Lazy Receding Horizon A* for Efficient Path Planning in Graphs with Expensive-to-Evaluate Edges

Aditya Mandalika
University of Washington
adityavk@cs.uw.edu *

Oren Salzman
Carnegie Mellon University
osalzman@andrew.cmu.edu *

Siddhartha Srinivasa
University of Washington
siddh@cs.uw.edu *

Abstract

Motion-planning problems, such as manipulation in cluttered environments, often require a collision-free shortest path to be computed quickly given a roadmap graph $\mathcal{G}$. Typically, the computational cost of evaluating whether an edge of $\mathcal{G}$ is collision-free dominates the running time of search algorithms. Algorithms such as Lazy Weighted A* (LWA*) and LazySP have been proposed to reduce the number of edge evaluations by employing a *lazy lookahead* (one-step lookahead and infinite-step lookahead, respectively). However, this comes at the expense of additional graph operations: the larger the lookahead, the more the graph operations that are typically required. We propose Lazy Receding-Horizon A* (LRA*) to minimize the *total planning time* by balancing edge evaluations and graph operations. Endowed with a lazy lookahead, LRA* represents a family of lazy shortest-path graph-search algorithms that generalizes LWA* and LazySP. We analyze the theoretic properties of LRA* and demonstrate empirically that, in many cases, to minimize the total planning time, the algorithm requires an intermediate lazy lookahead. Namely, using an intermediate lazy lookahead, our algorithm outperforms both LWA* and LazySP. These experiments span simulated random worlds in $\mathbb{R}^2$ and $\mathbb{R}^4$, and manipulation problems using a 7-DOF manipulator.

1 Introduction

Robotic motion-planning has been widely studied in the last few decades. Since the problem is computationally hard (Reif 1979; Sharir 2004), a common approach is to use sampling-based algorithms which typically construct a graph where vertices represent robot configurations and edges represent potential movements of the robot (Choset et al. 2005; LaValle 2006). A shortest-path algorithm then computes a path between two vertices on the graph.

There are numerous shortest-path algorithms, each suitable for a particular problem domain based on the computational efficiency of the algorithm. For example, A* (Hart, Nilsson, and Raphael 1968) is optimal with respect to node expansions, and planning techniques such as partial expansions (Yoshizumi, Miura, and Ishida 2000) and iterative deepening (Korf 1985a) are well-suited for problems with large graphs and large branching factors.

However, in most robotic motion-planning problems, path validations and edge evaluations are the major source of computational cost (LaValle 2006). Our work addresses these problems of quickly producing collision-free optimal paths, when the cost of evaluating an edge for collision is a computational bottleneck in the planning process.

A common technique to reduce the computational cost of edge evaluation and consequently the planning time is to employ a *lazy* approach. Two notable search-based planners that follow this paradigm are Lazy Weighted A* (LWA*) (Cohen, Phillips, and Likhachev 2014) and LazySP (Dellin and Srinivasa 2016; Haghtalab et al. 2017).

Both LWA* and LazySP assume there exists an efficient-to-compute lower bound on the weight of an edge. This lower bound is used as a lookahead (formally defined in Section 4) to guide the search without having to explicitly evaluate edges unless necessary. LazySP uses an infinite-step lookahead which can be shown to minimize the number of edge evaluations but requires large number of graph operations (node expansions, updating the shortest-path tree, etc.). On the other hand, LWA* uses a one-step lookahead which may result in a larger number of edge evaluations compared to LazySP but with much fewer graph operations.

Our key insight is that there should exist an optimal lookahead for a given environment, which balances the time for edge evaluations and graph operations, and minimizes the *total planning time*. We make the following contributions:

1. We present Lazy Receding-Horizon A* (LRA*), a family of lazy shortest-path algorithms parametrized by a *lazy lookahead* (Sections 4, 5) which allows us to continuously interpolate between LWA* and LazySP, balancing edge evaluations and graph operations.
2. We analyze the theoretic properties of LRA* (Section 6). Part of our analysis proves that LazySP is optimal with respect to minimizing edge evaluations thus closing a theoretic gap that was left open in (Dellin and Srinivasa 2016).
3. We demonstrate in Section 7, the efficacy of our algorithm on a range of planning problems for simulated $\mathbb{R}^n$ worlds and robot manipulators. We show that LRA* outperforms both LWA* and LazySP by minimizing not just edge evaluations or graph operations but the *total planning time*.

*This work was (partially) funded by the National Science Foundation IIS (#1409003), and the Office of Naval Research.

2 Related Work

A large number of motion-planning algorithms consist of (i) constructing a graph, or a roadmap, embedded in the configuration space and (ii) finding the shortest path in this graph. The graph can be constructed in a preprocessing stage (Kavraki et al. 1996; Karaman and Frazzoli 2011) or vertices and edges can be added in an incremental fashion (Gammell, Srinivasa, and Barfoot 2015; Salzman and Halperin 2015).

In domains where edge evaluations are expensive and dominate the planning time, a *lazy approach* is often employed (Bohlin and Kavraki 2000; Hauser 2015) wherein the graph is constructed *without* testing if edges are collision-free. Instead, the search algorithm used on this graph is expected to evaluate only a subset of the edges in the roadmap and hence save computation time. While standard search algorithms such as Dijkstra's (Dijkstra 1959) and A* (Hart, Nilsson, and Raphael 1968) can be used, specific search algorithms (Cohen, Phillips, and Likhachev 2014; Dellin and Srinivasa 2016) were designed for exactly such problems. They aim to further reduce the number of edge evaluations and thereby the planning time.

Alternative algorithms that reduce the number of edge evaluations have been studied. One approach was by foregoing optimality and computing near-optimal paths (Salzman and Halperin 2016; Dobson and Bekris 2014). Another approach was re-using information obtained from previous edge evaluations (Bialkowski et al. 2016; Choudhury, Dellin, and Srinivasa 2016; Choudhury et al. 2017).

In this paper, we propose an algorithm that makes use of a *lazy lookahead* to guide the search and minimize the total planning time. It is worth noting that the idea of a lookahead has previously been used in algorithms such as RTA*, LRTA* (Korf 1990) and LSS-LRTA* (Koenig and Sun 2009). However, these algorithms use the lookahead in a different context by interleaving planning with execution before the shortest path to the goal has been completely computed. Using a lazy lookahead has also been considered in the control literature (Kwon and Han 2006). Receding horizon optimization can be summarized as iteratively solving an optimal-control problem over a fixed future interval. Only the first step in the resulting optimal control sequence is executed and the process is repeated after measuring the state that was reached. Our lazy lookahead is analogous to the fixed horizon used by these algorithms. Additionally, the lazy lookahead can also be seen as a threshold that defines the extent to which (lazy) search is performed. This is similar to the Iterative Deepening version of A* (IDA*) (Korf 1985b) which performs a series of depth-first searches up to a (increasing) threshold over the solution cost.

3 Algorithmic Background

3.1 Single Source Shortest Path (SSSP) Problem

Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a cost function $w : \mathcal{E} \to \mathbb{R}^+$ on its edges, the Single Source Shortest Path (SSSP) problem is to find a path of minimum cost between two given vertices v_{source} and v_{target}. Here, a path $P = (v_1, \ldots, v_k)$ on the graph is a sequence of vertices where $\forall i, (v_i, v_{i+1}) \in \mathcal{E}$. An edge $e = (u, v)$ belongs to a path if $\exists i \; s.t. \; u = v_i, \; v = v_{i+1}$. The *cost* of a path is the sum of the weights of the edges along the path:

$$w(P) = \sum_{e \in P} w(e).$$

3.2 Solving the SSSP Problem

To solve the SSSP problem, algorithms such as Dijkstra (Dijkstra 1959) compute the shortest path by incrementally building a shortest-path tree $\mathcal{T}$ rooted at v_{source} and terminate once v_{target} is reached. This is done by maintaining a minimal-cost priority queue Q of nodes called the OPEN list. Each node τ_u is associated with a vertex u as well as with a pointer to u's parent in $\mathcal{T}$. The nodes are ordered in Q according to their cost-to-come i.e., the cost to reach u from v_{source} in $\mathcal{T}$.

The algorithm begins with $\tau_{v_{\text{source}}}$ (associated with v_{source}) in Q with a cost-to-come of 0. All other nodes are initialized with a cost-to-come of ∞. At each iteration, the node τ_u with the minimal cost-to-come is removed from Q and *expanded*, wherein the algorithm considers each of u's neighbours v, and evaluates if the path to reach v through u is cheaper than v's current cost-to-come. If so, then τ_v's parent is set to be τ_u (an operation we refer to as "rewiring") and is inserted into Q.

The search, i.e., the growth of $\mathcal{T}$, can be biased towards v_{target} using a heuristic function $h : \mathcal{V} \to \mathbb{R}$ which estimates the cost-to-go, i.e., the cost to reach v_{target} from vertex $v \in \mathcal{V}$. It can be shown that under mild conditions on h, if Q is ordered according to the sum of the cost-to-come and the estimated cost-to-go, then this algorithm, called A*, *expands* fewer nodes, during search, than any other search algorithm with the same heuristic (Dechter and Pearl 1983; Pearl 1984).

A key observation in the described approach for Dijkstra or A* is that for *every* node in Q, the algorithm computed the cost $w(e)$ of the edge to reach this node from its current parent, a process we will refer to as *evaluating* an edge. Edge evaluation occurs for all edges leading to nodes in the OPEN list Q irrespective of whether there exists a better parent to the node or whether the node will subsequently be expanded for search. In problem domains where computing the weight of an edge is an expensive operation, such as in robotic applications, this is highly inefficient. To alleviate this problem, we can apply *lazy* approaches for edge evaluations that can dramatically reduce the number of edges evaluated.

3.3 Computing SSSP via Lazy Computation

In problem domains where computing $w(e)$ is expensive, we assume the existence of a function $\hat{w} : \mathcal{E} \to \mathbb{R}^+$ which (i) is efficient to compute and (ii) provides a lower bound on the true cost of an edge i.e., $\forall e \in \mathcal{E}, \hat{w}(e) \leq w(e)$. We call $\hat{w}$ a *lazy estimate* of w.

Given such a function, Cohen, Phillips, and Likhachev proposed LWA* which modifies A* as follows: Each edge (u, v) is evaluated, i.e., $w(u, v)$ is computed, only when the algorithm believes that τ_v should be the next node to be expanded. Specifically, each node in Q is augmented with a flag stating whether the edge leading to this vertex has been

evaluated or not. Initially, every edge is given the estimated value computed using $\hat{w}$ for its cost and this is used to order the nodes in Q. Only when a node is selected for expansion, is the true cost of the edge leading to it evaluated. After the edge is evaluated, its cost may be found to be higher than the lazy estimate, or even ∞ (if the edge is untraversable—a notion we will refer to as "in collision"). In such cases, we simply discard the node. If it is valid, we now know the true cost of the edge, as well as the true cost-to-come for this vertex from its current parent. The node is marked to have its true cost determined and is inserted into Q again. Only when this node is chosen from Q the second time will it actually be expanded to generate paths to its neighbours. The algorithm terminates when v_{target} is removed from Q for the second time.

This approach increases the size of Q as there can be multiple nodes associated with every vertex, one for each incoming edge. Since the true cost of an edge is unknown until evaluated, it is essential that all these nodes be stored. Although this causes an increase in computational complexity and in the memory footprint of the algorithm, the approach can lead to fewer edges evaluated and hence reduce the overall running time of the search.

LWA* uses a one-step lookahead to reduce the number of edge evaluations. Namely, every path in the shortest-path tree $\mathcal{T}$ may contain one edge (the last) which has only been evaluated lazily. Taking this idea to the limit, Dellin and Srinivasa proposed the Lazy Shortest Path, or LazySP algorithm which uses an infinite-step lookahead. Specifically, it runs a series of shortest-path searches on the graph defined using $\hat{w}$ for all unevaluated edges. At each iteration, it chooses the shortest path to the goal and evaluates edges along this path[1]. When an edge is evaluated, the algorithm considers the evaluated true cost of the edge for subsequent iterations of the search. Hence, LazySP evaluates only those edges which potentially lie along the shortest path to the goal. The algorithm terminates when all the edges along the current shortest path have been evaluated to be valid (namely, not in collision).

A naïve implementation of LazySP would require running a complete shortest-path search every iteration. However, the search tree computed in the previous iterations can be reused: When an edge is found to be in collision, the search tree computed in previous iteration is locally updated using dynamic shortest-path algorithms such as LPA* (Koenig, Likhachev, and Furcy 2004).

3.4 Motivation

As described in Section 3.3, LWA* and LazySP attempt to reduce the number of edge evaluations by delaying evaluations until necessary. As we shall prove in Section 6, LazySP (with a lookahead of infinity), minimizes the number of edge evaluations, at the expense of greater graph operations. When an edge is found to be in collision, the entire subtree emanating from that edge needs to be updated (a process we will refer to as *rewiring*) to find the new shortest path to each node in the subtree. On the other hand, LWA* which has a lookahead of one, evaluates a larger number of edges relative to LazySP but does not perform any rewiring or repairing. When an edge to a node is found to be invalid, the node is simply discarded and the algorithm continues.

Therefore, these two algorithms, LWA* and LazySP, with a one-step and an infinite-step lookahead respectively, form two extremals to an entire spectrum of potential lazy-search algorithms based on the lookahead chosen. We aim to leverage the advantage that the lazy lookahead can provide to interpolate between LWA* and LazySP, and strike a balance between edge evaluations and graph operations to minimize the total planning time.

4 Problem Formulation

In this section we formally define our problem. To make this section self contained, we repeat definitions that were mentioned in passing in the previous sections. We consider the problem of finding the shortest path between source and target vertices v_{source} and v_{target} on a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Since we are motivated by robotic applications where edge evaluation, i.e., checking if the robot collides with its environment while moving along an edge, is expensive, we do not build the graph $\mathcal{G}$ with just feasible edges. Rather, as in the lazy motion-planning paradigm, the idea is to construct a graph with edges *assumed* to be feasible and delay the evaluation to only when absolutely necessary.

For simplicity, we assume the lazy estimate $\hat{w}$ to tightly estimate the true cost w for edges that are collision-free[2]. Therefore $\hat{w}$ is a *lazy estimate* of w such that

$$w(e) = \begin{cases} \hat{w}(e) & \text{if } e \text{ is not in collision,} \\ \infty & \text{if } e \text{ is in collision.} \end{cases} \tag{1}$$

We use the cost function w and its lazy estimate $\hat{w}$ to define the cost of a path on the graph. The *(true) cost* of a path P is the sum of the weights of the edges along P:

$$w(P) = \sum_{e \in P} w(e).$$

Similarly, the *lazy cost* of a path is the sum of the lazy estimates of the edges along the path:

$$\hat{w}(P) = \sum_{e \in P} \hat{w}(e).$$

Our algorithm will make use of paths which are only partially evaluated. Specifically, every path P will be a concatenation of two paths $P = P_{\text{head}} \cdot P_{\text{tail}}$ (here, $(\cdot)$ denotes the concatenation operator). Edges belonging to P_{head} will have been evaluated and known to be collision-free while edges belonging to P_{tail} will only be lazily evaluated. Notice that P_{tail} may be empty. We also define the *estimated total cost* of a path $P = P_{\text{head}} \cdot P_{\text{tail}}$ as:

$$\bar{w}(P) = w(P_{\text{head}}) + \hat{w}(P_{\text{tail}}).$$

[1] In the original exposition of LazySP, the method for which edges are evaluated along the shortest path is determined using a procedure referred to as an *edge selector*. In our work we consider the most natural edge selector, called forward edge selector. Here, the first unevaluated edge closest to the source is evaluated.

[2] We discuss relaxing the assumption that $\hat{w}(e)$ tightly estimates $w(e)$ in Section 8. In the general case, we require only that it is a lower bound i.e., $\forall e \in \mathcal{E}, \hat{w}(e) \leq w(e)$.

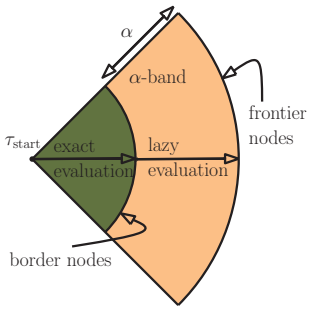


Figure 1: Search space of LRA*.

Although $\hat{w}$ helps guide the search of a *lazy* algorithm, as in LWA* or LazySP, as noted in Section 3.4, it can lead to additional computational overhead when the estimate is wrong, i.e., when the search algorithm encounters edges in collision. In this work we balance this computational overhead with the number of edge evaluations, by endowing our search algorithm with a *lookahead* α. In essence, the *lookahead* controls the extent to which we use $\hat{w}$ to guide our search.

As we will see later in Section 8, the lookahead α can be interpreted in various ways. However, in this paper we interpret the lookahead as the *number of edges* over which we use $\hat{w}$ to guide our search.

5 Lazy Receding-Horizon A* (LRA*)

5.1 Algorithmic Details

Our algorithm maintains a lazy shortest-path tree $\mathcal{T}$ over the graph $\mathcal{G}$. Every node in $\mathcal{T}$ is associated with a vertex of $\mathcal{G}$ and the tree is rooted at the node τ_{source} associated with the vertex v_{source}. We define the node entry $\tau \in \mathcal{T}$ as $\tau = (u, p, c, \ell, b)$, were $u[\tau] = u$ is the vertex associated with τ, $p[\tau] = p$ is τ's parent in $\mathcal{T}$ which can be backtracked to compute a path $P[\tau]$ from v_{source} to u. The node τ also stores $c[\tau] = c$ and $\ell[\tau] = \ell$ which are the costs of the evaluated and lazily-evaluated portions of $P[\tau]$, respectively. Namely, $c[\tau] = w(P[\tau]_{\text{head}})$ and $\ell[\tau] = \hat{w}(P[\tau]_{\text{tail}})$. Finally, $b[\tau] = b$ is the budget of $P[\tau]$ i.e., the number of edges that have been lazily evaluated in $P[\tau]$ or equivalently, the number of edges in $P[\tau]_{\text{tail}}$. Given a lookahead α, our algorithm will maintain shortest paths to a set of nodes represented by the search tree $\mathcal{T}$, where $\forall \tau \in \mathcal{T}$, $b[\tau] \leq \alpha$. The budget of any node in $\mathcal{T}$ never exceeds α.

Given a node $\tau \in \mathcal{T}$, we call it a *frontier* node if $b[\tau] = \alpha$ ($P[\tau]_{\text{tail}}$ has exactly α edges). Additionally, τ is said to belong to the *α-band* if $b[\tau] > 0$. We call τ a *border* node if it does not belong to the α-band but one of its children does. Finally, τ is called a *leaf* node if it has children in $\mathcal{G}$ but not in $\mathcal{T}$. Note that all frontier nodes are leaf nodes. See Fig. 1 for reference.

The algorithm maintains a priority queue Q_{frontier} that stores the frontier nodes ordered according to the estimated cost-to-come $\bar{w}(P[\tau]) = c[\tau] + \ell[\tau]$. This queue is used to choose which path to evaluate at each iteration. For ease of exposition, we present a high-level description of the algorithm (Alg. 1). For detailed pseudo-code, see (Mandalika, Salzman, and Srinivasa 2018).

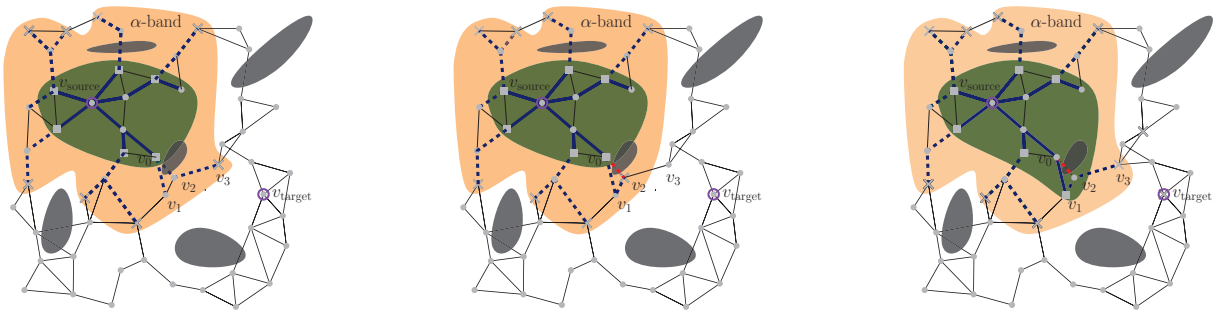
Algorithm 1 LRA*($\mathcal{G}$, v_{source}, v_{target}, α)

```
1:  Q_frontier, T := ∅                                   ▷ Initialization
2:  insert τ_{v_source} = (v_source, NIL, 0, 0, 0) into T
3:  for each leaf node τ ∈ T do                          ▷ Extend α-band
4:      add all nodes at distance (α − b[τ]) edges into T
5:  insert all frontier nodes in T into Q_frontier
6:  while Q_frontier is not empty do                     ▷ Search
7:      remove τ with minimal key w̄(τ) from Q_frontier
8:      evaluate first edge (u, v) along P[τ]_tail       ▷ Expensive
9:      if (u, v) is collision-free then
10:         update τ_v
11:         if v = v_target then
12:             return P[τ_{v_target}]
13:         update descendants τ of τ_v s.t τ ∈ T
14:     else                                             ▷ Edge is in collision
15:         remove edge (u, v) from graph
16:         for each descendant τ of τ_v s.t τ ∈ T do
17:             rewire τ to the best parent τ' ∈ T, τ' ≠ τ_{v_target}
18:     repeat steps 3-5 to extend the α-band
19: return failure
```

We are now ready to describe our algorithm, Lazy Receding-Horizon A* (LRA*), which begins by initializing the node τ_{source} associated with v_{source} (line 2). Our algorithm maintains the invariant that at the beginning of any iteration all leaf nodes are frontier nodes. When the algorithm starts, $\tau_{v_{\text{source}}}$ is a leaf node with $b[\tau_{v_{\text{source}}}] = 0 < \alpha$. Therefore we extend the α-band (lines 3-4) and consequently the search tree $\mathcal{T}$, adding all the frontier nodes to Q_{frontier} (line 5).

The algorithm iteratively finds the frontier node τ with minimal estimated cost (line 7) and evaluates the first edge along the lazy portion $P[\tau]_{\text{tail}}$ of the path $P[\tau]$ from $\tau_{v_{\text{source}}}$ to τ in $\mathcal{T}$ (line 8). If a collision-free shortest path to v_{target} is found during this evaluation (line 11-12), the algorithm terminates. Every evaluation of a collision-free edge (u, v) causes the node τ_v, that was previously in the α-band, to be a border node. Consequently the node entry is updated and this update is cascaded to all the nodes in the α-band belonging to the subtree rooted at τ_v (lines 10, 13). Specifically, the new cost, lazy cost and budget of τ_v is used to update the nodes in its subtree. However, if the edge (u, v) is found to be in collision, the edge is removed from the graph, and the entire subtree of τ_v is rewired appropriately (lines 14-17). This can potentially lead to some of the nodes being removed from the α-band. Both updating and rewiring subtrees can generate leaf nodes with budget less than α. Therefore at the end of the iteration, the α-band is again extended to ensure all leaf nodes have budget equal to the lookahead α (line 18). See Fig. 2 for an illustration.

As we will show in Section 6, the algorithm described is guaranteed to terminate with the shortest path, if one exists, and is hence complete for all values of α.



(a) (b) (c)

Figure 2: Visualization of LRA* running on $\mathcal{G}$ embedded in a workspace cluttered with obstacles (depicted in dark grey) and $\alpha = 2$. The regions where edges are evaluated and lazily evaluated are depicted by green and orange regions, respectively. Shortest-path tree $\mathcal{T}$ in the two regions is depicted by solid and dashed blue edges, respectively. Finally, vertices associated with border and frontier nodes are depicted by squares and crosses, respectively. Figure is best viewed in color. (a) Node associated with v_3 has the minimal key and the path ending with nodes v_0, v_2, v_3 is evaluated. Edge (v_0, v_2) is found to be in collision. (b) Node τ_2 associated with v_2 is rewired and the α-band is recomputed. Now τ_2 has the minimal key and the path ending with nodes v_0, v_1, v_2 is evaluated and found to be collision free. (c) The α-band is extended from v_2.

5.2 Implementation Details—Lazy computation of the α-band

Every time an edge (u, v) is evaluated, a series of updates is triggered (Alg. 1 lines 13 and 16-17) Specifically, let τ be the node associated with v and $\mathcal{T}(\tau)$ be the subtree of $\mathcal{T}$ rooted at τ. If the edge (u, v) is collision-free, then the budget of all the nodes $\mathcal{T}(\tau)$ needs to be updated. Alternatively, if the edge (u, v) is in collision, then a new path to every node in $\mathcal{T}(\tau)$ needs to be computed. These updates may be time-consuming and we would like to minimize them. To this end, we propose the following optimization which reduces the size of the α-band and subsequently, potentially reduces the number of nodes in $\mathcal{T}(\tau)$.

We suggest that if we already know that a node τ' in the α-band will *not* be part of a path that is chosen for evaluation in an iteration, then we defer expanding the α-band through this node. The key insight behind the optimization is that there is no need to expand a node τ' in the α-band if its key, $\bar{w}(P[\tau'])$, is larger than the key of the first node in Q_{frontier}. Using this optimization may potentially reduce the size of $\mathcal{T}(\tau)$ and save computations.

5.3 Implementation Details—Heuristically guiding the search

We described our algorithm as a lazy extension of Dijkstra's algorithm which orders its search according to cost-to-come. In practice we will want to heuristically guide the search similar to A*, which orders its search queues according to the sum of cost-to-come to a vertex from v_{source} and an estimate of the cost-to-go to v_{target} from the vertex, i.e., a heuristic.

We apply a similar approach by assuming that the algorithm is given a heuristic function that under estimates the cost to reach v_{target}. In Section 6 we state and prove that as the heuristic is strictly more informative, the number of edge evaluations and rewires further reduce, for a given lazy lookahead.

5.4 Discussion—LRA* as an approximation of optimal heuristic

In this section, we provide an intuition on the role lazy lookahead plays when guided by a heuristic. Given a graph $\mathcal{G}$, we can define the optimal heuristic $h^*_{\mathcal{G}}(v)$ as the length of the shortest path from v to v_{target} in $\mathcal{G}$. Indeed, if all edges of $\mathcal{G}$ are collision-free, an algorithm such as A* guided by $h^*_{\mathcal{G}}$ will only evaluate edges along the shortest path to v_{target}. To take advantage of this, LazySP proceeds by computing $h^*_{\mathcal{G}}$. If an edge is found to be in collision, it is removed from $\mathcal{G}$ and $h^*_{\mathcal{G}}$ is recomputed. This is why no other algorithm can perform fewer edge evaluations (see Section 6).

Using a finite *lookahead* and a *static* admissible heuristic, LRA* can be seen as a method to *approximate* the optimal heuristic. Every frontier node τ is associated with the key $c[\tau] + \ell[\tau] + h(u[\tau])$. The minimal of all such keys forms the approximation for the optimal heuristic $h^*_{\mathcal{G}}(v_{\text{source}})$ i.e., if τ_v associated with vertex v has the minimal key, we have,

$$h^*_{\mathcal{G}}(v_{\text{source}}) \geq c[\tau_v] + \ell[\tau_v] + h(v) \geq h(v_{\text{source}})$$

and the algorithm chooses to evaluate an edge along the path from v_{source} to v in $\mathcal{T}$. This approximation improves as the α-band approaches the target. When the algorithm starts, this approximation may be crude (when a small lazy lookahead is used). However, as the algorithm proceeds and α-band is expanded, this approximation dynamically converges to the optimal heuristic. We formalize this idea in Section 6 and show this phenomenon empirically in Section 7.

5.5 Discussion—Is greediness beneficial?

A possible extension to LRA* is to employ greediness in edge evaluation: Given a path, we currently evaluate the first edge along this path (Alg. 1, lines 7 and 8). However, we can choose to evaluate more than one edge, hence performing an exploitative action. This introduces a second parameter $\beta \leq \alpha$ 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. This is only the case when we seek *optimal* paths. If we relax the algorithm to produce suboptimal paths, greediness may be of use in early termination. While this relaxation is out of the scope of the paper, proofs pertaining to the superiority of no greediness are addressed in (Mandalika, Salzman, and Srinivasa 2018) for the case that optimal paths are required.

6 Correctness, Optimality and Complexity

In this section we provide theoretical properties regarding our family of algorithms LRA*. For proofs please see (Mandalika, Salzman, and Srinivasa 2018). We start in Section 6.1 with a correctness theorem stating that upon termination of the algorithm, the shortest path connecting v_{source} and v_{target} is found. We continue in Section 6.2 to detail how the lazy lookahead affects the performance of the algorithm with respect to edge evaluations. Specifically, we show that for $\alpha = \infty$, the algorithm is edge optimal. That is, it tests the minimal number of edges possible (this notion is formally defined). Furthermore, we examine how the lazy lookahead affects the number of edges evaluated by our algorithm. Finally, in Section 6.3 we bound the running time of the algorithm as well as its space complexity as a function of the lazy lookahead α. Here, we show that the running time (governed, in this case, by graph operations) can grow exponentially with the lazy lookahead α. This further backs our intuition that in order to minimize the running time in practice, an intermediate lookahead is required to balance edge evaluation and graph operations.

The following additional notation will be used throughout this section: Let P^*_v denote the shortest collision-free path from v_{source} to a vertex v and let $w^*(v) = w(P^*_v)$ be the minimal *true* cost-to-come to reach v from v_{source}. Finally, for the special case of v_{target}, we will use $w^* = w^*(v_{\text{target}})$. That is, w^* denotes the minimal cost-to-come to reach v_{target} from v_{source}.

6.1 Correctness

Lemma 1. *Let (v_0, v) be an edge evaluated by LRA* and found to be collision free. Then the shortest path to the node τ_v associated with vertex v has been found and $c[\tau_v] = w^*(v)$.*

Replacing v with v_{target}, we have,

Corollary 1. *LRA* is complete, i.e., if an edge (v_0, v_{target}) is found to be collision-free, the shortest path to $\tau_{v_{\text{target}}}$ associated with v_{target} has been found.*

6.2 Edge Optimality

In this section we analyze how the lazy lookahead allows to balance between the number of edge evaluations and rewiring operations. We start by looking at the extreme case where there is an infinite lookahead ($\alpha = \infty$). We define a natural and general family of algorithms $\mathcal{SP}$ that solve the shortest-path problem and show in Lemma 2 that when $\alpha = \infty$, no other algorithm in $\mathcal{SP}$ can perform fewer edge evaluations. We then continue by showing in Lemma 3 that the larger the lookahead, the fewer edge evaluations an algorithm LRA* will perform.

Recall that a shortest-path problem consists of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a lazy estimate of the weights $\hat{w}$, a weight function w and start and goal vertices, v_{source} and v_{target}, respectively. Let $\mathcal{SP}$ be the family of shortest-path algorithms that, given a shortest-path problem, proceed by building a shortest-path tree $\mathcal{T}$ rooted at v_{source}. Additionally, assume that for every shortest-path problem, there are no two paths in $\mathcal{G}$ that have the same weight[3].

An algorithm $\texttt{ALG} \in \mathcal{SP}$ can only call the weight function w for an edge $e = (u, v)$ where $u \in \mathcal{T}$. Furthermore, when terminating, it must report the shortest path from v_{source} to v_{target} and validate that no shorter path exists. Thus, if $P^* = P^*_{v_{\text{target}}}$ denotes the shortest path connecting v_{source} to v_{target}, then for any other path P connecting v_{source} to v_{target} with $\hat{w}(P) < w(P^*)$, ALG must explicitly test an edge $e \in P$ with $w(e) = \infty$. Notice that since ALG constructs a shortest-path tree, this will be the first edge on P that is in collision.

Finally, an algorithm $\texttt{ALG} \in \mathcal{SP}$ is said to be *edge-optimal* if for any other algorithm $\texttt{ALG'} \in \mathcal{SP}$, and any shortest-path problem, ALG will test no more edges than ALG'.

Lemma 2. *LRA* with $\alpha = \infty$ is edge-optimal.*

An immediate corollary that follows is,

Corollary 2. *LazySP is edge-optimal.*

Lemma 3. *For every graph $\mathcal{G}$ and every $\alpha_1 > \alpha_2$, we have that $E_1 \subseteq E_2$. Here, E_i denotes the set of edges evaluated by LRA* with $\alpha = i$.*

6.3 Complexity

In this section we analyse LRA* with respect to the space (Lemma 4) and running time (Lemma 5) complexity.

Lemma 4. *The total space complexity of our algorithm is bounded by $O(n + m)$, where n and m are the number of vertices and edges in $\mathcal{G}$, respectively.*

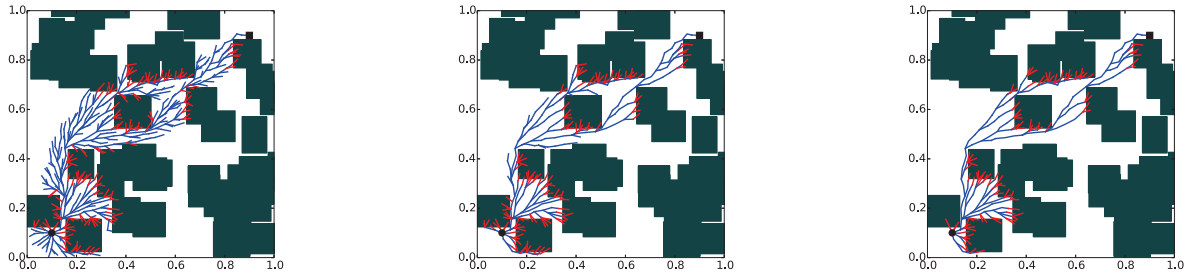
Lemma 5. *The total running time of the algorithm is bounded by $O(nd^{\alpha} \cdot \log(n) + m)$, where n and m are the number of vertices and edges, d is the maximal degree of a vertex and α is the lookahead.*

7 Results

In this section we empirically evaluate LRA*. We start by demonstrating the different properties of LRA* as a *family* of algorithms parameterized by α. Specifically, we show that to minimize the total planning time, an optimal lookahead α^* exists (where $1 < \alpha^* < \infty$) that allows to balance between edge evaluation and graph operations.

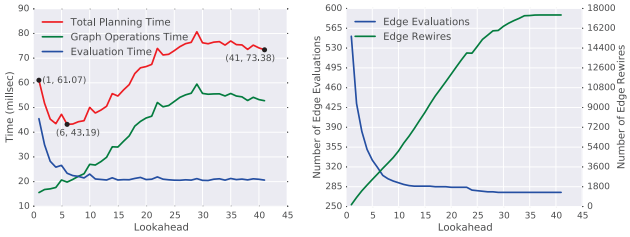
We then continue to evaluate properties of the optimal lookahead α^*. While choosing the exact lookahead value is out of the scope of the paper (see Section 8), we provide general guidelines regarding this choice.

[3]It is required that no two paths have the same weight to avoid handling tie-breaking in our proofs. Note that LRA* itself does not require this assumption.



(a) LWA* (b) LRA* (α^*) (c) LazySP

Figure 3: Visualization of edge evaluations by (a) LWA*, (b) LRA* with an optimal lookahead α^*, and (c) LazySP. Source and target are $(0.1, 0.1)$ and $(0.9, 0.9)$, respectively. Edges evaluated to be in collision and free are marked red and blue, respectively.



(a) (b)

Figure 4: Computation times (a) and number of operations (b) of LRA* as a function of the lookahead α.

7.1 Experimental Setup

We evaluated LRA* on a range of planning problems in simulated random $\mathbb{R}^2$ and $\mathbb{R}^4$ environments as well as real-world manipulation problems on HERB (Srinivasa et al. 2009), a mobile manipulator with 7-DOF arms. We implemented the algorithm using the Open Motion Planning Library (OMPL) (Sucan, Moll, and Kavraki 2012)[4]. Our source code is publicly available and can be accessed at https://github.com/personalrobotics/LRA-star.

Random environments We generated 10 different random environments for $\mathbb{R}^2$ and $\mathbb{R}^4$. For a given environment, we considered 10 distinct random roadmaps for a total of 100 trials for each dimension. Each roadmap was constructed by generating a set of vertices in a unit hypercube using Halton sequences (Halton 1964), which are characterized by low dispersion. The vertex positions were offset by uniform random values to generate distinct roadmaps. 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 (Janson et al. 2015). The number of vertices was chosen such that the roadmap contained a solution. Specifically, it was 2000 for $\mathbb{R}^2$ and 3000 for $\mathbb{R}^4$.

The source and target were set to $(0.1, 0.1, \ldots, 0.1)^d$ and $(0.9, 0.9, \ldots, 0.9)^d$, respectively, with $d \in \{2, 4\}$. For the 2D environments, the obstacles were a set of axis-aligned hypercubes that occupied 70% of an environment to simulate a cluttered space. One such randomly-generated environment is shown in Fig. 3 along with the edges evaluated by LWA*, LazySP and LRA* with an optimal lookahead. For the 4D environments, we chose a maze generated similar to the recursive mazes defined by Janson et al.. The choice of such a maze in $\mathbb{R}^4$ is motivated by the fact that it is inherently a *hard* problem to solve, since many lazy shortest paths need to be invalidated before a true shortest path is determined by the planner. A more detailed discussion about the complexity of the recursive maze problem is found in Janson et al..

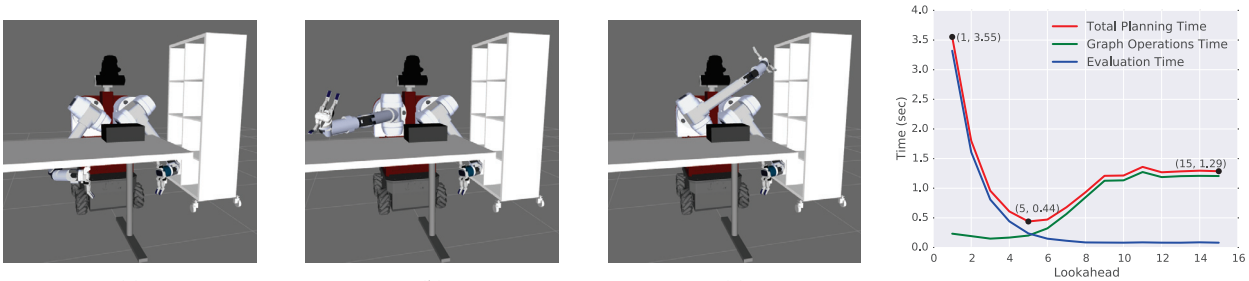
Manipulation Our manipulation problems simulate the task of reaching into a bookshelf while avoiding obstacles such as a table. We consider 10 different roadmaps, each with 30,000 vertices constructed by applying a random offset to the 7D Halton sequence. Two vertices are connected if their Euclidean distance is less than $r = 1.3$ radians. These choices are similar to the simulated $\mathbb{R}^n$ worlds, where we choose r using the bounds provided by Janson et al. and enough vertices such that we are ensured a solution exists on the roadmap. Fig. 5 illustrates the environment and the planning problem considered.

7.2 Properties of LRA*

Figures 3 and 5 visualize the search space for our simulated $\mathbb{R}^2$ environments and manipulation problems. For both settings, we ran LRA* with a range of lookahead values.

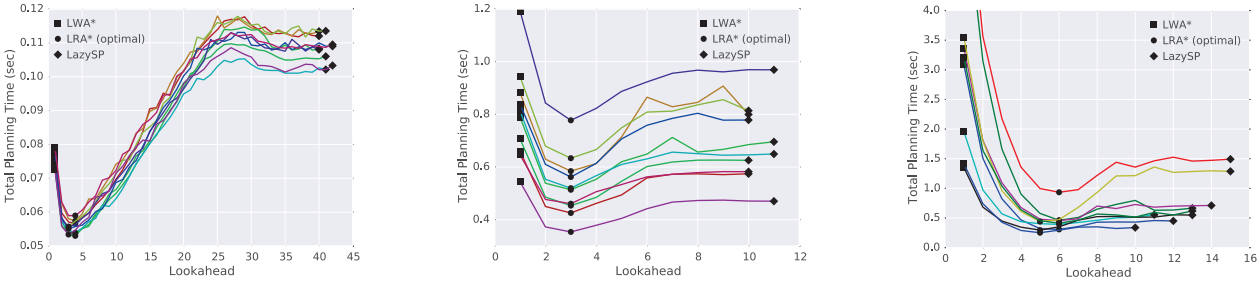
Notice that the number of edge evaluations as a function of the lookahead is a monotonically decreasing function (Fig. 4b and Lemma 3). However, the time spent on edge evaluations (Fig. 4a) is not monotonic since the time for evaluating an edge depends on the edge length and if it is in collision. Nevertheless, the overall trend of this plot decreases as the lookahead increases. In addition, the time spent on rewiring (Fig. 4a and 5d) roughly increases with lookahead. These two trends show that in both experiments, an intermediate lookahead does indeed balance edge evaluations and graph operations, therefore reducing the planning time. For additional experiments, see (Mandalika, Salzman, and Srinivasa 2018).

[4]Simulations were run on a desktop machine with 16GB RAM and an Intel i5-6600K processor running a 64-bit Ubuntu 14.04.



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

Figure 5: Manipulation experiments. (a-c) HERB is required to reach into the bookshelf while avoiding collision with the table. (d) Edge evaluation, rewiring and total planning time as a function of the lookahead.



(a) $\mathbb{R}^2$ environments (b) $\mathbb{R}^4$ environments (c) Manipulation environments

Figure 6: Planning time vs. lookahead for similar problems on different environments.

7.3 Properties of Optimal Lookahead α^*

While determining how to choose the lookahead value for a specific problem instance is beyond the scope of this paper (see Section 8), we provide some insight on some properties of optimal lookahead α^*. In Fig. 6 we plotted the planning time as a function of the lookahead for different random instances. We observe two phenomena: (i) the value of the optimal lookahead α^* has a very small variance when considering similar environments. Thus, if we will face multiple problems on a specific type of environment, it may be beneficial to run a preprocessing phase to estimate α^*. (ii) As the dimension increases, the relative speedup over LazySP diminishes. We conjecture that this is because the cost of edge evaluation increases with the complexity of the robot.

8 Future Work

Setting the lazy lookahead Our formulation assumed that the lazy lookahead α is fixed and provided by the user. In practice, we would like to automatically find the value of α and, possibly, change its value through the running time of the algorithm. This is especially useful when the search algorithm is interleaved with graph construction—namely, when vertices and edges are incrementally added to $\mathcal{G}$ (see, e.g., (Gammell, Srinivasa, and Barfoot 2015)).

Non-tight estimates of edge weights In this paper we assumed $\hat{w}$ tightly estimates the true cost w (see Eq. 1), however it can be easily extended to take into account non-tight estimates. Once an edge (u, v) is evaluated, if its true cost is larger than the estimated cost, the entire subtree rooted at v may need to be rewired to potentially better parents.

Alternative budget definitions and optimization criteria In this paper, we defined the budget and the optimization criteria in terms of number of unevaluated edges and path length respectively. However, the same approach can be used for alternative definitions. For example, we can define the budget in terms of the *length* of the unevaluated path. This definition is somewhat more realistic since the computational cost of evaluating an edge is typically proportional to its length. A different optimization criteria that we wish to consider is minimizing the *expected* number of edges checked given some belief over the probability that edges are collision-free. This can be further extended to balance between path length (a proxy for execution time) and number of edge evaluations (a proxy for planning time) where the optimization criteria would be some combination of path length and probability of being collision-free.

9 Acknowledgements

The authors would like to thank Shushman Choudhury, previously at Personal Robotics Lab, currently at Stanford University, for his valuable insights in the development of this work.

References

Bialkowski, J.; Otte, M. W.; Karaman, S.; and Frazzoli, E. 2016. Efficient collision checking in sampling-based motion

planning via safety certificates. *I. J. Robotics Res.* 35(7):767–796.

Bohlin, R., and Kavraki, L. E. 2000. Path planning using lazy PRM. In *ICRA*, volume 1, 521–528. IEEE.

Choset, H.; Lynch, K. M.; Hutchinson, S.; Kantor, G.; Burgard, W.; Kavraki, L. E.; and Thrun, S. 2005. *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT Press.

Choudhury, S.; Salzman, O.; Choudhury, S.; and Srinivasa, S. S. 2017. Densification strategies for anytime motion planning over large dense roadmaps. In *ICRA*, 3770–3777.

Choudhury, S.; Dellin, C. M.; and Srinivasa, S. S. 2016. Pareto-optimal search over configuration space beliefs for anytime motion planning. In *IROS*, 3742–3749.

Cohen, B. J.; Phillips, M.; and Likhachev, M. 2014. Planning Single-arm Manipulations with N-Arm Robots. In *RSS*.

Dechter, R., and Pearl, J. 1983. The Optimality of A* Revisited. In *AAAI*, 95–99.

Dellin, C. M., and Srinivasa, S. S. 2016. A Unifying Formalism for Shortest Path Problems with Expensive Edge Evaluations via Lazy Best-First Search over Paths with Edge Selectors. In *ICAPS*, 459–467.

Dijkstra, E. W. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1(1):269–271.

Dobson, A., and Bekris, K. E. 2014. Sparse roadmap spanners for asymptotically near-optimal motion planning. *I. J. Robotics Res.* 33(1):18–47.

Gammell, J. D.; Srinivasa, S. S.; and Barfoot, T. D. 2015. Batch Informed Trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *ICRA*, 3067–3074.

Haghtalab, N.; Mackenzie, S.; Procaccia, A. D.; Salzman, O.; and Srinivasa, S. S. 2017. The Provable Virtue of Laziness in Motion Planning. *CoRR* abs/1710.04101.

Halton, J. H. 1964. Algorithm 247: Radical-inverse Quasi-random Point Sequence. *Commun. ACM* 7(12):701–702.

Hart, P. E.; Nilsson, N. J.; 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.

Hauser, K. 2015. Lazy collision checking in asymptotically-optimal motion planning. In *ICRA*, 2951–2957.

Janson, L.; Schmerling, E.; Clark, A. A.; and Pavone, M. 2015. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *I. J. Robotics Res.* 34(7):883–921.

Karaman, S., and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *I. J. Robotics Res.* 30(7):846–894.

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

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

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong Planning A*. *Artif. Intell.* 155(1-2):93–146.

Korf, R. E. 1985a. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence* 27:97–109.

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

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

Kwon, W. H., and Han, S. H. 2006. *Receding horizon control: model predictive control for state models*. Springer Science & Business Media.

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

Mandalika, A.; Salzman, O.; and Srinivasa, S. 2018. Lazy Receding Horizon A* for Efficient Path Planning in Graphs with Expensive-to-Evaluate Edges. https://personalrobotics.cs.washington.edu/publications/mandalika2018lrastarfull.pdf. [Online; accessed 13-March-2018].

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

Reif, J. H. 1979. Complexity of the mover's problem and generalizations. In *FOCS*, 421–427. IEEE.

Salzman, O., and Halperin, D. 2015. Asymptotically-optimal Motion Planning using lower bounds on cost. In *ICRA*, 4167–4172.

Salzman, O., and Halperin, D. 2016. Asymptotically Near-Optimal RRT for Fast, High-Quality Motion Planning. *IEEE Trans. Robotics* 32(3):473–483.

Sharir, M. 2004. Algorithmic motion planning. In *Handbook of Discrete and Computational Geometry, Second Edition.* 1037–1064.

Srinivasa, S. S.; Ferguson, D.; Helfrich, C. J.; Berenson, D.; Collet, A.; Diankov, R.; Gallagher, G.; Hollinger, G.; Kuffner, J.; and Weghe, M. V. 2009. HERB: a home exploring robotic butler. *Autonomous Robots* 28(1):5.

Sucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics Automation Magazine* 19(4):72–82.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with Partial Expansion for Large Branching Factor Problems. In *AAAI*, 923–929.