A Unifying Formalism for Shortest Path Problems with Expensive Edge Evaluations via Lazy Best-First Search over Paths with Edge Selectors

Christopher M. Dellin and **Siddhartha S. Srinivasa**
The Robotics Institute, Carnegie Mellon University
{cdellin, siddh}@cs.cmu.edu

Abstract

While the shortest path problem has myriad applications, the computational efficiency of suitable algorithms depends intimately on the underlying problem domain. In this paper, we focus on domains where evaluating the edge weight function dominates algorithm running time. Inspired by approaches in robotic motion planning, we define and investigate the *Lazy Shortest Path* class of algorithms which is differentiated by the choice of an *edge selector* function. We show that several algorithms in the literature are equivalent to this lazy algorithm for appropriate choice of this selector. Further, we propose various novel selectors inspired by sampling and statistical mechanics, and find that these selectors outperform existing algorithms on a set of example problems.

1 Introduction

Graphs provide a powerful abstraction capable of representing problems in a wide variety of domains from computer networking to puzzle solving to robotic motion planning. In particular, many important problems can be captured as *shortest path problems* (Figure 1), wherein a path p^* of minimal length is desired between two query vertices through a graph G with respect to an edge weight function w.

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

The computational costs incurred by an algorithm can be broadly categorized into three sources corresponding to the blocks in Figure 1. One such source consists of queries on the structure of the graph G itself. The most commonly discussed such operation, *expanding* a vertex (determining its successors), is especially fundamental when the graph is represented implicitly, e.g. for domains with large graphs such as the 15-puzzle or Rubik's cube. It is with respect to vertex expansions that A* (Hart, Nilsson, and Raphael 1968) is optimally efficient.

A second source of computational cost consists of maintaining ordered data structures inside the algorithm itself, which is especially important for problems with large branching factors. For such domains, approaches such as partial expansion (Yoshizumi, Miura, and Ishida 2000) or iterative deepening (Korf 1985) significantly reduce the number of vertices generated and stored by either selectively filtering surplus vertices from the frontier, or by not storing the frontier at all.

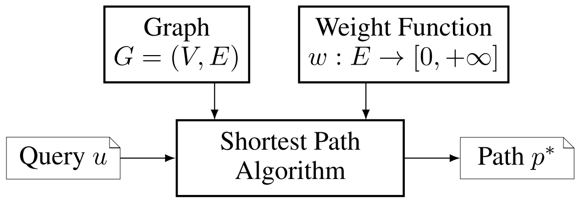


Figure 1: While solving a shortest path query, a shortest path algorithm incurs computation cost from three sources: examining the structure of the graph G, evaluating the edge weight function w, and maintaining internal data structures.

The third source of computational cost arises not from reasoning over the structure of G, but instead from evaluating the edge weight function w (i.e. we treat discovering an out-edge and determining its weight separately). Consider for example the problem of articulated robotic motion planning using roadmap methods (Kavraki et al. 1996). While these graphs are often quite small (fewer than 10^5 vertices), determining the weight of each edge requires performing many collision and distance computations for the complex geometry of the robot and environment, resulting in planning times of multiple seconds to find a path.

In this paper, we consider problem domains in which evaluating the edge weight function w dominates algorithm running time and investigate the following research question:

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

We make three primary contributions. First, inspired by lazy collision checking techniques from robotic motion planning (Bohlin and Kavraki 2000), we formulate a class of shortest-path algorithms that is well-suited to problem domains with expensive edge evaluations. Second, we show that several existing algorithms in the literature can be expressed as special cases of this algorithm. Third, we show that the extensibility afforded by the algorithm allows for

Algorithm 1 Lazy Shortest Path (LazySP)

```
1: function LAZYSHORTESTPATH($G, u, w, w_{\text{est}}$)
2:     $E_{\text{eval}} \leftarrow \emptyset$
3:     $w_{\text{lazy}}(e) \leftarrow w_{\text{est}}(e) \quad \forall e \in E$
4:     loop
5:         $p_{\text{candidate}} \leftarrow$ SHORTESTPATH($G, u, w_{\text{lazy}}$)
6:         if $p_{\text{candidate}} \subseteq E_{\text{eval}}$ then
7:             return $p_{\text{candidate}}$
8:         $E_{\text{selected}} \leftarrow$ SELECTOR($G, p_{\text{candidate}}$)
9:         for $e \in E_{\text{selected}} \setminus E_{\text{eval}}$ do
10:            $w_{\text{lazy}}(e) \leftarrow w(e)$        ▷ Evaluate (expensive)
11:            $E_{\text{eval}} \leftarrow E_{\text{eval}} \cup e$
```

novel edge evaluation strategies, which can outperform existing algorithms over a set of example problems.

2 Lazy Shortest Path Algorithm

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

Problem Definition

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

$$\text{len}(p, w) = \sum_{e \in p} w(e). \tag{1}$$

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

$$p^* = \underset{p \in P_u}{\arg\min}\, \text{len}(p, w). \tag{2}$$

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

The Algorithm

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

At each iteration of the search, the algorithm uses w_{lazy} to compute a candidate path $p_{\text{candidate}}$ by calling an existing solver SHORTESTPATH (note that this invocation requires no evaluations of w). Once a candidate path has been found, it is returned if it is fully evaluated. Otherwise, an *edge selector* is employed which selects graph edge(s) for evaluation. The true weights of these edges are then evaluated (incurring the requisite computational cost), and the algorithm repeats.

Algorithm 2 Various Simple LazySP Edge Selectors

```
1: function SELECTEXPAND($G, p_{\text{candidate}}$)
2:     $e_{\text{first}} \leftarrow$ first unevaluated $e \in p_{\text{candidate}}$
3:     $v_{\text{frontier}} \leftarrow G$.source($e_{\text{first}}$)
4:     $E_{\text{selected}} \leftarrow G$.out_edges($v_{\text{frontier}}$)
5:     return $E_{\text{selected}}$
6: function SELECTFORWARD($G, p_{\text{candidate}}$)
7:     return {first unevaluated $e \in p_{\text{candidate}}$}
8: function SELECTREVERSE($G, p_{\text{candidate}}$)
9:     return {last unevaluated $e \in p_{\text{candidate}}$}
10: function SELECTALTERNATE($G, p_{\text{candidate}}$)
11:    if LazySP iteration number is odd then
12:        return {first unevaluated $e \in p_{\text{candidate}}$}
13:    else
14:        return {last unevaluated $e \in p_{\text{candidate}}$}
15: function SELECTBISECTION($G, p_{\text{candidate}}$)
16:    return $\left\{ \begin{array}{l} \text{unevaluated } e \in p_{\text{candidate}} \\ \text{furthest from nearest evaluated edge} \end{array} \right\}$
```

LazySP is complete and optimal:

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

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

The optimality of LazySP depends on the admissibility of w_{est} in the same way that the optimality of A* depends on the admissibility of its goal heuristic h. Theorem 2 establishes the general bounded suboptimality of LazySP w.r.t. the inflation parameter ϵ. While our theoretical results (e.g. equivalences) hold for any choice of ϵ, for clarity our examples and experimental results focus on cases with $\epsilon = 1$. Proofs are available in (Dellin and Srinivasa 2016).

The Edge Selector: Key to Efficiency

The LazySP algorithm exhibits a rough similarity to optimal replanning algorithms such as D* (Stentz 1994) which plan a sequence of shortest paths for a mobile robot as new edge weights are discovered during its traverse. D* treats edge changes passively as an aspect of the problem setting (e.g. a sensor with limited range).

The key difference is that our problem setting treats edge evaluations as an active choice that can be exploited. While any choice of edge selector that meets the conditions above will lead to an algorithm that is complete and optimal, its *efficiency* is dictated by the choice of this selector. This motivates the theoretical and empirical investigation of different edge selectors in this paper.

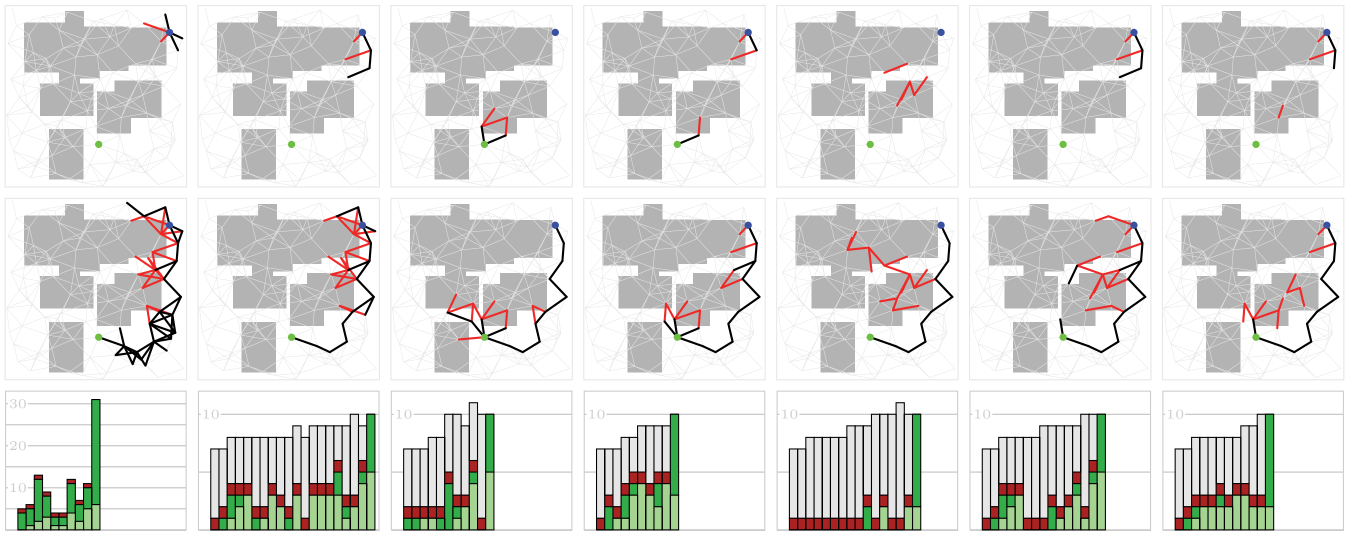

(a) Expand[77] (b) Forward[34] (c) Reverse[24] (d) Alternate[23] (e) Bisection[25] (f) WeightSamp[22] (g) Partition[22]

Figure 2: Snapshots of the LazySP algorithm using each edge selector discussed in this paper on the same obstacle roadmap graph problem, with start (●) and goal (●). At top, the algorithms after evaluating five edges (evaluated edges labeled as / valid or / invalid). At middle, the final set of evaluated edges. At bottom, for each unique path considered from left to right, the number of edges on the path that are 🟩 already evaluated, 🟩 evaluated and valid, 🟥 evaluated and invalid, and ⬜ unevaluated. The total number of edges evaluated is noted in brackets. Note that the scale on the Expand plot has been adjusted because the selector evaluates many edges not on the candidate path at each iteration.

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

The Alternate selector simply alternates between Forward and Reverse on each iteration. This can be motivated by both bidirectional search algorithms as well as motion planning algorithms such as RRT-Connect (Kuffner and LaValle 2000) which tend to perform well w.r.t. state evaluations.

The Bisection selector chooses among those unevaluated edges the one furthest from an evaluated edge on the candidate path. This selector is roughly analogous to the collision checking strategy employed by the Lazy PRM (Bohlin and Kavraki 2000) as applied to our problem on abstract graphs.

In the following section, we demonstrate that instances of LazySP using simple selectors yield equivalent results to existing vertex algorithms. We then discuss two more sophisticated selectors motivated by weight function sampling and statistical mechanics.

3 Edge Equivalence to A* Variants

In the previous section, we introduced LazySP as the path-selection analogue to BFS vertex-selection algorithms. In this section, we make this analogy more precise. In particular, we show that LazySP-Expand is edge-equivalent to a variant of A* (and Weighted A*), and that LazySP-Forward is edge-equivalent to a variant of Lazy Weighted A* (see Table 1). It is important to be specific about the conditions under which these equivalences arise, which we detail here. Proofs are available in (Dellin and Srinivasa 2016).

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

Table 1: LazySP equivalence results. The A*, LWA*, and BHFFA algorithms use reopening and the dynamic h_{lazy} heuristic (4).

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

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

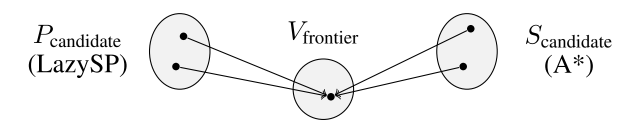


Figure 3: Illustration of the equivalence between A* and LazySP-Expand. After evaluating the same set of edges, the next edges to be evaluated by each algorithm can both be expressed as a surjective mapping onto a common set of unexpanded frontier vertices.

able to B such that the same edge(s) are evaluated by each.

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

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

$$h_{\text{est}}(v) = \min_{p:v \to v_g} \text{len}(p, w_{\text{est}}). \qquad (3)$$

Note that the value of this heuristic could be computed as a pre-processing step using Dijkstra's algorithm (Dijkstra 1959) before iterations begin. However, in order for the equivalences to hold, we require the use of the lazy heuristic

$$h_{\text{lazy}}(v) = \min_{p:v \to v_g} \text{len}(p, w_{\text{lazy}}). \qquad (4)$$

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

Equivalence to A*

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

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

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

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

We will use the goal heuristic h_{lazy} from (4). Note that if an admissible edge weight estimator $\hat{w}$ exists (that is, $\hat{w} \leq w$), then our A* can approximate the Weighted A* algorithm (Pohl 1970) with parameter ϵ by using $w_{\text{est}} = \epsilon\,\hat{w}$, and the suboptimality bound from Theorem 2 holds.

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

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

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

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

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

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

Equivalence to Lazy Weighted A*

In a conventional vertex expansion algorithm, determining a successor's cost is a function of both the cost of the edge and the value of the heuristic. If either of these components is expensive to evaluate, an algorithm can defer its computation by maintaining the successor on the frontier with an approximate cost until it is expanded. The Fast Downward algorithm (Helmert 2006) is motivated by expensive heuristic evaluations in planning, whereas the Lazy Weighted A* (LWA*) algorithm (Cohen, Phillips, and Likhachev 2014) is motivated by expensive edge evaluations in robotics.

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

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

The most obvious difference is that we present the original OPEN list as separate vertex (Q_v) and edge (Q_e) priority queues, with sorting keys shown on lines 3 and 4. A vertex v in the original OPEN with $trueCost(v) = true$ corresponds to a vertex v in Q_v, whereas a vertex v' in the original OPEN with $trueCost(v') = false$ (and parent v) corresponds to an edge (v, v') in Q_e. Use of the edge queue obviates the need for duplicate vertices on OPEN with different parents and the $conf(v)$ test for identifying such duplicates. This presentation also highlights the similarity between LWA* and the inner loop of the Batch Informed Trees (BIT*) algorithm (Gammell, Srinivasa, and Barfoot 2015).

Algorithm 3 Lazy Weighted A* (without CLOSED list)

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

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

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

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

We will use $h(v) = h_{\text{lazy}}(v)$ from (4) and $\hat{w} = w_{\text{lazy}}$. Note that the use of these dynamic heuristics requires that the Q_v and Q_e be resorted after every edge is evaluated.

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

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

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

Relation to Bidirectional Heuristic Search

LazySP-Alternate chooses unevaluated edges from either the beginning or the end of the candidate path at each iteration. We conjecture that an alternating version of the Expand selector is edge-equivalent to the Bidirectional Heuristic Front-to-Front Algorithm (Sint and de Champeaux 1977) for appropriate lazy vertex pair heuristic, and that LazySP-Alternate is edge-equivalent to a bidirectional LWA*.

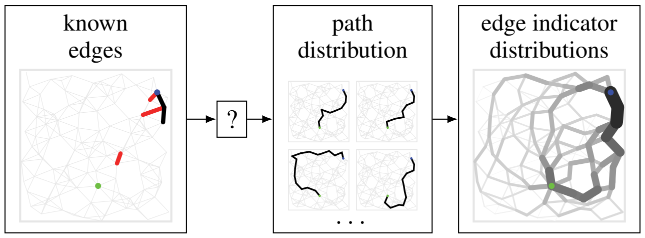


Figure 4: Illustration of maximum edge probability selectors. A distribution over paths (usually conditioned on the known edge evaluations) induces on each edge e a Bernoulli distribution with parameter $p(e)$ giving the probability that it belongs to the path. The selector chooses the edge with the largest such probability.

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

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

4 Novel Edge Selectors

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

Weight Function Sampling Selector

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

$$\mathcal{D}_p : \text{SP}(w) \text{ for } w \sim \mathcal{D}_w(E_{\text{eval}}). \quad (5)$$

For example, the distribution $\mathcal{D}_w$ might consist of the edge weights induced by a model of the distribution of environment obstacles (Figure 5). Since this obstacle distribution is conditioned on the results of known edge evaluations, we consider the subset of worlds which are consistent with the edges we have evaluated so far. However, depending on

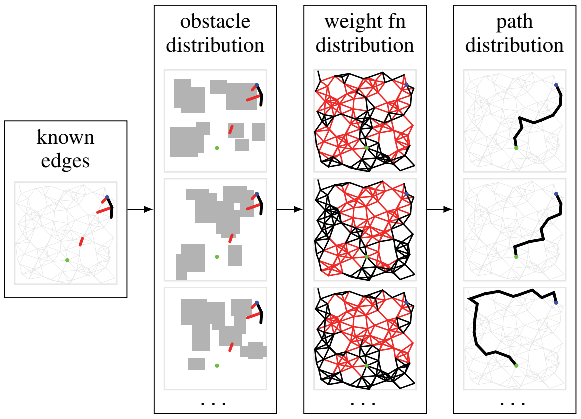


Figure 5: The WeightSamp selector uses the path distribution induced by solving the shortest path problem on a distribution over possible edge weight functions $\mathcal{D}_w$. In this example, samples from $\mathcal{D}_w$ are computed by drawing samples from $\mathcal{D}_O$, the distribution of obstacles that are consistent with the known edge evaluations.

the fidelity of this model, solving the corresponding shortest path problem for a given sampled obstacle arrangement might require as much computation as solving the original problem, since it requires computing the resulting edge weights. In practice, we can approximate $\mathcal{D}_w$ by assuming that each edge is independently distributed.

Partition Function Selector

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

$$\mathcal{D}_p : f_P(p) \propto \exp(-\beta \operatorname{len}(p, w_{\text{lazy}})). \tag{6}$$

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

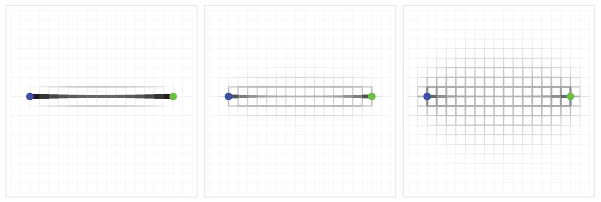
$$Z(P) = \sum_{p \in P} \exp(-\beta \operatorname{len}(p, w_{\text{lazy}})). \tag{7}$$

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

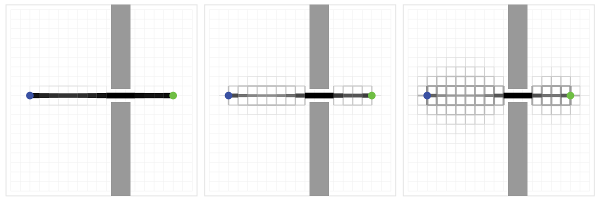
$$p(e) = 1 - \frac{Z(P \setminus e)}{Z(P)}. \tag{8}$$

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

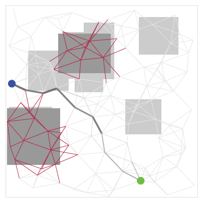
It may appear advantageous to restrict P to only *simple* paths, since all optimal paths are simple. Unfortunately, an algorithm for computing (7) efficiently is not currently known in this case. However, in the case that P consists of all paths, there exists an efficient incremental calculation of (7) via a recursive formulation which we detail here.



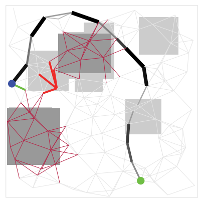
(a) Initial $p(e)$ scores on a constant-weight grid with β: 50, 33, 28



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



(c) Initial $p(e)$ scores



(d) Scores after five evaluations

Figure 6: Examples of the Partition selector's $p(e)$ edge score function. (a) With no known obstacles, a high β assigns near-unity score to only edges on the shortest path; as β decreases and more paths are considered, edges immediately adjacent to the roots score highest. (b) Since all paths must pass through the narrow passage, edges within score highly. (c) For a problem with two a-priori known obstacles (dark gray), the score first prioritizes evaluations between the two. (d) Upon finding these edges are blocked, the next edges that are prioritized lie along the top of the world.

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

$$Z'_{xy} = Z_{xy} + \frac{Z_{xa} Z_{by}}{\exp(\beta w_{ab}) - Z_{ba}} \text{ if } \exp(\beta w_{ab}) > Z_{ba}. \tag{9}$$

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

This incremental formulation of (7) allows for the corresponding score $p(e)$ for edges to be updated efficiently during each iteration of LazySP as the w_{lazy} value for edges

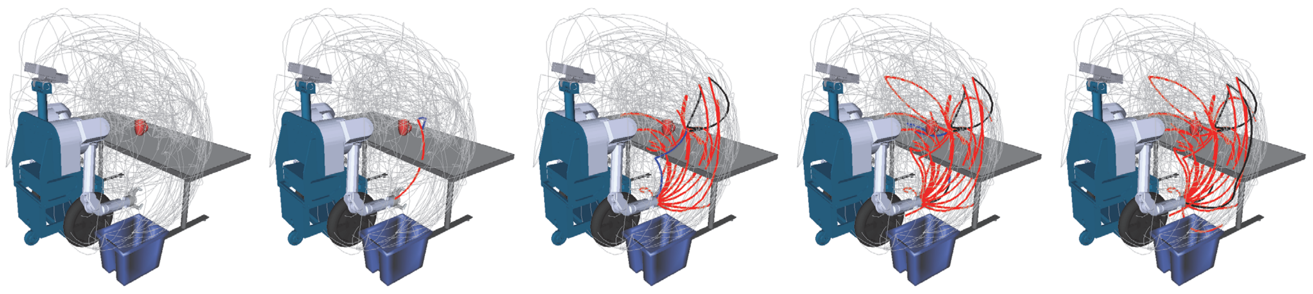


Figure 7: Visualization of the first of three articulated motion planning problems in which the HERB robot must move its right arm from the start configuration (pictured) to any of seven grasp configurations for a mug. Shown is the progression of the Alternate selector on one of the randomly generated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

chosen for evaluation are updated. In fact, if the values Z are stored in a square matrix, the update for all pairs after an edge weight change consists of a single vector outer product.

5 Experiments

We compared the seven edge selectors on three classes of shortest path problems. The average number of edges evaluated by each, as well as timing results from our implementations, are shown in Figure 8. In each case, the estimate was chosen so that $w_{\text{est}} \leq w$, so that all runs produced optimal paths. The experimental results serve primarily to illustrate that the A* and LWA* algorithms (i.e. Expand and Forward) are not optimally edge-efficient, but they also expose differences in behavior and prompt future research directions. All experiments were conducted using an open-source implementation.[1] Motion planning results were implemented using OMPL (Şucan, Moll, and Kavraki 2012).

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

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

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

Roadmap graphs for robot arm motion planning. We considered roadmap graphs in the configuration space corresponding to the 7-DOF right arm of the HERB home robot (Srinivasa et al. 2012) across three motion planning problems inspired by a table clearing scenario (see Figure 7). The problems consisted of first moving from the robot's home configuration to one of 7 feasible grasp configurations for a mug (pictured), second transferring the mug to one of 72 feasible configurations with the mug above the blue bin, and third returning to the home configuration. Each problem was solved independently. This common scenario spans various numbers of starts/goals and allows a comparison w.r.t. difficulty at different problem stages as discussed later.

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

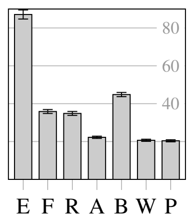
6 Discussion

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

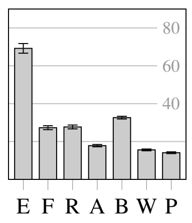
Since the Forward and Reverse selectors are simply mirrors of each other, they exhibit similar performance averaged across the PartConn and UnitSquare problem classes, which are symmetric. However, this need not the case for a particular instance. For example, the start of ArmPlan1 and the goal of ArmPlan3 consist of the arm's single home configuration in a relatively confined space. As shown in the table in Figure 8a, it appears that the better selector on these problems attempts to solve the more constrained side of the problem first. While it may be difficult to determine a priori which part of the problem will be the most constrained, the simple

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

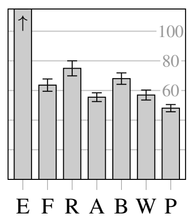
(a) Average number of edges evaluated for each problem class and selector. The minimum selector, along with any selector within one unit of its standard error, is shown in bold. The ArmPlan class is split into its three constituent problems. Online timing results are also shown, including the components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



(b) PartConn



(c) UnitSquare



(d) ArmPlan

Figure 8: Experimental results for the three problem classes across each of the seven selectors, E:Expand, F:Forward, R:Reverse, A:Alternate, B:Bisection, W:WeightSamp, and P:Partition. In addition to the summary table (a), the plots (b-d) show summary statistics for each problem class. The means and standard errors in (b-c) are across the 1000 and 900 problem instances, respectively. The means and standard errors in (d) are for the average across the three constituent problems for each of the 50 sampled roadmaps.

Alternate selector's respectable performance suggests that it may be a reasonable compromise.

The per-path plots at the bottom of Figure 2 allow us to characterize the selectors' behavior. For example, Alternate often evaluates several edges on each path before finding an obstacle. Its early evaluations also tend to be useful later, and it terminates after considering 10 paths on the illustrated problem. In contrast, Bisection exhibits a fail-fast strategy, quickly invalidating most paths after a single evaluation, but needing 16 such paths (with very little reuse) before it terminates. In general, the Bisection selector did not outperform any of the lazy selectors in terms of number of edges evaluated. However, it may be well suited to problem domains in which evaluations that fail tend be less costly.

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

The path distribution selectors are motivated by focusing evaluation effort in areas that are useful for many distinct candidate paths, as illustrated in Figure 6. Note that in the absence of a priori knowledge, the edges nearest to the start and goal tend to have the highest $p(e)$ score, since they are members of many potential paths. Because it tends to focus evaluations in a similar way, the Alternate selector may serve as a simple proxy for the more complex selectors.

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

Timing results. Figure 8a shows that the five simple selectors incur a negligible proportion of the algorithm's runtime. The WeightSamp and Partition selectors both require additional time (especially the former) in order to reduce the time spent evaluating edges. This tradeoff depends intimately on the problem domain considered. In the ArmPlan problem class, the Partition selector was able to reduce average total online runtime slightly despite an additional 1.54s of selector time. Note that Partition requires an expensive computation of the graph's initial Z-values, which are independent of the true weights and start/goal vertices (and can therefore be pre-computed, e.g. across all ArmPlan instances). Full timing results are available in (Dellin and Srinivasa 2016).

Optimizations. While we have focused on edge evaluations as the dominant source of computational cost, other considerations may also be important. There are a number of optimizations that allow for efficient implementation of LazySP.

The first relates to the repeated invocations of the inner shortest path algorithm (line 5 of Algorithm 1). Because only a small number of edges change weights between invocations, an incremental search algorithm such as SSSP (Ramalingam and Reps 1996) or LPA* (Koenig, Likhachev, and Furcy 2004) can be used to greatly improve the speed of the inner searches. Since the edge selector determines where on the graph edges are evaluated, the choices of the selector and the inner search algorithm are related. For example, using the Forward selector with an incremental inner search rooted at the goal results in behavior similar to D* (Stentz 1994) (albeit without the need to handle a moving start location) since a large portion of the inner tree can be reused.

An optimization commonly applied to vertex searches called *immediate expansion* is also applicable to LazySP. If an edge is evaluated with weight $w \leq w_{\text{est}}$, the inner search need not be run again before the next edge is evaluated.

A third optimization is applicable to domains with infinite edge costs (e.g. to represent infeasible transitions). If the length of the path returned by the inner shortest path algorithm is infinite, LazySP can return this path immediately even if some of its edges remain unevaluated without affecting its (sub)optimality. This reduces the number of edge evaluations needed in the case that no feasible path exists.

Other methods for expensive edge evaluations. An alternative to lazy evaluations is based on the observation that when solved by vertex expansion algorithms, such problems are characterized by slow vertex expansions. To mitigate this, approaches such as Parallel A* (Irani and Shih 1986) and Parallel A* for Slow Expansions (Phillips, Koenig, and Likhachev 2014) aim to parallelize such expansions. We believe that a similar approach can be applied to LazySP.

Another approach to finding short paths quickly is to relax the optimization objective (2) itself. While LazySP already supports a *bounded-suboptimal* objective via an inflated edge weight estimate (Theorem 2), it may also be possible to adapt the algorithm to address *bounded-cost* problems (Stern, Puzis, and Felner 2011).

Acknowledgements

We would like to thank Aaron Johnson and Michael Koval for their comments on drafts of this work. This work was (partially) funded by the National Science Foundation IIS (#1409003), Toyota Motor Engineering & Manufacturing (TEMA), and the Office of Naval Research.

References

Bohlin, R., and Kavraki, E. 2000. Path planning using Lazy PRM. In *IEEE International Conference on Robotics and Automation*, volume 1, 521–528.

Cohen, B.; Phillips, M.; and Likhachev, M. 2014. Planning single-arm manipulations with n-arm robots. In *Robotics: Science and Systems*.

Dellin, C. M., and Srinivasa, S. S. 2016. A unifying formalism for shortest path problems with expensive ege evaluations via lazy best-first search over paths with edge selectors. *arXiv* (1603.03490 [cs.DS]). Extended version with proofs and timing results.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerishe Mathematik* 1(1):269–271.

Gammell, J.; Srinivasa, S.; and Barfoot, T. 2015. 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*, 3067–3074.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Helmert, M. 2006. The fast downward planning system. *Artificial Intelligence Research* 26:191–246.

Irani, K. B., and Shih, Y. 1986. Parallel A* and AO* algorithms: An optimality criterion and performance evaluation. In *International Conference on Parallel Processing*, 274–277.

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

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

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

Kuffner, J., and LaValle, S. 2000. RRT-Connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, volume 2, 995–1001.

Phillips, M.; Koenig, S.; and Likhachev, M. 2014. Parallel A* for planning with time-consuming state expansions. In *International Conference on Automated Planning and Scheduling*.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1(34):193 – 204.

Ramalingam, G., and Reps, T. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21(2):267 – 305.

Sint, L., and de Champeaux, D. 1977. An improved bidirectional heuristic search algorithm. *J. ACM* 24(2):177–191.

Srinivasa, S.; Berenson, D.; Cakmak, M.; Collet, A.; Dogar, M.; Dragan, A.; Knepper, R.; Niemueller, T.; Strabala, K.; Vande Weghe, M.; and Ziegler, J. 2012. HERB 2.0: Lessons learned from developing a mobile manipulator for the home. *Proceedings of the IEEE* 100(8):2410–2428.

Stentz, A. 1994. Optimal and efficient path planning for partially-known environments. In *IEEE International Conference on Robotics and Automation*, volume 4, 3310–3317.

Stern, R.; Puzis, R.; and Felner, A. 2011. Potential search: A bounded-cost search algorithm. In *International Conference on Automated Planning and Scheduling*.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine* 19(4):72–82. http://ompl.kavrakilab.org.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with partial expansion for large branching factor problems. In *AAAI Conference on Artificial Intelligence*, 923–929.