can be explained with the data generated by the model and therefore minimizing it ensures proper reconstruction of the training examples. On the other hand, the second term captures the divergence between the prior distribution over latent variable and the posterior given the training examples. Minimizing it ensures that the two distributions are similar. When the value of λ is zero, the behavior of the corresponding VAE is similar to a traditional autoencoder in its capability to reconstruct the training examples. When the value of λ is equal to 1, the objective function is as in a VAE. However this often leads to *over-pruning* [127] where many of the dimensions of the latent variable are ignored in an attempt to reduce the KL divergence. By tuning the value of λ between 0 and 1, one could weigh the two objectives appropriately to obtain the desired generative model behavior (Figure B.3).

Experiments

In this section, we discuss the offline computation involved in training the CVAE for different planning environments considered in Section 6.4.2.



(a) (b) (c)

Figure B.3: Learned distributions for the narrow passage problem for different values of regularization parameter (λ), (a) 2×10^{-8} (b) 2×10^{-4} (chosen value of λ) (c) 2×10^{-2}.

Training Procedure

2D Point Robot Planning The training data consisted of 20 randomly generated environments as shown in Figure B.4 with 20 planning problems (start-goal pairs) in each of the environments. The environments were randomized in positions of the vertical and horizontal walls and the narrow passages through them. The CVAE was conditioned upon a vector of 102 features which included the start-goal pair (4 features) as well the 10×10 occupancy grid (100 features). The dataset generation took 4-5 hours while the training time was around 25 minutes. The CVAE was trained using samples from $\mathcal{G}_{\text{dense}}$ with 3000 samples. The CVAE was trained to sample configurations (in $\mathbb{R}^2$) of the point robot.

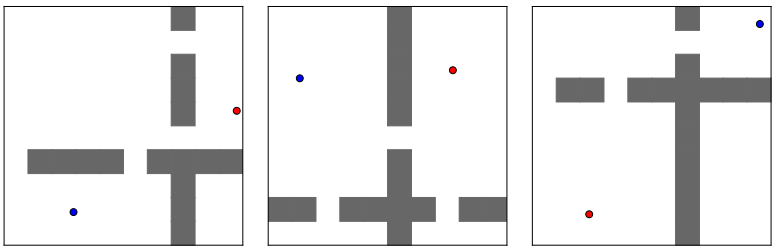


Figure B.4: Environments sampled in $\mathbb{R}^2$ to train the CVAE.

N-Link Arm Planning The training procedure for the robot in $\mathbb{R}^3$, $\mathbb{R}^5$ consisted of a $\mathcal{G}_{\text{dense}}$ with 6000 samples which was used to plan for 20 planning problems in each of 20 randomly generated 2D environments. Figure B.5 visualizes some of the environments sampled to train the CVAE. The red and blue positions show the start and goal states respectively. The environment has randomly placesd obstacles. The CVAE was conditioned on a vector of features which included the start-goal pair as well the 10×10 occupancy grid (100 features). The dataset generation took 6-7 hours while the training time was close to 30 minutes.

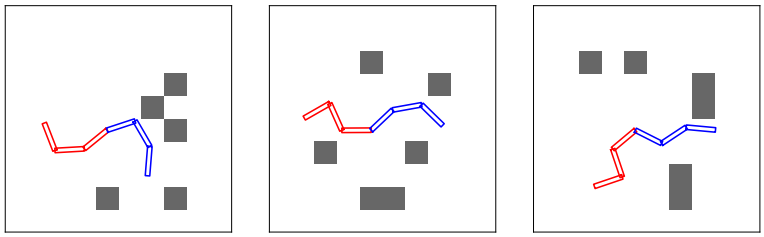


Figure B.5: Environments sampled in $\mathbb{R}^3$ to train the CVAE.

Snake Robot Planning For $\mathbb{R}^5$, the training procedure was similar to that in the $\mathbb{R}^2$ problems. The training procedure for the robot in $\mathbb{R}^9$ consisted of a $\mathcal{G}_{\text{dense}}$ with 6000 samples which was used to plan for 20 planning problems in each of 20 randomly generated 2D environments. Figure B.6 visualizes some of the environments sampled to train the CVAE. The red and blue positions show the start and goal states respectively. The environments were modified in the wall being horizontal or vertical, the offset in its position, and the position of the narrow passage through it. The CVAE was conditioned on a vector of 118 features which included the start-goal pair (18 features) as well the 10×10 occupancy grid (100 features). The dataset generation took 6-7 hours while the training time was close to 30 minutes. The CVAE was trained to sample configurations of the snake robot that included the base location as well as the revolute joint angles between each of the links.

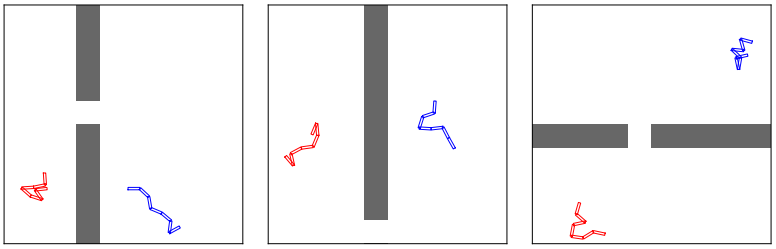


Figure B.6: Environments sampled in $\mathbb{R}^9$ to train the CVAE.

Manipulator Arm Planning The training data consisted of 20 random environments where the obstacles in the environment were arbitrarily repositioned. In each of the randomly generated environment, 50 planning problems were considered as an input to the train the CVAE model. Figure B.7 visualized three such environments, where the positions of the table and that of the obstacle on the table are modified along with start and goal configurations. The CVAE in the constrained problem was conditioned on a vector of 46[1] features which included the start and goal configurations (14 features) and the poses of the ta-

[1] 48 in the unconstrained problem since the configuration of the robot includes an additional degree of freedom.

ble and the obstacle represented as 4×4 homogeneous matrices (32 features). The dataset was generated in 7-8 hours while the training took around an hour. Samples from a $\mathcal{G}_{\text{dense}}$ with 30,000 configurations were used to train the CVAE. The CVAE learned to sample the robot configurations which included the joint angles at the seven revolute joints of the arm in the constrained example. The unconstrained $\mathbb{R}^8$ example consisted of an additional prismatic joint value denoting where the stick is held in the hand.

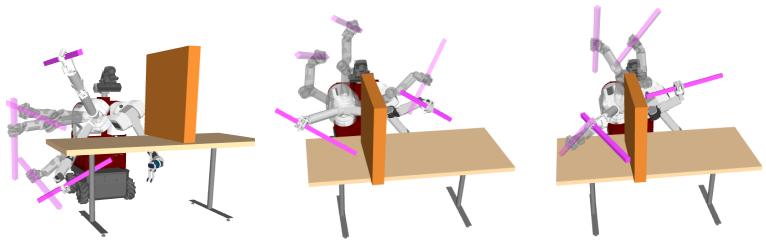


Figure B.7: Manipulator arm environments sampled to train the CVAE. The solutions obtained using samples generated by LEGO are also visualized.

Additional Experiment Results

BottleneckNode *and* DiversePathSet In addition to the qualitative observations presented in Chapter 4, Section 4.5 and Chapter 4, Figure 4.4, we present here the analysis of the performance of the foundational algorithms of LEGO , namely BottleneckNode and DiversePathSet when compared to ShortestPath . Figure B.9(a) shows that on a $\mathbb{R}^2$ world, BottleneckNode has a significantly higher success rate that ShortestPath , almost converging to 1.0 by 400 samples. Figure B.9(b) shows that in terms of path length, ShortestPath is initially better but both are eventually comparable. This is expected because of the near-optimality objective of Chapter 4, BottleneckNode (4.4). Figure B.9(c) shows that DiversePathSet has a better success rate. Finally, Figure B.9(d) shows that while both algorithms are comparable in terms of path length, DiversePathSet has a smaller variance.

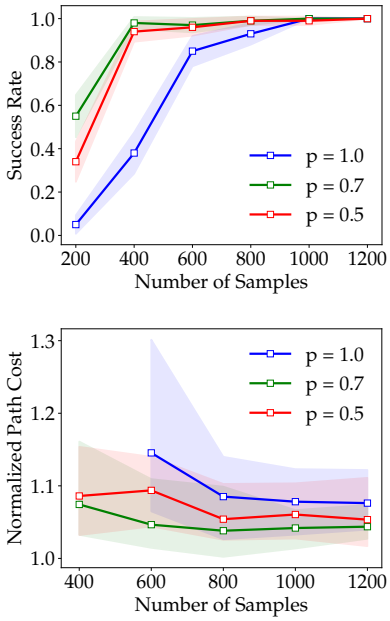


Figure B.8: Performance of LEGO on different graphs. The parameter p denotes the ratio of Halton samples to learned samples in the roadmap).

Roadmap Construction

To evaluate the performance of LEGO , we construct sparse roadmaps, $\mathcal{G}_{\text{sparse}}$. The sparse graph consisted of 200 samples in $\mathbb{R}^2$, $\mathbb{R}^5$ problems and 300 samples in case of $\mathbb{R}^7$, $\mathbb{R}^8$ and $\mathbb{R}^9$ problems. Not however, that this sparse roadmap contains both the learned samples as well as samples generated from Halton sequence. While the learned samples are concentrated near the bottleneck regions and along diverse paths, Halton samples ensure the coverage over the free regions of the

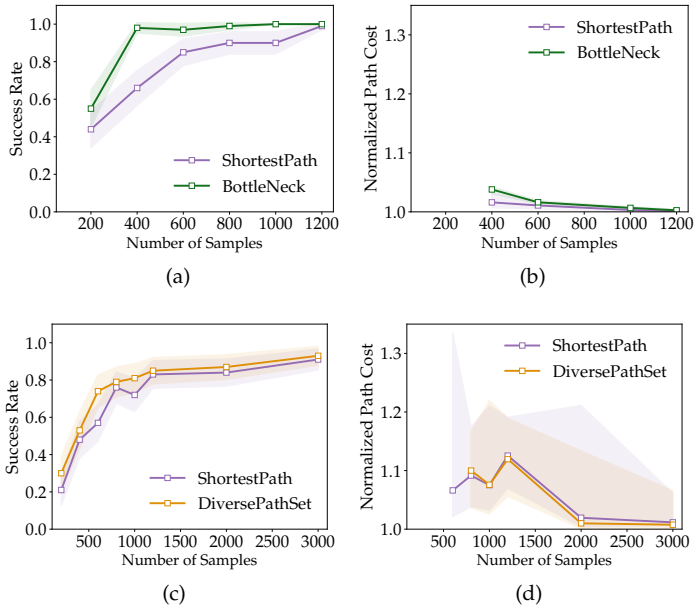


Figure B.9: Comparison of ShortestPath against BottleneckNode (top row) and DiversePathSet (bottom row) on success rate (left column) and normalized path length (right column).

configuration space as well. We analyze different proportions of Halton samples and learned samples. Figure B.8 shows the performance characteristics of LEGO on roadmaps constructed with different proportions of Halton and learned samples for the 2D point robot example. We observe that LEGO over a roadmap of 200 samples with just 30% learned samples significantly outperforms LEGO over a Halton graph ($p = 1$). Figure B.10 visualizes the samples generated by LEGO represented by the end-effector positions (blue) in the workspace.

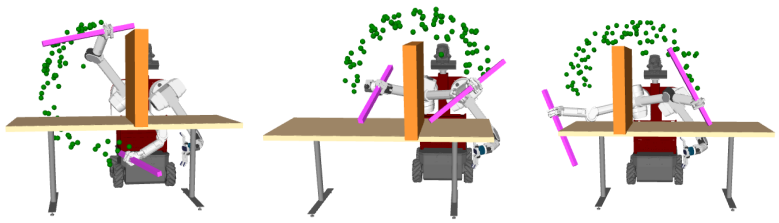


Figure B.10: Samples generated by LEGO for manipulator arm planning. The blue dots represent the end effector positions corresponding to the samples.

Bibliography

[1] S. Aine and M. Likhachev. Anytime truncated d* : Anytime replanning with truncation. In M. Helmert and G. Röger, editors, *Proceedings of the Sixth Annual Symposium on Combinatorial Search, SOCS 2013, Leavenworth, Washington, USA, July 11-13, 2013*. AAAI Press, 2013.

[2] S. Aine and M. Likhachev. Truncated incremental search. *Artificial Intelligence*, 234(C):49 – 77, May 2016.

[3] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev. Multi-heuristic a. *The International Journal of Robotics Research*, 35(1-3):224–243, 2016.

[4] N. M. Amato, O. B. Bayazit, and L. K. Dale. OBPRM: An obstacle-based PRM for 3D workspaces. 1998.

[5] O. Arslan and P. Tsiotras. Use of relaxation methods in sampling-based algorithms for optimal motion planning. In *ICRA*, pages 2421–2428, 2013.

[6] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM journal on computing*, 32(1):48–77, 2002.

[7] R. Bellman. The theory of dynamic programming. Technical report, DTIC Document, 1954.

[8] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[9] M. Bhardwaj, S. Choudhury, and S. Scherer. Learning heuristic search via imitation. In *Conference on Robot Learning*, pages 271–280. PMLR, 2017.

[10] M. Bhardwaj, S. Chowdhury, B. Boots, and S. Srinivasa. Leveraging experience in lazy search. 2019.

[11] J. Bialkowski, M. Otte, and E. Frazzoli. Free-configuration biased sampling for motion planning. In *IEEE International Conference on Intelligent Robots and Systems*, pages 1272–1279. IEEE, 2013.

[12] J. Bialkowski, M. W. Otte, S. Karaman, and E. Frazzoli. Efficient collision checking in sampling-based motion planning via safety certificates. *The International Journal of Robotics Research*, 35(7):767–796, 2016.

[13] 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.

[14] R. Bohlin and L. E. Kavraki. Path planning using lazy PRM. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 521–528. IEEE, 2000.

[15] 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*, pages 1018–1023, 1999.

[16] B. Burns and O. Brock. Sampling-based motion planning using predictive models. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 3120–3125. IEEE, 2005.

[17] B. Burns and O. Brock. Toward optimal configuration space sampling. In *Robotics: Science and Systems (RSS)*, 2005.

[18] C. Chamzas, Z. K. Kingston, C. Quintero-Peña, A. Shrivastava, and L. E. Kavraki. Learning sampling distributions using local 3d workspace decompositions for motion planning in high dimensions. *CoRR*, abs/2010.15335, 2020.

[19] C. Chamzas, A. Shrivastava, and L. E. Kavraki. Using local experiences for global motion planning. In *IEEE International Conference on Robotics and Automation*, pages 8606–8612, May 2019.

[20] N. Chen, M. Karl, and P. van der Smagt. Dynamic movement primitives in latent space of time-dependent variational autoencoders. In *Humanoids*, 2016.

[21] H. M. Choset. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

[22] S. Choudhury, M. Bhardwaj, S. Arora, A. Kapoor, G. Ranade, S. Scherer, and D. Dey. Data-driven planning via imitation learning. *The International Journal of Robotics Research*, 2018.

[23] S. Choudhury, C. M. Dellin, and S. S. Srinivasa. Pareto-optimal search over configuration space beliefs for anytime motion planning. In *IEEE International Conference on Intelligent Robots and Systems*, 2016.

[24] S. Choudhury, C. M. Dellin, and S. S. Srinivasa. Pareto-optimal search over configuration space beliefs for anytime motion planning. In *IEEE International Conference on Intelligent Robots and Systems*, pages 3742–3749, 2016.

[25] S. Choudhury, S. Javdani, S. Srinivasa, and S. Scherer. Near-optimal edge evaluation in explicit generalized binomial graphs. In *Advances in Neural Information Processing Systems*, pages 4634–4644, 2017.

[26] S. Choudhury, S. Srinivasa, and S. Scherer. Bayesian Active Edge Evaluation on Expensive Graphs. In *International Joint Conference on Artificial Intelligence*, pages 4890–4897, 2018.

[27] B. J. Cohen, M. Phillips, and M. Likhachev. Planning Single-arm Manipulations with n-Arm Robots. In *Robotics: Science and Systems (RSS)*, 2014.

[28] C.Zhang, J.Huh, and D.D.Lee. Learning Implicit Sampling Distributions for Motion Planning. *arXiv preprint arXiv:1806.01968*, 2018.

[29] S. Dalibard and J. Laumond. Linear dimensionality reduction in random motion planning. *The International Journal of Robotics Research*, 2011.

[30] R. Dechter and J. Pearl. The optimality of A* revisited. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 95–99, 1983.

[31] C. M. Dellin and S. S. Srinivasa. A unifying formalism for shortest path problems with expensive edge evaluations via lazy best-first search over paths with edge selectors. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 459–467, 2016.

[32] R. Diankov and J. Kuffner. Randomized statistical path planning. In *IEEE International Conference on Intelligent Robots and Systems*, 2007.

[33] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, Dec. 1959.

[34] A. Dobson and K. E. Bekris. Sparse roadmap spanners for asymptotically near-optimal motion planning. *The International Journal of Robotics Research*, 33(1):18–47, 2014.

[35] C. Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[36] A. Dor, E. Greenshtein, and E. Korach. Optimal and myopic search in a binary random vector. *Journal of applied probability*, 1998.

[37] M. A. et al. TensorFlow: Large-scale machine learning on heterogeneous systems.

[38] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 1998.

[39] 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.

[40] 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.

[41] J. D. Gammell, T. D. Barfoot, and S. S. Srinivasa. Batch informed trees (bit*): Informed asymptotically optimal anytime search. *IJRR*, 39(5), 2020.

[42] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot. Informed RRT*: Optimal incremental path planning focused through an admissible ellipsoidal heuristic. In *IEEE International Conference on Intelligent Robots and Systems*, 2014.

[43] J. D. Gammell, S. 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, ICRA*, 2015.

[44] J. D. Gammell, S. 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*, pages 3067–3074, 2015.

[45] A. V. Goldberg and C. Harrelson. Computing the shortest path: *A* search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 156–165. SIAM, 2005.

[46] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 129–143. SIAM, 2006.

[47] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In R. Raman and M. F. Stallmann, editors, *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*, pages 129–143. SIAM, 2006.

[48] J. Ha, H. Chae, and H. Choi. Approximate inference-based motion planning by learning and exploiting low-dimensional latent variable models. *IEEE Robotics and Automation Letters*, 2018.

[49] N. Haghtalab, S. Mackenzie, A. D. Procaccia, O. Salzman, and S. S. Srinivasa. The Provable Virtue of Laziness in Motion Planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 106–113, 2018.

[50] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.

[51] J. H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commununications of the ACM*, 7(12):701–702, Dec. 1964.

[52] P. E. Hart, N. J. 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, 1968.

[53] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 1968.

[54] K. Hauser. Lazy collision checking in asymptotically-optimal motion planning. In *IEEE International Conference on Robotics and Automation*, pages 2951–2957, 2015.

[55] C. Holleman and L. E. Kavraki. A framework for using the workspace medial axis in PRM planners. In *IEEE International Conference on Robotics and Automation*, 2000.

[56] R. C. Holte. Common misconceptions concerning heuristic search. In *Symposium on Combinatorial Search (SoCS)*, pages 46–51, 2010.

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

[58] D. Hsu, J. Latombe, and H. Kurniawati. On the probabilistic foundations of probabilistic roadmap planning. *The International Journal of Robotics Research*, 2006.

[59] D. Hsu, J. Latombe, and R. Motwani. Path planning in expansive configuration spaces. *IJCGA*, 4:495–512, 1999.

[60] D. Hsu, G. Sánchez-Ante, and Z. Sun. Hybrid PRM sampling with a cost-sensitive adaptive strategy. In *IEEE International Conference on Robotics and Automation*, 2005.

[61] J. Huh and D. D. Lee. Learning high-dimensional mixture models for fast collision detection in rapidly-exploring random trees. In *IEEE International Conference on Robotics and Automation*, 2016.

[62] B. Ichter, J. Harrison, and M. Pavone. Learning sampling distributions for robot motion planning. In *IEEE International Conference on Robotics and Automation*, 2018.

[63] B. Ichter and M. Pavone. Robot motion planning in learned latent spaces. *arXiv preprint arXiv:1807.10366*, 2018.

[64] L. Janson, B. Ichter, and M. Pavone. Deterministic sampling-based motion planning: Optimality, complexity, and performance. *The International Symposium on Robotics Research*, 2015.

[65] L. Janson, B. Ichter, and M. Pavone. Deterministic sampling-based motion planning: Optimality, complexity, and performance. *arXiv preprint arXiv:1505.00023*, 2015.

[66] 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.

[67] L. Janson, E. Schmerling, A. 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.

[68] S. S. Joshi and P. Tsiotras. Relevant region exploration on general cost-maps for sampling-based motion planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021*, pages 6689–6695. IEEE, 2020.

[69] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *Proc. Robotics: Science and Systems*, 2010.

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

[71] S. Karaman and E. Frazzoli. Incremental sampling-based algorithms for optimal motion planning. In *RSS*, 2010.

[72] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[73] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30(7):846–894, 2011.

[74] 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.

[75] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transsactions on Robotics and Automation*, 12(4):566–580, 1996.

[76] L. E. Kavraki, P. Svestka, J. C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics*, 1996.

[77] A. Khan, A. Ribeiro, V. Kumar, and A. G. Francis. Graph neural networks for motion planning. *CoRR*, abs/2006.06248, 2020.

[78] D. Kim, Y. Kwon, and S.-e. Yoon. Adaptive lazy collision checking for optimal sampling-based motion planning. In *International Conference on Ubiquitous Robots (UR)*, pages 320–327. IEEE, 2018.

[79] D. P. Kingma and M. Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[80] S. Koenig and M. Likhachev. Real-time adaptive a. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288, 2006.

[81] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1-2):93–146, 2004.

[82] S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence*, 155(1):93–146, 2004.

[83] S. Koenig and X. Sun. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems*, 18(3):313–341, Jun 2009.

[84] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97 – 109, 1985.

[85] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2):189 – 211, 1990.

[86] Y. Kuo, A. Barbu, and B. Katz. Deep sequential models for sampling-based planning. *arXiv preprint arXiv:1810.00804*, 2018.

[87] H. Kurniawati and D. Hsu. Workspace importance sampling for probabilistic roadmap planning. In *IEEE International Conference on Intelligent Robots and Systems*.

[88] H. Kurniawati and D. Hsu. Workspace-based connectivity oracle: An adaptive sampling strategy for prm planning. In *Algorithmic Foundation of Robotics VII*, pages 35–51. Springer, 2008.

[89] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

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

[91] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[92] M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Advances in neural information processing systems*, pages 767–774, 2004.

[93] A. Mandalika, S. Choudhury, O. Salzman, and S. Srinivasa. Generalized Lazy Search for Robot Motion Planning: Interleaving Search and Edge Evaluation via Event-based Toggles. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 745–753, 2019.

[94] A. Mandalika, O. Salzman, and S. Srinivasa. Lazy Receding Horizon A* for Efficient Path Planning in Graphs with Expensive-to-Evaluate Edges. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 476–484, 2018.

[95] M.Morales, L.Tapia, R.Pearce, S.Rodriguez, and N.M.Amato. A machine learning approach for feature-sensitive motion planning. In *Algorithmic Foundations of Robotics VI*, pages 361–376. Springer, 2005.

[96] S. Murray, W. Floyd-Jones, Y. Qi, D. J. Sorin, and G. Konidaris. Robot motion planning on a chip. In *Robotics: Science and Systems (RSS)*, 2016.

[97] V. Narayanan and M. Likhachev. Heuristic search on graphs with existence priors for expensive-to-evaluate edges. In *ICAPS*, 2017.

[98] C. L. Nielsen and L. E. Kavraki. A 2 level fuzzy prm for manipulation planning. In *IEEE International Conference on Intelligent Robots and Systems*, 2000.

[99] B. Paden, Y. Nager, and E. Frazzoli. Landmark guided probabilistic roadmap queries. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4828–4834. IEEE, 2017.

[100] J. Pan, S. Chitta, and D. Manocha. Faster sample-based motion planning using instance-based learning. In *WAFR*, 2012.

[101] J. Pearl. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. 1984.

[102] M. Phillips, B. Cohen, S. Chitta, and M. Likhachev. E-graphs: Bootstrapping planning with experience graphs. In *Proceedings of Robotics: Science and Systems*, 2012.

[103] A. H. Qureshi and M. C. Yip. Deeply Informed Neural Sampling for Robot Motion Planning. In *IEEE International Conference on Intelligent Robots and Systems*, pages 6582–6588. IEEE, 2018.

[104] 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.

[105] S. Rodriguez, S. Thomas, R. Pearce, and N. M. Amato. Resampl: A region-sensitive adaptive motion planner. In *Algorithmic Foundation of Robotics VII*, pages 285–300. Springer, 2008.

[106] O. Salzman and D. Halperin. Asymptotically-optimal motion planning using lower bounds on cost. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4167–4172. IEEE, 2015.

[107] O. Salzman and D. Halperin. Asymptotically-optimal Motion Planning using lower bounds on cost. In *IEEE International Conference on Robotics and Automation*, pages 4167–4172, 2015.

[108] O. Salzman and D. Halperin. Asymptotically Near-Optimal RRT for Fast, High-Quality Motion Planning. *IEEE Transactions on Robotics*, 32(3):473–483, 2016.

[109] M. Sharir. Algorithmic motion planning. In *Handbook of Discrete and Computational Geometry, Second Edition.*, pages 1037–1064. 2004.

[110] K. Sohn, H. Lee, and X. Yan. Learning structured output representation using deep conditional generative models. In *Advances in Neural Information Processing Systems*, 2015.

[111] 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.

[112] S. S. Srinivasa, D. Ferguson, C. J. Helfrich, D. Berenson, A. Collet, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, and M. V. Weghe. HERB: a home exploring robotic butler. *Autonomous Robots*, 28(1):5, Nov 2009.

[113] S.Srinivasa, D.Berenson, M.Cakmak, A.C.Romea, M.Dogar, A.Dragan, R.Knepper, T.Niemueller, K.Strabala, J.M.Vandeweghe, and J.Ziegler. Herb 2.0: Lessons learned from developing a mobile manipulator for the home. *Proceedings of the IEEE*, 100(8):1–19, July 2012.

[114] M. P. Strub and J. D. Gammell. Adaptively informed trees (ait*): Fast asymptotically optimal path planning through adaptive heuristics. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*, pages 3191–3198. IEEE, 2020.

[115] M. P. Strub and J. D. Gammell. Adaptively Informed Trees (AIT*): Fast Asymptotically Optimal Path Planning through Adaptive Heuristics. In *ICRA*, pages 3191–3198. IEEE, 2020.

[116] M. P. Strub and J. D. Gammell. Advanced bit* (abit*): Sampling-based planning with advanced graph-search techniques. In *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*, pages 130–136. IEEE, 2020.

[117] M. P. Strub and J. D. Gammell. Advanced BIT* (ABIT*): Sampling-Based Planning with Advanced Graph-Search Techniques. In *ICRA*, pages 130–136. IEEE, 2020.

[118] 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.

[119] I. A. Sucan, M. Moll, and L. E. Kavraki. The open motion planning library. *IEEE Robotics Automation Magazine*, 19(4):72–82, Dec 2012.

[120] I. A. Sucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics and Automation Magazine*, 19(4):72–82, December 2012. `http://ompl.kavrakilab.org`.

[121] X. Tang, J. Lien, and N. Amato. An obstacle-based rapidly-exploring random tree. In *IEEE International Conference on Robotics and Automation*, 2006.

[122] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE International Conference on Intelligent Robots and Systems*, 2017.

[123] J. P. Van den Berg and M. H. Overmars. Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners. *The International Journal of Robotics Research*, 2005.

[124] P. Vernaza and D. D. Lee. Learning dimensional descent for optimal motion planning in high-dimensional spaces. In *Association for the Advancement of Artificial Intelligence (AAAI)*, 2011.

[125] S. A. Wilmarth, N. M. Amato, and P. F. Stiller. MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *IEEE International Conference on Robotics and Automation*, 1999.

[126] Y. Yang and O. Brock. Adapting the sampling distribution in prm planners based on an approximated medial axis. In *IEEE International Conference on Robotics and Automation*, 2004.

[127] S. Yeung, A. Kannan, Y. Dauphin, and L. Fei-Fei. Tackling over-pruning in variational autoencoders. *arXiv preprint arXiv:1706.03643*, 2017.

[128] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 923–929, 2000.

[129] S. Zheng, Y. Yue, and J. Hobbs. Generating long-term trajectories using deep hierarchical networks. In *Advances in Neural Information Processing Systems*, 2016.

[130] M. Zucker, J. Kuffner, and J. A. Bagnell. Adaptive workspace biasing for sampling-based planners. In *IEEE International Conference on Robotics and Automation*, 2008.