Provable Indefinite-Horizon Real-Time Planning for Repetitive Tasks

Fahad Islam, Oren Salzman, Maxim Likhachev[*]
The Robotics Institute, Carnegie Mellon University
{fi,osalzman}@andrew.cmu.edu, maxim@cs.cmu.edu

Abstract

In many robotic manipulation scenarios, robots often have to perform highly-repetitive tasks in structured environments e.g. sorting mail in a mailroom or pick and place objects on a conveyor belt. In this work we are interested in settings where the tasks are similar, yet not identical (e.g., due to uncertain orientation of objects) and motion planning needs to be extremely fast. Preprocessing-based approaches prove to be very beneficial in these settings—they analyze the configuration-space offline to generate some auxiliary information which can then be used in the query phase to speedup planning times. Typically, the tighter the requirement is on query times the larger the memory footprint will be. In particular, for high-dimensional spaces, providing real-time planning capabilities is extremely challenging. While there are planners that guarantee real-time performance by limiting the planning horizon, we are not aware of general-purpose planners capable of doing it for indefinite horizon (i.e., planning to the goal). To this end, we propose a preprocessing-based method that provides *provable* bounds on the query time while incurring only a small amount of memory overhead in the query phase. We evaluate our method on a 7-DOF robot arm and show a speedup of over tenfold in query time when compared to the PRM algorithm.

1 Introduction

We consider the problem of planning robot motions for highly-repetitive tasks while ensuring bounds on the planning times. There exists manipulation domains where the planning takes non-trivial amount of time, despite the fact that the scenarios are well-structured. Specifically we consider the settings where the environment does not change, the start and goal in each task are similar, yet not identical to the start and goal in previous tasks.

As a running example, consider the problem of mail-sorting where a robot has to put envelopes into appropriate bins (see Fig. 1). The newly-inserted envelopes do not present any new clutter, and therefore the domain is really static. This domain is being actively pursued in both industry (see for example (Dorabot 2019)) and in academia (e.g., (Hwang et al. 2015)).

Figure 1: Motivating scenario—a robot (PR2) to perform mail-sorting task in a mailroom environment.

Another well-suited domain for our planner is a manipulator working at a conveyor belt where it has to pick up objects (arbitrarily positioned and oriented) coming on the conveyor (or drop them off onto a conveyor). In such a setting the robot has to deal with one object at a time and other objects do not cause obstructions. As a result, the domain is also static but start/goal configurations may change. An autonomous robot working at a conveyor is also actively being pursued in industry (see again (Dorabot 2019)) and in academia (see (Cowley et al. 2013; Menon, Cohen, and Likhachev 2014)).

Clearly, every time a task is presented to the robot, it can compute a desired path. However, this may incur large online planning times that may be unacceptable in many settings. Alternatively, we could attempt to precompute for each start and goal pair a robot path. However, as the set of possible start and goal locations may be large, caching precomputed paths for all these queries in advance is unmanageable in high-dimensional configuration spaces[1]. Thus, we need to balance memory constraints while providing provable real-time query times.

As we detail in Sec. 2, there has been intensive work for fast online planning (Lehner and Albu-Schaffer 2018)

[*]This research was in part sponsored by ARL, under the Robotics CTA program grant W911NF-10-2-0016.

[1]A robot configuration is a d-dimensional point describing the position of each one of the robot's d joints. The configuration space of a robot is the d-dimensional space of all robot configurations.

and for learning from experience in known environments (Phillips et al. 2012; 2013; Berenson, Abbeel, and Goldberg 2012; Coleman et al. 2015). Similarly, compressing precomputed data structures in the context of motion-planning is a well-studied problem with efficient algorithms (Salzman et al. 2014; Dobson and Bekris 2014). However, to the extent of our knowledge, there is no approach that can *provably guarantee* that a solution will be found to *any* query with bounds on planning time and using a small memory footprint.

In this work, we consider the specific case where the start is fixed. Returning to our running example, this corresponds to having a fixed pickup location above the cart (see Fig. 1). Our key insight is that given any state s, we can efficiently compute a set of states for which a greedy search (to be defined formally in Sec. 3) towards s is collision free. Importantly, the runtime-complexity of such a greedy search is bounded and there is no need to perform computationally-complex collision-detection operations. This insight allows us to generate in an offline phase a small set of so-called "attractor vertices" together with a path between each attractor vertex and the start. In the query phase, a path is generated by performing a greedy search from the goal to an attractor state followed by the precomputed path from the start. We describe our approach in Sec. 3 and analyze it in Sec. 4.

We evaluate our approach in Sec. 5 in simulation on the PR2 robot[2] (see Fig. 1). We demonstrate a speedup of over tenfold in query time when compared to the PRM algorithm with a memory footprint of less than 8 Mb while guaranteeing a maximal query time of less than 3 milliseconds (on our machine).

2 Related work

A straightforward approach to efficiently preprocess a known environment is using the PRM algorithm (Kavraki et al. 1996) which generates a *roadmap*[3]. Once a a dense roadmap has been pre-computed, any query can be efficiently answered online by connecting the start and goal to the roadmap. Query times can be significantly sped up by further preprocessing the roadmaps using landmarks (Paden, Nager, and Frazzoli 2017). Unfortunately, there is no guarantee that a query can be connected to the roadmap as PRM only provides *asymptotic* guarantees (Kavraki, Kolountzakis, and Latombe 1998). Furthermore, this connecting phase requires running a collision-detection algorithm which is typically considered the computational bottleneck in many motion-planning algorithms (LaValle 2006).

Recently, the repetition roadmap (Lehner and Albu-Schaffer 2018) was suggested as a way to extend the PRM for the case of multiple highly-similar scenarios. While this approach exhibits significant speedup in computation time, it still suffers from the previously-mentioned shortcomings.

A complementary approach to aggressively preprocess a given scenario is by minimizing collision-detection time. However this requires designing robot-specific circuitry (Murray et al. 2016) or limiting the approach to standard manipulators (Yang et al. 2018).

An alternative approach to address our problem is to precompute a set of complete paths into a library and given a query, attempt to match complete paths from the library to the new query (Berenson, Abbeel, and Goldberg 2012; Jetchev and Toussaint 2013). Using paths from previous search episodes (also known as using experience) has also been an active line of work (Phillips et al. 2012; 2013; Berenson, Abbeel, and Goldberg 2012; Coleman et al. 2015). Some of these methods have been integrated with sparse motion-planning roadmaps (see e.g., (Salzman et al. 2014; Dobson and Bekris 2014)) to reduce the memory footprint of the algorithm. Unfortunately, none of the mentioned algorithms provide bounded planning-time guarantees that are required by our applications.

Our work bears resemblance to previous work on subgoal graphs (Uras and Koenig 2017; 2018) and to real-time planning (Koenig and Likhachev 2006; Koenig and Sun 2009; Korf 1990). However, in the former, the entire configuration space is preprocessed in order to efficiently answer queries between *any* pair of states which deems it applicable only to low-dimensional spaces (e.g., 2D or 3D). Similarly, in the latter, to provide guarantees on planning time the search only looks at a finite horizon, generating a partial plan, and interleaves planning and execution.

Finally, our notion of attractor states is similar to control-based methods that ensure safe operation over local regions of the free configuration space (Conner, Rizzi, and Choset 2003; Conner, Choset, and Rizzi 2006). These regions are then used within a high-level motion planner to compute collision-free paths.

3 Algorithm Framework

In this section we describe our algorithmic framework. We start (Sec. 3.1) by formally defining our problem and continue (Sec. 3.2) by describing the key idea that enables our approach. We then proceed (Sec. 3.3) to detail our algorithm and conclude with implementation details (Sec. 3.4).

3.1 Problem formulation and assumptions

Let $\mathcal{X}$ be the configuration space of a robot operating in a static environment containing obstacles. We say that a configuration is valid (invalid) if the robot, placed in that configuration does not (does) collide with obstacles, respectively. We are given in advance a start configuration $s_{\text{start}} \in \mathcal{X}$ and some goal region $G \subset \mathcal{X}$. We emphasize that the goal region may contain invalid configurations. In the query phase we are given multiple queries $(s_{\text{start}}, s_{\text{goal}})$ where $s_{\text{goal}} \in G$ is a valid configuration and for each query, we need to compute a collision-free path connecting s_{start} to s_{goal}.

Coming back to our motivating example of mail sorting—the start configuration would be some predefined configuration above the cart where the robot can pick up the envelopes from and the goal region would comprise of all possible placements of the robot's end effector in the cubbies. The environment is static as the only obstacles in the environ-

[2] http://www.willowgarage.com/pages/pr2/overview

[3]A roadmap is a graph embedded in the configuration space where vertices correspond to configurations and edges correspond to paths connecting close-by configurations.

ment are the shelves and the cart which remain stationary in between queries.

We discretize $\mathcal{X}$ into a state lattice $\mathcal{S}$ such that any state $s \in \mathcal{S}$ is connected to a set of successors and predecessors via a mapping Succs/Preds: $\mathcal{S} \rightarrow 2^{\mathcal{S}}$. Define $G_{\mathcal{S}} := \mathcal{S} \cap G$ to be the states that reside in the goal region. Note that although our approach is applicable to general graphs (directed or undirected), to be able to reuse the planned path in the reverse direction (e.g. in our motivating example for the motion from the shelve to the start configuration) the graph needs to be undirected. We make the following assumptions:

A1 $G_{\mathcal{S}}$ is a relatively small subset of S. Namely, it is feasible to exhaustively iterate over all states in $G_{\mathcal{S}}$. However, storing a path from s_{start} to each state in $G_{\mathcal{S}}$ is infeasible.

A2 The planner has access to a heuristic function $h : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$ which can estimate the distance between any two states in $G_{\mathcal{S}}$. Moreover,

- The heuristic function should be *weakly-monotone* with respect to $G_{\mathcal{S}}$, meaning that $\forall s_1, s_2 \in G_{\mathcal{S}}$ where $s_1 \neq s_2$, it holds that,
$$h(s_1, s_2) \geq \min_{s_1' \in \text{Preds}(s_1)} h(s_1', s_2).$$
- The heuristic function h should induce that the goal region is *convex* with respect to h, meaning that $\forall s_1, s_2 \in G_{\mathcal{S}}$ where $s_1 \neq s_2$, it holds that,
$$\arg\min_{s_1' \in \text{Preds}(s_1)} h(s_1', s_2) \in G_{\mathcal{S}}.$$

Namely, for any distinct pair of states (s_1, s_2) in $G_{\mathcal{S}}$, at least one of s_1's predecessors has a heuristic value less than or equal to its heuristic value. Moreover, the predecessor with minimal heuristic value lies in the goal region.

A3 The planner has access to a tie-breaking rule that can be used to define a total order [4] over all states with the same heuristic value.

These assumptions allow us to establish strong theoretical properties regarding the efficiency of our planner. Namely, that within a known bounded time, we can compute a collision-free path from s_{start} to any state in $G_{\mathcal{S}}$.

Assumption **A2** may seem too restrictive (especially convexity), imposing that the goal region cannot be of arbitrary structure. However, after we detail our algorithm (Sec. 3.3) and analyze its theoretical properties (Sec. 4), we sketch how we can relax this assumption to be less restrictive.

3.2 Key idea

Our algorithm relies heavily on the notion of a greedy search. Thus, before we describe of our algorithm, we formally define the terms greedy predecessor and greedy search.

[4]A total order is a binary relation on some set which is anti-symmetric, transitive, and a convex relation.

Definition 1. *Let s be some state and $h(\cdot)$ be some heuristic function. A state $s' \in$ Preds(s) is said to be a* greedy *predecessor of s according to h if it has the minimal h-value among all of s's predecessors.*

Note that if h is weakly monotone with respect to $G_{\mathcal{S}}$ (Assumption **A2**) and we have some tie-breaking rule (Assumption **A3**), then every state has a greedy predecessor in the $G_{\mathcal{S}}$ and it is unique. In the rest of the text, when we use the term greedy predecessor, we assume that it is unique and in $G_{\mathcal{S}}$.

Definition 2. *Given a heuristic function $h(\cdot)$, an algorithm is said to be a* greedy search *with respect to h if for every state it returns its greedy predecessor according to h.*

Note that we define the greedy search in terms of the predecessors and not the successors to account for the directionality of the graph. This will become more clear in Sec. 3.3.

Remark: Now that we have the notion of greedy search, we can better explain our definition of convexity with respect to h (Assumption **A2**); this assumption ensures that a greedy search between any pair of states lies within the goal region, analogously to the standard notion of a convex region where for every pair of points within the region, every point on the straight line segment that joins the pair of points is also within the region.

Our key insight is to precompute in an offline phase subregions within the goal region where a greedy search to a certain ("attractor") state is guaranteed to be collision free and use these subregions in the query phase. Specifically, in the preprocessing phase, $G_{\mathcal{S}}$ is decomposed into a finite set of (possibly overlapping) subregions $\mathcal{R}$. Each subregion $R_i \in \mathcal{R}$ is a hyper-ball defined using a center which we refer to as the "attractor state" $s_i^{\text{attractor}}$ and a radius r_i. These subregions, which may contain invalid states, are constructed in such a way that the following two properties hold

P1 For any valid goal state $s_{\text{goal}} \in R_i \cap G_{\mathcal{S}}$, a greedy search with respect to $h(s, s_i^{\text{attractor}})$ over $\mathcal{S}$ starting at s_{goal} will result in a collision-free path to $s_i^{\text{attractor}}$.

P2 The union of all the subregions completely cover the valid states in $G_{\mathcal{S}}$. Namely, $\forall s \in G_{\mathcal{S}}$ s.t. s is valid, $\exists R \in \mathcal{R}$ *s.t.* $s \in R$.

In the preprocessing stage, we precompute a library of collision-free paths $\mathcal{L}$ which includes a path from s_{start} to each attractor state. In the query phase, given a query s_{goal}, we (i) identify a subregion R_i such that $s_{\text{goal}} \in R_i$ (using the precomputed radii r_i), (ii) run a greedy search towards $s_i^{\text{attractor}}$ by greedily choosing at every point the predecessor that minimizes h and (iii) append this path with the precomputed path in $\mathcal{L}$ to s_{start} to obtain the complete plan. For a visualization of our algorithm, see Fig. 2.

3.3 Algorithm

Preprocessing Phase The preprocessing phase of our algorithm, detailed in Alg. 1, takes as input the start state s_{start}, the goal region $G_{\mathcal{S}}$ and a conventional motion planner $\mathcal{P}$, and outputs a set of subregions $\mathcal{R}$ and the corresponding library of paths $\mathcal{L}$ from $s_{\text{start}}$to each $s_i^{\text{attractor}}$.

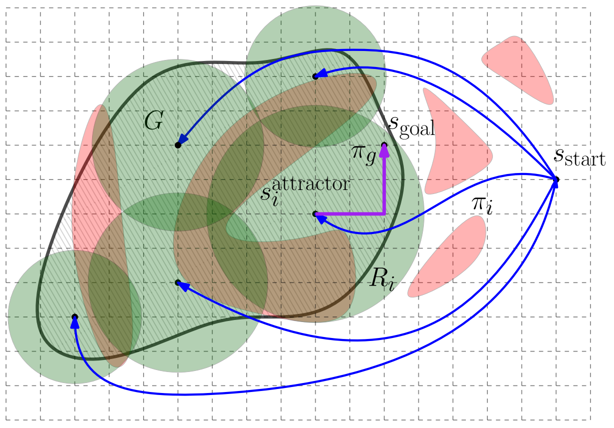


Figure 2: Visualization of approach. Subregions are depicted in green, goal region G is depicted in tiled gray containing obstacles (red). Precomputed paths from s_{start} to attractor states are depicted in blue. Given a query $(s_{\text{start}}, s_{\text{goal}})$, the returned path is given by π_i appended with a greedy path π_g from s_{goal} to $s_i^{\text{attractor}}$ (reversed), which is depicted in purple.

The algorithm covers $G_\mathcal{S}$ by iteratively finding a state s not covered[5] by any subregion and computing a new subregion centered at s. To ensure that $G_\mathcal{S}$ is completely covered (Property **P2**) we maintain a set V of valid (collision free) and a set I of invalid (in collision) states called *frontier states* (lines 3 and 4, respectively). We also construct subregions centered around invalid states $\hat{\mathcal{R}}$ to efficiently store which invalid states have been considered. We start by initializing V with some random state in $G_\mathcal{S}$ and iterate until both V and I are empty, which will ensure that $G_\mathcal{S}$ is indeed covered even if $G_\mathcal{S}$ is not fully connected.

At every iteration, we pop a state from V (line 8), and if there is no subregion covering it, we add it as a new attractor state and compute a path π_i from s_{start} (line 11) using the planner $\mathcal{P}$. We then compute the corresponding subregion (line 12 and Alg. 2).

As we will see shortly, computing a subregion corresponds to a Dijkstra-like search centered at the attractor state. The search terminates with the subregion's radius r_i and a list of frontier states that comprise of the subregion's boundary. The valid and invalid frontier states are then added to V and I, respectively (lines 13 and 14).

Once V gets empty the algorithm starts to search for states which are valid and yet uncovered by growing subregions around invalid states popped from I (lines 16-20 and detailed in Alg. 3). If a valid and uncovered state is found, it is added to V and the algorithm goes back to computing subregions centered at valid states (lines 21-23), otherwise if I also gets empty, the algorithm terminates and it is guaranteed that each valid state contained in $G_\mathcal{S}$ is covered by at least one subregion.

Reachability Search The core of our planner lies in the way we compute the subregions (Alg. 2 and Fig. 3) which

[5]Here, a state s is said to be covered if there exists some subregion $R \in \mathcal{R}$ such that $s \in R$.

Algorithm 1 Goal Region Preprocessing

```
    Inputs: $G_\mathcal{S}$, $s_{\text{start}}$, $\mathcal{P}$    ▷ goal region, start state and a planner
    Outputs: $\mathcal{R}$, $\mathcal{L}$    ▷ subregions and corresponding paths to $s_{\text{start}}$
 1: procedure PREPROCESSREGION($G_\mathcal{S}$)
 2:     $s \leftarrow$ SAMPLEVALIDSTATE($G_\mathcal{S}$)
 3:     $V \leftarrow \{s\}$    ▷ valid frontier states initialized to random state
 4:     $I = \emptyset$    ▷ invalid frontier states
 5:     $i \leftarrow 0 \quad \mathcal{L} = \emptyset \quad \mathcal{R} = \emptyset \quad \hat{\mathcal{R}} = \emptyset$

 6:     while $V$ and $I$ are not empty do
 7:         while $V$ is not empty do
 8:             $s \leftarrow V$.pop()
 9:             if $\nexists R \in \mathcal{R}$ s.t. $s \in R$ then    ▷ $s$ is not covered
10:                 $s_i^{\text{attractor}} \leftarrow s$
11:                 $\pi_i = \mathcal{P}$.PLANPATH($s_{\text{start}}, s_i^{\text{attractor}}$);    $\mathcal{L} \leftarrow \mathcal{L} \cup \{\pi_i\}$
12:                 (OPEN, $r_i$) $\leftarrow$ COMPUTEREACHABILITY($s_i^{\text{attractor}}$)
13:                 insert Valid(OPEN) in $V$
14:                 insert Invalid(OPEN) in $I$
15:                 $R_i \leftarrow (s_i^{\text{attractor}}, r_i)$;    $i \leftarrow i+1 \quad \mathcal{R} \leftarrow \mathcal{R} \cup \{R_i\}$

16:         while $I$ is not empty do
17:             $s \leftarrow I.pop()$
18:             if $\nexists R \in \mathcal{R} \cup \hat{\mathcal{R}}$ s.t. $s \in R$ then    ▷ $s$ is not covered
19:                 $(X, r) \leftarrow$ FINDVALIDUNCOVEREDSTATE($s$)
20:                 $\hat{R} \leftarrow (s, r)$;    $\hat{\mathcal{R}} \leftarrow \hat{\mathcal{R}} \cup \{\hat{R}\}$    ▷ invalid subregion
21:                 if $X$ is not empty then    ▷ valid state found
22:                     insert $X$ in $V$
23:                     break

24:     return $\mathcal{R}, \mathcal{L}$
```

we call a "Reachability Search". The algorithm maintains a set of *reachable* states $S_{\text{reachable}}$ for which Property **P1** holds. As we will see, this will ensure that in the query phase, we can run a greedy search from any reachable state $s \in S_{\text{reachable}}$ and it will terminate in the attractor state. The following recursive definition formally captures the notion of a reachable state.

Definition 3. *Given some attractor state $s_i^{\text{attractor}}$, we say that a state $s \in G_\mathcal{S}$ is reachable under some function $h(\cdot)$ with respect to $s_i^{\text{attractor}}$ if either (i) $s = s_i^{\text{attractor}}$ or (ii) the greedy predecessor of s with respect to $h(s, s_i^{\text{attractor}})$ is reachable.*

The algorithm computes a subregion that covers the maximum number of reachable states that can fit into a hyperball defined by $h(s, s_i^{\text{attractor}})$. The search maintains a priority queue OPEN ordered according to $h(s, s_i^{\text{attractor}})$. Initially, the successors of $s_i^{\text{attractor}}$ are inserted in the OPEN (line 3). For each expanded successor, if its valid greedy predecessor is in $S_{\text{reachable}}$, then the successor is also labeled as reachable (lines 10 and 11).

The algorithm terminates when the search pops a state which is valid but does not have a greedy predecessor state in $S_{\text{reachable}}$ (line 12). Intuitively, this corresponds to the condition when the reachability search exits an obstacle (see Fig. 3). At termination, all the states within the boundary of radius r_i (excluding the boundary) are reachable.

Query Phase Given a query goal state $s_{\text{goal}} \in G_\mathcal{S}$ our algorithm, detailed in Alg. 4, starts by finding a subregion

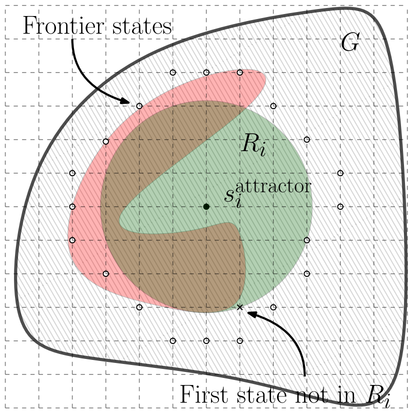


Figure 3: Visualization of Alg 2. Subregion R_i (green) grown from $s_i^{\text{attractor}}$ in a goal region G (tiled grey) containing an obstacle (red). Frontier states and first state not in R_i are depicted by circles and a cross, respectively.

Algorithm 2 Reachability Search

```
1:  procedure COMPUTEREACHABILITY($s_i^{\text{attractor}}$)
2:      $S_{\text{reachable}} \leftarrow \{s_i^{\text{attractor}}\}$                 ▷ reachable set
3:      OPEN ← {Succs($s_i^{\text{attractor}}$)}                 ▷ key: $h(s, s_i^{\text{attractor}})$
4:      CLOSED ← $\emptyset$
5:      $r_i \leftarrow 0$
6:      while OPEN$\neq \emptyset$ do
7:          $s \leftarrow$ OPEN.pop()
8:          insert $s$ in CLOSED
9:          $s'_g \leftarrow \arg\min_{s' \in \text{Preds}(s)} h(s', s_i^{\text{attractor}})$     ▷ greedy predecessor
10:         if $s'_g \in S_{\text{reachable}}$ and Valid(edge(s,$s'_g$)) then
11:             $S_{\text{reachable}} \leftarrow S_{\text{reachable}} \cup \{s\}$          ▷ $s$ is greedy
12:         else if Valid($s$) then
13:             $r_i \leftarrow h(s, s_i^{\text{attractor}})$
14:             return (OPEN, $r_i$)
15:         for each $s' \in$ Succs($s$) $\cap\, G_{\mathcal{S}}$ do
16:             if $s' \notin$ CLOSED then
17:                 insert $s'$ in OPEN with priority $h(s', s_i^{\text{attractor}})$
18:     $r_i \leftarrow h(s, s_i^{\text{attractor}}) + \epsilon$          ▷ $\epsilon$ is a small positive constant
19:     return (OPEN, $r_i$)
```

$R_i \in \mathcal{R}$ which covers it (line. 9). Namely, a subregion R_i for which $h(s_{\text{goal}}, s_i^{\text{attractor}}) < r_i$. We then run a greedy search starting from s_{goal} by iteratively finding for each state s the predecessor with the minimum heuristic $h(s, s_i^{\text{attractor}})$ value until the search reaches $s_i^{\text{attractor}}$ (lines 10 and 1- 6). The greedy path π_g is then appended to the corresponding precomputed path $\pi_i \in \mathcal{L}$ (line 11). Note that at no point do we need to perform collision checking in the query phase (given the fact that the environment is static).

3.4 Implementation details

Ordering subregions for faster queries Recall that in the query phase we iterate over all subregions to find one that covers s_{goal}. In the worst case we will have to go over all subregions. However, our algorithm typically covers most of the goal region $G_{\mathcal{S}}$ using a few very large subregions (namely, with a large radii r_i) and the rest of $G_{\mathcal{S}}$ is covered by a number of very small subregions.

Algorithm 3 Find valid uncovered state

```
1:  procedure FINDVALIDUNCOVEREDSTATE($\hat{s}$)
2:      OPEN ← $\{\hat{s}\}$
3:      while OPEN$\neq \emptyset$ do
4:          $s \leftarrow$ OPEN.pop()
5:          insert $s$ in CLOSED
6:          if $\nexists R \in \mathcal{R}$ s.t. $s \in R$ and Valid(s) then
7:              return ($\{s\}$, $h(s, \hat{s})$)
8:          for each $s' \in$ {Succs($s$) ∪ Preds($s$)} $\cap\, G_{\mathcal{S}}$ do
9:              if $s' \notin$ CLOSED then
10:                 insert $s'$ in OPEN with priority $h(s', s_i^{\text{attractor}})$
11:     $r = h(s, \hat{s})) + \epsilon$          ▷ $\epsilon$ is a small positive constant
12:     return ($\emptyset$, $r$))
```

Algorithm 4 Query

```
1:  procedure FINDGREEDYPATH($s_1, s_2$)
2:      $s_{\text{curr}} \leftarrow s_2$;   $\pi \leftarrow \emptyset$
3:      while $s_{\text{curr}} \neq s_1$ do
4:          $\pi \leftarrow \pi \cdot s_{\text{curr}}$                ▷ append current state to path
5:          $s_{\text{curr}} \leftarrow \arg\min_{s \in \text{Preds}(s_{\text{curr}})} h(s, s_1)$     ▷ greedy predecessor
6:      return REVERSE ($\pi$)    ▷ reverse to get a path from $s_1$ to $s_2$

7:  procedure COMPUTE PATH($s_{\text{goal}}$)
8:      for each $R_i \in \mathcal{R}$ do
9:          if $h(s_{\text{goal}}, s_i^{\text{attractor}}) < r_i$ then          ▷ $R_i$ covers $s_{\text{goal}}$
10:             $\pi_g \leftarrow$ FINDGREEDYPATH ($s_i^{\text{attractor}}$, $s_{\text{goal}}$)
11:             return $\pi_i \cdot \pi_g$               ▷ append $\pi_g$ to $\pi_i \in \mathcal{L}$
```

Thus, if we order our subregions (offline) according to their corresponding radii, there is a higher chance of finding a covering subregion faster. While this optimization does not change our worst-case analysis (Sec. 4), it speeds up the query time in case the number of subregions is very large.

Efficiently constructing $\mathcal{L}$ Constructing paths to the attractor states (Alg. 1, line 11) can be done using any motion planning algorithm $\mathcal{P}$. In our implementation we chose to use RRT-Connect (Kuffner and LaValle 2000). Interestingly, this step dominates the running time of the preprocessing step. To improve the preprocessing time, we initially set a small timeout for RRT-Connect to compute a path from an attractor state s_{start} to $s_i^{\text{attractor}}$. Attractor states for which RRT-Connect fails to find a path to $s_i^{\text{attractor}}$ are marked as *bad* attractors and we do not grow the subregions from them. These, so-called *bad* attractors are discarded when other subregions cover them.

When Alg. 1 terminates, we reload the valid list V with the remaining bad attractors and rerun Alg. 1 but this time with a large timeout for RRT-Connect. We can also increase the timeout with smaller increments and run Alg. 1 iteratively until there are no more bad attractors (assuming that there exists a solution for each goal state $\in G_{\mathcal{S}}$).

Pruning redundant subregions To reduce the number of precomputed subregions we remove redundant ones after the Alg. 1 terminates. In order to do that, we iterate through all the subregions and remove the ones which are fully contained within any other subregion. This step reduces both the query complexity (see Sec. 4) as well as the memory consumption.

4 Analysis

In this section we formally prove that our algorithm is correct (Sec. 4.1) and analyze its computational complexity, completeness and bound on solution quality (Sec. 4.2, 4.3 and 4.4 respectively).

4.1 Correctness

To prove that our algorithm is correct, we show that indeed all states of every subregion are reachable and we can identify if a state belongs to a subregion using its associated radius. Furthermore, we show that a path obtained by a greedy search within any subregion is valid and that all states in $G_{\mathcal{S}}$ are covered by some subregion. These notions are captured by the following set of lemmas.

Lemma 1. *Let $S_{reachable}$ be the set of states computed by Alg. 2 for some attractor vertex $s_i^{attractor}$. Every state $s \in S_{reachable}$ is reachable with respect to $s_i^{attractor}$.*

Proof. The proof is constructed by an induction over the states added to $S_{\text{reachable}}$. The base of the induction is trivial as the first state added to $S_{\text{reachable}}$ is $s_i^{\text{attractor}}$ (line 2) which, by definition, is reachable with respect to $s_i^{\text{attractor}}$. A state s is added to $S_{\text{reachable}}$ only if its greedy predecessor is in $S_{\text{reachable}}$ (line 9) which by the induction hypothesis is reachable with respect to $s_i^{\text{attractor}}$. This implies by definition that s is reachable with respect to $s_i^{\text{attractor}}$. Note that this argument is true because the greedy predecessor of every state is unique (Assumption **A3**). □

Lemma 2. *Let $S_{reachable}$ be the set of states computed by Alg. 2 for some attractor vertex $s_i^{attractor}$. A state s is in $S_{reachable}$ iff $h(s, s_i^{attractor}) < r_i$.*

Proof. Alg. 2 orders the nodes to be inserted to $S_{\text{reachable}}$ according to $h(s, s_i^{\text{attractor}})$ (line 3). As our heuristic function is weakly monotonic (Assumption **A2**), the value of r_i monotonically increases as the algorithm adds states to $S_{\text{reachable}}$ (line 13). Thus, for every state $s \in S_{\text{reachable}}$, we have that $h(s, s_i^{\text{attractor}}) < r_i$.

For the opposite direction, assume that there exists a state $s \notin S_{\text{reachable}}$ such that $h(s, s_i^{\text{attractor}}) < r_i$. This may be because Alg. 2 terminated due to a node s' that was popped from the open list with $h(s', s_i^{\text{attractor}}) \leq h(s, s_i^{\text{attractor}})$. However, using the fact that our heuristic function is weakly monotonic (Assumption **A2**) we get a contradiction to the fact that $h(s, s_i^{\text{attractor}}) < r_i$. Alternatively, this may be because we prune away states that are in $G_{\mathcal{S}}$ (Alg. 2 line 15). However, using the fact that our goal region is convex with respect to h (Assumption **A2**), this cannot hold. □

Lemma 3. *Let $R_i \in \mathcal{R}$ be a subregion computed by Alg. 2. for some attractor vertex $s_i^{attractor}$. A greedy search with respect to $h(s, s_i^{attractor})$ starting from any valid state $s \in R_i$ is complete and valid.*

Proof. Given a state $s \in R_i$, we know that $s \in S_{\text{reachable}}$ (Lemma 2) and that it is reachable with respect to $s_i^{\text{attractor}}$ (Lemma 1). It is easy to show (by induction) that any greedy search starting at a state $S_{\text{reachable}}$ will only output states in $S_{\text{reachable}}$. Furthermore, a state is added to $S_{\text{reachable}}$ only if the edge connecting to its greedy predecessor is valid (line 10). Thus, if $s \in S_{\text{reachable}}$ is valid, the greedy search with respect to $h(s, s_i^{\text{attractor}})$ starting from s is complete and valid. □

Lemma 4. *At the end of Alg. 1, every state $s \in G_{\mathcal{S}}$ is covered by some subregion $R \in \mathcal{R}$.*

Proof. Assume that this does not hold and let $s \in G_{\mathcal{S}}$ be a state that is not covered by any subregion but has a neighbor (valid or invalid) that is covered. If s is valid, then it would have been in the valid frontier states V and either been picked to be an attractor state (line 10) or covered by an existing subregion (Alg. 2). A similar argument holds if s is not valid. □

From the above we can immediately deduce the following corollary:

Corollary 1. *After preprocessing the goal region $G_{\mathcal{S}}$ (Alg. 1 and 2), in the query phase we can compute a valid path for any valid state $s \in G_{\mathcal{S}}$ using Alg. 4.*

Remark We can relax Assumption **A2** in two ways. The first is by explicitly tracking states not in the goal region instead of pruning them away (Alg. 2 line 15). Unfortunately, this may require the algorithm to store many states not in $G_{\mathcal{S}}$ which may be impractical (recall that Assumption **A1** only states that we can exhaustively store in memory the states in the goal region). An alternative, more practical way, to relax Assumption **A2** is by terminating the search when we encounter a state not in the goal region. This may cause the algorithm to generate much more subregions (with smaller radii) which may increase the memory footprint and the preprocessing times.

4.2 Time Complexity of Query Phase

The query time comprises of (i) finding the containing subregion R_i and (ii) running the greedy search to $s_i^{\text{attractor}}$. Step (i) requires iterating over all subregions (in the worst case) which takes $O(|\mathcal{R}|)$ steps while step (ii) requires expanding the states along the path from s_{goal} to $s_i^{\text{attractor}}$ which requires $O(\mathcal{D})$ expansions where $\mathcal{D}$ is the depth (maximal number of expansions of a greedy search) of the deepest subregion. For each expansion we need to find the greedy predecessor, considering at most b predecessors, where b is the maximal branching factor of our graph. We can measure the depth of each subregion in Alg. 2 by keeping track of the depth of each expanded state from the root i.e., $s_i^{\text{attractor}}$. Hence, overall the query phase takes $O(|\mathcal{R}| + \mathcal{D} \cdot b)$ operations. The maximal query time can also be empirically profiled after the preprocessing phase.

Remark: Note that we can also bound the number of expansions required for the query phase by bounding the maximum depth of the subregions. We can do that by terminating Alg. 2 when the R_i reaches the maximum depth or if the existing termination condition (line 12) is satisfied. Having said that, this may come at the price of increasing the number of subregions.

4.3 Algorithm Completeness

Our method generates plans by stitching together a path from the library $\mathcal{L}$ (computed offline using some planner $\mathcal{P}$), and a path computed online which is returned by the greedy search on a discretized graph $G_{\mathcal{S}}$. For the former, our method simply inherits the completeness properties of the planner $\mathcal{P}$, whereas for the latter, our method is resolution complete; it follows from the correctness discussion (Section 4.1).

4.4 Bound on Solution Quality

Let the function $c(\cdot)$ denote the cost of a path and $c^*(\cdot)$ denote the cost of an optimal path. Assuming that we precompute each path $\pi_i \in \mathcal{L}$ with the optimal cost $c^*(\pi_i)$, from the triangle inequality it can be trivially shown that the quality of complete path π computed by our method has an additive suboptimality bound; i.e., $c(\pi) - c^*(\pi) < 2 * c(\pi_g)$, where $c(\pi_g)$ is the cost of the greedy path from the attractor to the goal state.

5 Evaluation

We evaluated our algorithm on the PR2 robot for a single-arm (7-DOF) motion-planning problem. The illustrated task here is to pick up envelopes from a cart and put them in the cubby shelves (see Fig. 1). Such settings are common in mailroom environments where the robot may have to encounter the same scenario over and over again. The start state is a fixed state corresponding to the pickup location, whereas the task-relevant goal region G is specified by bounding the position and orientation of the end effector. For this domain, we define G as a bounding box covering all possible positions and orientations of the end effector within the cubby shelf.

The search is done on an implicit graph $G_{\mathcal{S}}$ constructed using *motion primitives* which are small kinematically feasible motions. We define the primitives in the task-space respresentation as small motions for the robot end effector in position axes (*x, y, z*) and orientation axes of Euler angles (*roll, pitch, yaw*), and a joint-angle motion for the redundant joint of the 7-DOF arm. The heurstic function is the Euclidean distance in these seven dimensions. The discretization we use for the graph $G_{\mathcal{S}}$ is 2 cm for the position axes, 10 degrees for the Euler axes and 5 degrees for the redundant joint. For this domain, we keep the *pitch* and *roll* of the end effector fixed and allow motion primitives along the remaining five dimensions i.e. *x, y, z, yaw* and the reduntant joint angle. We also limit the *yaw* to be between -30 and 30 degrees. Note that the specification of the $G_{\mathcal{S}}$ is purely task specific and we exploit the task constraints to limit the size of $G_{\mathcal{S}}$ which makes our preprocessing step tractable (Assumption **A1**).

We compared our approach with different single- and multi-query planners in terms of planning times, success rates and memory consumption (see Table 1) for 200 uniformly sampled goal states from G. Among the multi-query planners, we implemented PRM, a multi-query version of RRT which we name MQ-RRT and the E-graph planner. For MQ-RRT, we precompute an RRT tree rooted at s_{start} offline (similar to PRM) and query it by trying to connect s_{goal} to the nearest nodes of the precomputed tree. We use the same connection strategy for MQ-RRT as the one that the asymptotically-optimal version of PRM uses[6]. For PRM, we precomputed the paths from all the nodes in G to s_{start} (this is analogous to our library $\mathcal{L}$). The query stage thus only required the connect operation (i.e. attempting to connect to k nearest neighbors of s_{goal}). For both of these planners we also added a goal-region bias by directly sampling from G, five percent of the time[7].

For single-query planning, we only report the results for RRT-Connect as it has the fastest run times from our experience. For PRM and MQ-RRT, if the connect operation fails for a query, we considered that case as a failure. For RRT-Connect, we set a timeout of 10 seconds.

For our method, preprocessing (Alg. 1) took 1,445 seconds and returned 1,390 subregions. For precomputing paths (Alg. 1, line 11), we use RRT-Connect. For the first run of Alg. 1, we set the timeout to be 10 seconds and for the second run (after reloading V with the bad attractors), we set the timeout to be 60 seconds. By doing so the algorithm finishes successfully with having no remaining bad attractors.

Table 1 shows the numerical results of our experiments. Our method being provably guaranteed to find a plan in bounded time, shows a success rate of 100 percent. For the two preprocessing-based planners PRM and MQ-RRT, we report the results for a preprocessing time of 4T (T being the time consumed by our method in preprocessing). The E-graph planner is bootstrapped with a hundred paths precomputed for uniformly sampled goals in $G_{\mathcal{S}}$. For each of these three multi-query planners while running the experiments, the newly-created edges were appended to the auxilary data (tree, roadmap or E-graph) to be used for the subsequent queries.

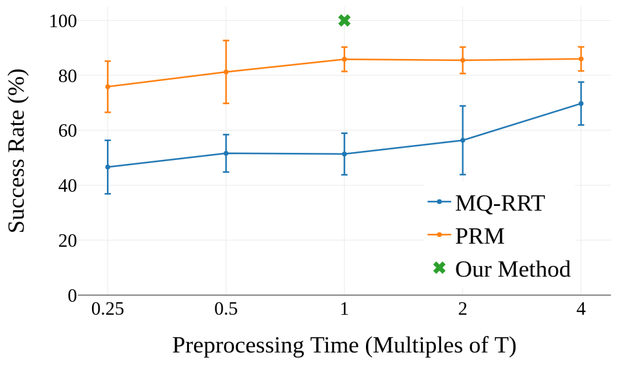
Among other planners, PRM shows the highest success rate but at the cost of a large memory footprint. The E-graph planner has a small memory footprint but it shows significantly longer planning times. RRT-Connect being a single-query planner happens to be the slowest. Our method shows a speedup of over tenfold in query time as compared to PRM and MQ-RRT and about three orders of magnitude speedup over the E-graph planner and RRT-Connect. The plots in Fig. 4 show how the success rate and the memory footprints of PRM and MQ-RRT vary as a function of the preprocessing time. For our domain the PRM seems to saturate in terms of the success rate after T time whereas MQ-RRT continues to improve provided more preprocess-

[6]In order for the quality of paths obtained the PRM to converge to the quality of the optimal solution, a query should be connected to its k nearest neighbors where $k = e(1 + 1/d)\log(n)$. Here n is the number of nodes in the tree/roadmap and d is the dimensionality of the configuration space (Karaman and Frazzoli 2011; Solovey, Salzman, and Halperin 2016).

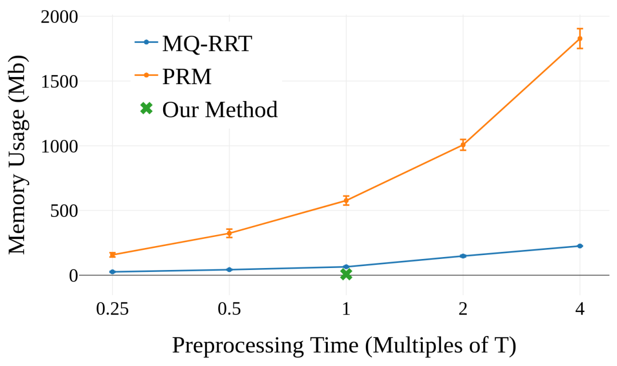
[7]We used OMPL (Şucan, Moll, and Kavraki 2012) for comparisons with the sampling-based planners and modified the implementations as per needed.

	PRM (4T)	MQ-RRT (4T)	E-graph	RRT-Connect	**Our method**
Planning time [ms]	21.7 (59.6)	21.2 (35.5)	497.8 (9678.5)	1960 (9652)	**1.0 (1.6)**
Success rate [%]	86	69.75	76.5	83.8	**100**
Memory usage [Mb]	1,828	225.75	**2.0**	-	7.8

Table 1: Experimental results comparing our method with other single- and multi-query planners tested on Intel® Core i7-5600U (2.6GHz) machine with 16GB RAM. The table shows the mean/worst-case planning times, success rates and memory usage for our method and for other multi-query planners preprocessed with quadruple the time that our method takes in precomputation (T = 1,445 seconds). Note that the worst-case time for our method shown in these results (∼1.6 millisecond) is the empirical one and not the computed provable time bound which is 3 milliseconds (on our machine) for this environment. Results of sampling-based planners are averaged over 200 uniformly sampled queries. For the sampling-based planners the results were averaged over 4 trials (for the same set of 200 queries) with different random number generation seeds.



(a)



(b)

Figure 4: Preprocessing time vs (a) the success rates and (b) memory useage for the 200 queries averaged over 4 trials with different random number generation seeds for the PRM and the MQ-RRT algorithms. The results were computed at the intervals which are multiple of the time T = 1,455 seconds, that our method takes for precomputation. The green cross shows our method for reference.

ing time. In terms of memory usage, PRM's memory footprint grows more rapidly than MQ-RRT.

6 Conclusion and future work

We proposed a preprocessing-based motion planning algorithm that provides provable real-time performance guarantees for repetitive tasks and showed simulated results on a PR2 robot. Key questions that remain open for future work regard providing stronger guarantees on the solution quality and bounding the number of covering subregions. We conjuncture that computing the minimal number of subregions is an NP-Hard problem and, if this is the case, we can possibly seek to compute a set of subregions whose size is within some constant-factor approximation of the size of the optimal set. Finally, we plan to extend our approach to semi-static environments where there exists both static and non-static obstacles. In such settings, we can also use our planner as an initial pass, having a conventional planner as a backup. We can preprocess our planner with only static obstacles and in the query time we can do a validity check of the computed path; if the path intersects any non-static obstacle, we can fall back to the conventional planner. This can immensely increase the overall throughput of a system if the environment is largely static.

References

Berenson, D.; Abbeel, P.; and Goldberg, K. 2012. A robot path planning framework that learns from experience. In *IEEE International Conference on Robotics and Automation (ICRA)*, 3671–3678.

Coleman, D.; Şucan, I. A.; Moll, M.; Okada, K.; and Correll, N. 2015. Experience-based planning with sparse roadmap spanners. In *IEEE International Conference on Robotics and Automation (ICRA)*, 900–905.

Conner, D. C.; Choset, H.; and Rizzi, A. 2006. Integrated planning and control for convex-bodied nonholonomic systems using local feedback control policies. In *Robotics: Science and Systems (RSS)*.

Conner, D. C.; Rizzi, A.; and Choset, H. 2003. Composition of local potential functions for global robot control and navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 4, 3546–3551.

Cowley, A.; Cohen, B.; Marshall, W.; Taylor, C. J.; and Likhachev, M. 2013. Perception and motion planning for pick-and-place of dynamic objects. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 816–823. IEEE.

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

Dorabot. 2019. Dorabot pick and place systems. https://www.dorabot.com/solutions/robotics.

Hwang, V.; Phillips, M.; Srinivasa, S.; and Likhachev, M. 2015. Lazy validation of experience graphs. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 912–919. IEEE.

Jetchev, N., and Toussaint, M. 2013. Fast motion planning from experience: trajectory prediction for speeding up movement generation. *Autonomous Robots* 34(1-2):111–127.

Karaman, S., and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research* 30(7):846–894.

Kavraki, L. E.; Švestka, 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.

Kavraki, L. E.; Kolountzakis, M. N.; and Latombe, J. 1998. Analysis of probabilistic roadmaps for path planning. *IEEE Trans. Robotics and Automation* 14(1):166–171.

Koenig, S., and Likhachev, M. 2006. Real-time adaptive A*. In *International joint conference on Autonomous agents and multi-agent systems*, 281–288. ACM.

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.

Korf, R. E. 1990. Real-time heuristic search. *Artificial Intelligence* 42(2-3):189–211.

Kuffner, J. J., and LaValle, S. M. 2000. RRT-connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 995–1001.

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

Lehner, P., and Albu-Schaffer, A. 2018. The repetition roadmap for repetitive constrained motion planning. *IEEE Robotics and Automation Letters*. to appear.

Menon, A.; Cohen, B.; and Likhachev, M. 2014. Motion planning for smooth pickup of moving objects. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 453–460. IEEE.

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

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

Phillips, M.; Cohen, B. J.; Chitta, S.; and Likhachev, M. 2012. E-graphs: Bootstrapping planning with experience graphs. In *Robotics: Science and Systems (RSS)*.

Phillips, M.; Dornbush, A.; Chitta, S.; and Likhachev, M. 2013. Anytime incremental planning with E-graphs. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2444–2451.

Salzman, O.; Shaharabani, D.; Agarwal, P. K.; and Halperin, D. 2014. Sparsification of motion-planning roadmaps by edge contraction. *IJRR* 33(14):1711–1725.

Solovey, K.; Salzman, O.; and Halperin, D. 2016. New perspective on sampling-based motion planning via random geometric graphs. In *Robotics: Science and Systems (RSS)*.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The open motion planning library. *Robotics & Automation Magazine* 19(4):72–82.

Uras, T., and Koenig, S. 2017. Feasibility study: Subgoal graphs on state lattices. In *Symposium on Combinatorial Search, SOCS*, 100–108.

Uras, T., and Koenig, S. 2018. Fast near-optimal path planning on state lattices with subgoal graphs. In *Symposium on Combinatorial Search, SOCS*, 106–114.

Yang, Y.; Merkt, W.; Ivan, V.; Li, Z.; and Vijayakumar, S. 2018. HDRM: A resolution complete dynamic roadmap for real-time motion planning in complex scenes. *IEEE Robotics and Automation Letters* 3(1):551–558.