Dartmouth College
Dartmouth Digital Commons

Dartmouth College Undergraduate Theses | Theses and Dissertations

5-26-2020

Regression-based motion planning

Josiah K. Putman
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

 Part of the Computer Sciences Commons

Recommended Citation

Putman, Josiah K., "Regression-based motion planning" (2020). *Dartmouth College Undergraduate Theses*. 150.
https://digitalcommons.dartmouth.edu/senior_theses/150

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.

Dartmouth College

Senior Honors Thesis

Dartmouth Computer Science Technical Report TR2020-882

Regression-based motion planning

Author:
Josiah Putman

Advisor:
Devin Balkcom

May 26, 2020

Contents

Abstract

This thesis explores two novel approaches to sample-based motion planning that utilize regressions as continuous function approximations to reduce the memory cost of planning. The first approach, Field Search Trees (FST) provides a solution for single-start planning by iteratively building local regressions of the cost-to-arrive function. The second approach, the Regression Complex (RC), constructs a complex of cells with each cell containing a regression of the distance between any two points on its boundary, creating a useful data structure for any start and goal planning query. We provide formal definitions of both approaches and experimental results of running the algorithms on different simulated robot systems. We conclude that regression-based motion planning provides key advantages over traditional sample-based motion planning in certain cases, but more work is required to extend these approaches into higher dimensional configuration spaces.

1 Introduction

Sample-based motion planners such as PRM* and RRT* provide asymptotically optimal paths, but at a significant cost of time and memory. To approach the optimal solution, sample-based algorithms must place samples in a tube around the optimal path. Because the location of the optimal path is unknown, such algorithms must drastically increase their sampling density over the entire search space to achieve asymptotic optimality. Although much work has been done to improve these algorithms by reducing redundant sampling and collision detection ([27], [4]), even the most sophisticated approaches still rely on discrete graphs for representing the intrinsically continuous optimal motion of a system.

We explore replacing discrete graphs with continuous function approximations for general purpose motion planning and provide algorithms for searching among these regression-based representations. We begin by proposing two approaches to decomposing and regressing the configuration space into searchable regions, the Field Search Trees (FST*) and the Regression Complex (RC). The FST* is targeted towards single-goal motion planning, and is formally presented in section 3. The RC targets motion planning for any start and goal query, and is presented in section 4. Both approaches are regression-agnostic, meaning that any regression technique can be used as a

module for these approaches. To provide a survey of various regression techniques and their performance in the context of regressing distance metrics, we provide a comparative study in Appendix A. All algorithms and regression methods have open-source Julia implementations through our custom library, `MotionPlanning.jl`, which is discussed in detail in Appendix C.

The present work demonstrates several advantages of working with continuous representations of metric spaces. Continuous representations, which may be more computationally expensive to create than discrete graph structures like roadmaps, generally have smaller memory footprints and provide more reliably optimal solutions. With the rise of distributed computing in the cloud increasing computational capacity, memory often becomes the bottleneck in motion planning. Large discrete data structures can be constructed using offline preprocessing, but to make the information available to robots locally requires storage or transmission across a network. Distributed computing also benefits greatly from parallelization, which divide-and-conquer approaches like the RC lend themselves towards.

1.1 Notation

For our discussion and analysis of FST* and RC, we use the following notation for describing motion planning problems. Let $Q \in \mathbb{R}^D$ denote the D-dimensional configuration space of the robot. Let $Q_f \subseteq Q$ denote the free regions of Q, and $\Delta : Q_f^2 \mapsto \{0, 1\}$ be a local planner that determines if two states $q_1, q_2 \in Q_f$ are connectable. If the states are connectable, we assume that the local planner also provides the cost of the optimal path between q_1 and q_2. Let $d : Q^2 \mapsto \mathbb{R}_{\geq 0}$ denote the distance metric between pairs of states. FST* and RC require such a local planner and distance metric, and treats these modules as a black box.

The runtime and memory cost of both the FST* and RC algorithms can be significantly improved via calculating *certificates* in the configuration space. This involves taking advantage of lower bounds on a given configuration's distance to the nearest obstacle to reduce the number of local planner calls. We use the definition of certificate used by [4], which specifies that the local planner is also able to provide this lower bound, denoted by $d_{\min}$. We use d_{true} to denote the exact minimum distance to the nearest obstacle. Certificates are defined on a problem-by-problem basis, and we formally describe the certificate methods used for each problem in subsection 3.3.

2 Related work

Both the FST* and RC follow the same structure as well known sample-based approaches [17] by decomposing the planning problem into the *learning* (or construction) phase and the *query* phase. During the learning phase, the algorithm samples the configuration space, checks for collisions, and queries the local planner to construct a data structure that can be used during the query phase. In the case of single-goal planning, this data structure is constructed assuming that any query will lead to the same goal (as with RRT* [16]), whereas general purpose planning is goal agnostic (as with PRM* [16]).

Although both PRM* and RRT* have been shown to be asymptotically optimal, they fail to take advantage of large free regions in the configuration space and require constructing immensely redundant discrete graphs. Karaman shows [16] in figure 11 (a) of his evaluation of PRM* in a planar without obstacles that the PRM* needs over 50,000 samples to get within 1% of the optimal path. This is to be expected of asymptotically optimal sample-based planners, since path improvements require the sampler to randomly choose configurations close to the optimal path. Similar issues exist for the RRT*. The Informed RRT* [13] provides a slight improvement on sampling accuracy by focusing samples on the hyper-ellipsoid around the optimal path, but the improvement is marginal and the resulting structure still relies on an explicit discrete graph. Approaches that remove or avoid placing vertices altogether, such as the Visibility PRM [27] and certificate methods [4, 9], must sacrifice optimality. Similarly, trees or graphs of controllers (e.g. LQR trees [28]) may cover large regions of space nicely, but are focused on robust control rather than optimal path cost.

Work on graph spanners [20, 18, 29] has shown that many of the edges in such graphs can be deleted without too much harm to the path quality. Prior work in our lab on metric cells [1] provides formal upper bounds on the complexity of approximating optimal trajectories through cell decomposition, but have high memory costs.

Towards finding effective continuous representation of the configuration space, a modification of the D* algorithm called Field D* [12] explores using linear interpolation of the cost function in cells for 2D motion planning. Our FST* algorithm pairs this idea of linear approximations with a tree-based construction procedure based on the Fast Marching Trees (FMT*) algorithm [15], which utilizes a dynamic programming approach to grow a tree of paths outwards in cost-to-arrive space.

Beyond linear approximations, more sophisticated smooth approximations of value functions are being increasingly applied to motion planning. In [7], a metric for swept volume is learned, and serves as an effective and admissible heuristic for planning. Work by Rayner, Bowling, and Sturtevant [24] remaps a motion problem with obstacles in such a way that Euclidean distance in the warped map serves as a heuristic for the original problem. Recent work by Faust *et al.* [11] explores a combined sampling and reinforcement-learning approach to long-range planning, addressing some of the same computational issues present in FST* and RC, but with less focus on optimality. Network embeddings also attempt to find a function that expresses distance between vertices in a network; [8] provides a recent survey.

Neural networks have been used to learn value functions derived from optimal control; one of the earliest examples being Munos's work using neural networks to approximate solutions of the HJB equation [21]. Distance metric approximation for RRTs using supervised learning [3] has been used for tasks such as pendulum swing-up problems, but is limited to one-shot planning.

3 Field Search Trees

To begin our exploration of regression-based motion planning, let us simplify the planning problem to have only one specified starting point. For this constrained problem, we no longer need to serve any pair of configurations as queries, which lets us build a much smaller data structure that avoids many all-pairs. Existing algorithms such as RRT* [16] and FMT* [15] make similar assumptions. For the sake of our analysis, let us denote this specific starting point as $q_s \in Q_f$.

3.1 A simple example

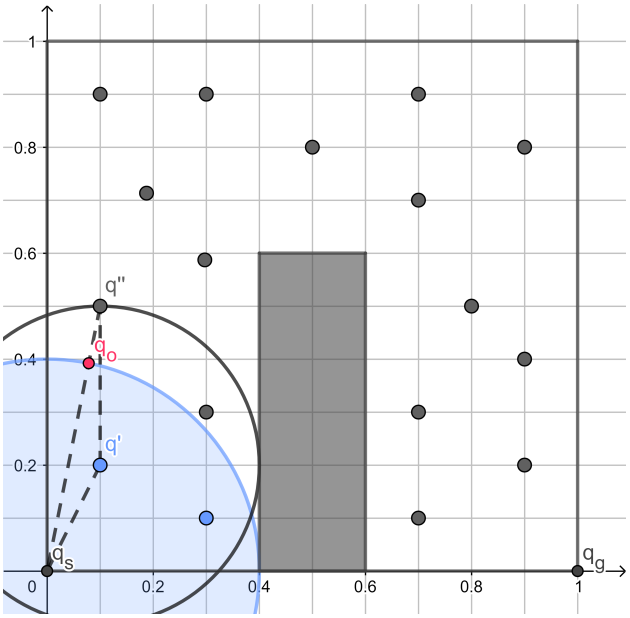


Figure 1: FST* field construction.

Consider a point robot navigating amongst polygonal obstacles shown in Figure 1. Our object is to find a path from the origin $q_s = (0, 0)$ to the goal $q_g = (1, 0)$, but we want to avoid relying on a discrete graph in the way RRT* and PRM* do. Namely, we want to iteratively build a continuous representation of how far any given point is from the start, and then use this representation to find our final path.

Approximating the cost-to-arrive function is easy near the start. Take all nearby points in the neighborhood of q_s (within a certain ball radius, shown in blue in Figure 1), calculate their Euclidean distance to q_s, and regress this function based on those data points. For simplicity, we use a linear regression. We now have a first-order approximation of the cost-to-arrive function around the start.

Next, we want to build this regression outwards, using q_s's regression as a reference point. Pick a point inside of the neighborhood of q_s, and call it q'. To build a regression in the neighborhood of q', we must first calculate the cost-to-arrive of all points in its neighborhood that weren't already calculated during the regression of q_s. Select one such point q'', which is in the neighborhood of q' but not q_s. Outside of q_s's neighborhood, we are not guaranteed that configurations are directly connectable to q_s. Instead, we must find the optimal point in q_s's field that minimizes q''s cost-to-arrive but is still connectable. This optimal point is shown by q_o in red in Figure 1. When we find this optimal point, we know the approximate cost-to-arrive of q''. Without this operation, we would have to rely on the the distance from q'' to q' and the distance form q' to q_s, which is clearly greater than the distance obtained by passing through q_o. Repeatedly applying this procedure to all neighbors of q', we can regress the cost-to-arrive around q'.

This step of taking a configuration with a regression of its neighborhood's cost-to-arrive and building regressions for each of the points in its neighborhood is the core of the FST* algorithm. For concision, we call the regressed neighborhood of a configuration q the *field* of q.

Figure 2 shows the plot of the FST*'s approximation for cost-to-arrive, along with a path extracted from the sequence of fields used to reach the goal. The light grey lines represent the structure of the search tree.

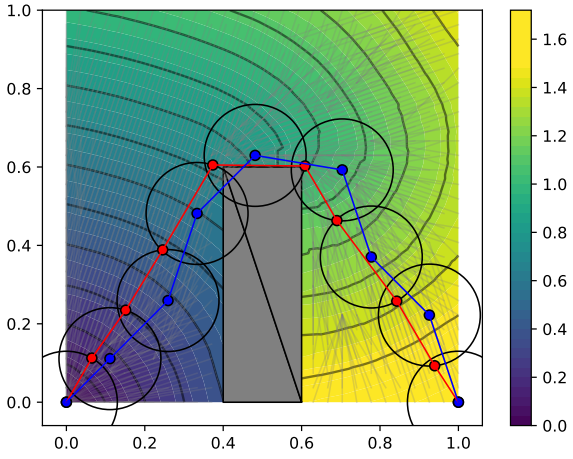


Figure 2: FST* path.

3.2 FST* algorithm

The above example provides a flavor of how FST* builds the distance function approximation using previous regressions as a reference point. In this section, we formally describe the full FST* algorithm utilizing this iterative regression. See algorithm 1 for the complete pseudo code.

FST* functions as a hybrid between FMT*'s tree-growth algorithm and Field D*'s use of linear regressions to generate fields for each sample for smoothing the resulting path. Similar to FMT*, FST* grows its tree outwards from q_s using a priority queue in cost-to-arrive order, which allows the algorithm to take a single pass through the configuration space and never needs to update previous values. FST* departs from FMT* in two significant ways: firstly, it uses certificates to determine if points are neighbors and reduce local planner calls; secondly, instead of simply generating points, it generates local regressions of the distance to goal.

The core idea is to build a regression of cost-to-arrive function outwards from the q_s. The immediate regression around q_s is trivial: take all neighbors of q_s and directly calculate their distances. Using this distance data as training data, we can regress a field about q_s to approximate the cost-to-arrive.

For the rest of the points, we rely on the regression fields built by previously calculated points to estimate the cost-to-arrive. To this end, two

mappings are maintained: $\mathcal{D} : Q_f \mapsto \mathbb{R}$ which stores each point's cost-to-arrive, and $\mathcal{R} : Q_f \mapsto R$ where R denotes an arbitrary function regression for the cost-to-arrive of each point in the corresponding field.

A heap priority queue is initialized to contain only q_s and is in order of $\mathcal{D}(q)$ of each contained configuration q. The queue will only contain points that have their field regressed, i.e. there exists $\mathcal{R}(q)$. When a sample q is popped off, we know that each of its neighbors q', $q' \in \mathcal{D}$, since $\mathcal{D}(q')$ was used to train the regression $\mathcal{R}(q)$. To make progress, we want to regress the distance to start function around each neighbor q'. To do this, we must first calculate the cost-to-arrive of each of q''s neighbors, denoted by q''.

Instead of calculating distance based on the parent's exact distance, the distance is calculated using the parent's regression, with an optimally chosen point in the parent's range. This optimization step is the key advantage of this approach, as it maintains approximate optimality and mitigates the accumulation of error that limits the performance of discrete graphs. Line 7 in algorithm 1 formally defines the optimization.

When a connectable path of cells reaches the goal, we apply an optimizing approach between each cell's region to get the final path. Start from q_g and its nearest field F_g and the cost-minimizing point connecting it to its parent field. Recursively apply this approach until the parent field is the field containing q_s.

Data: Set of sample configurations Q_S in Q_{free}
1 $\mathcal{F} \leftarrow \{F(q) q \in Q_S\}$;
2 $P \leftarrow [q_s]$;
3 **while** $|P| > 0$ **do**
4 $q \leftarrow$ ExtractMin(P);
5 **for** $q' \in$ Neighborhood(Q_s, q) **do**
6 **for** $q'' \in$ Neighborhood(Q_s, q') **do**
7 $q_o \leftarrow \min_{q_o \in \mathcal{F}(q)} \mathcal{R}(q)(q_o) + d(q'', q_o)$;
8 $\mathcal{D}(q'') \leftarrow \mathcal{R}(q)(q_o) + d(q'', q_o)$;
  end
9 $\mathcal{R}(q') \leftarrow$ Regress(q');
  end
end

Algorithm 1: ConstructFST

3.3 Experiments

In this section, we run simple evaluations in the planar case against a visibility graph to check optimality and against RRT* toe valuate practical performance. We also provide performance evaluations for simple cases in higher dimensional spaces, including the Reeds-Shepp car and n-joint revolute robot arms. We find that the FST* provides lower cost paths than RRT* for the same memory cost, and that for spaces with efficient local planners, finds its solution in shorter amount of time as well.

The memory cost in floating points (FPs) for an FST* with n samples in d dimensional configuration space is the sum of all numbers required to represent each field center, regression, and parent, which is calculated as follows:

$$\text{FP}(\text{FST*}) = n(d + (d + 1) + 1) = 2n(d + 1)$$

To keep the RRT* at the same memory cost for comparison, we terminate the construction algorithm when the size of its graph exceeds the size of the FST*.

Environment	Algorithm	Memory cost (FP)	Time cost (sec)	Path cost
WORLD-HALFDOOR	FST*	2,700	0.138	1.564
	RRT*	2,700	0.221	1.629
WORLD-MAZE	FST*	48,600	4.021	4.180
	RRT*	48,600	25.12	-
	RRT*	120,000	103.3	4.211
WORLD-SPLIT	FST*	3,600	0.216	1.459
	RRT*	3,600	0.306	1.488
ARMS-2D	FST*	3,600	1.125	2.946
	RRT*	3,600	4.521	3.328
ARMS-3D	FST*	97,600	489.3	3.822
	RRT*	97,600	502.8	3.911
RSCAR-1	FST*	97,600	807.1	5.134
	RRT*	97,600	527.9	5.527

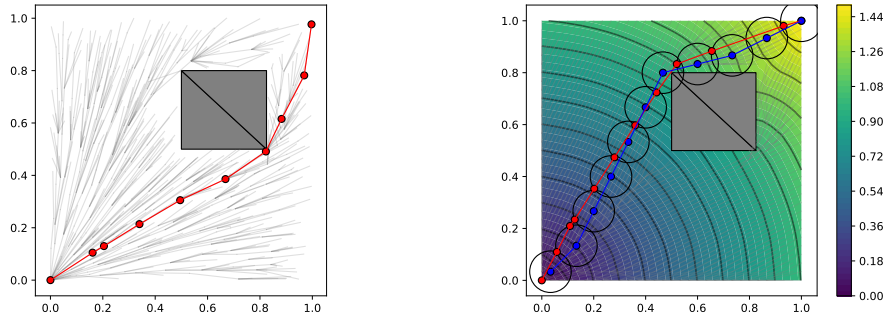
Table 1: Experimental results for FST* and RRT*.

3.3.1 Planar point robot

The example shown in subsection 3.1 used a planar point robot with a single polygonal obstacle. Here the metric d was defined as Euclidean distance, and the local planner Δ was implemented by checking a straight line intersection with all polygonal obstacles. For computing certificates in the planar space, the nearest obstacle can be easily detected using the GJK algorithm, as described in [5]. Because GJK requires convex bodies, we use the ear-clipping algorithm described in [10].

We ran experiments on the same search space but with more complicated obstacle configurations to demonstrate statistical and practical advantages of FST* over RRT*. For all experiments described below, refer to Table 1 for the experimental results. The results demonstrate a clear memory and time cost advantage of FST* in this workspace, along with having useful properties as shown through experiments on WORLD-SPLIT and WORLD-MAZE.

One useful property of FST* is its accuracy in terms of the optimal path's homotopy class. In worlds like WORLD-SPLIT where an obstacle splits possible paths into two homotopy classes, the RRT* has a high chance of choosing the wrong path. Thus even with post-processing optimizations like CHOMP [23], the RRT* gives the wrong answer. However, because FST* maintains a relatively accurate approximation of the cost-to-arrive function, the approach is much more likely to produce paths of the optimal homotopy class.



(a) RRT* on WORLD-SPLIT

(b) FST* on WORLD-SPLIT

Figure 3: FST* (path cost 1.459) picking the optimal homotopy class over RRT* (path cost cost 1.488).

The FST* is also better at exploring obstacle-dense regions. Consider the environment of World-Maze, shown in Figure 4, where the RRT* fails to find a path even after 10k sample points are placed. With the same memory cost (about half as many sample points) the FST* finds a near optimal path.

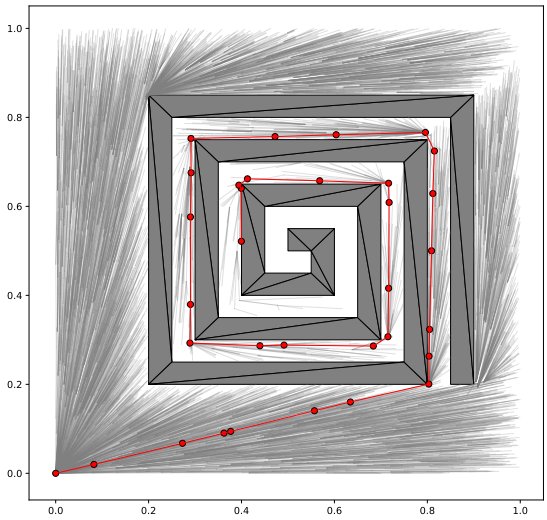


Figure 4: RRT* over world-maze, path incomplete.

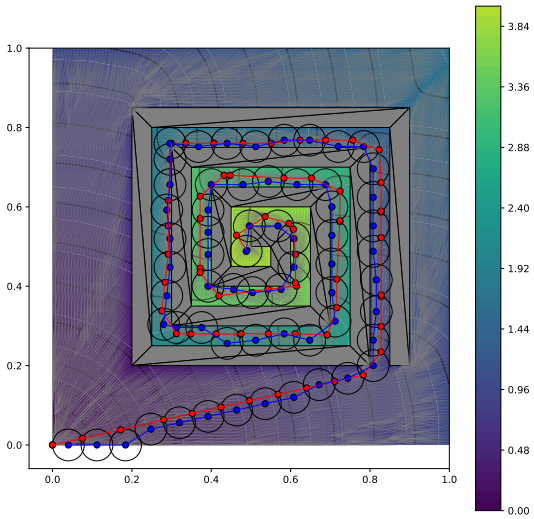


Figure 5: FST* over world-maze, path cost 4.180.

3.3.2 n-joint revolute arm

To test the algorithm in more complicated configuration spaces, we explore applying FST* to an n-joint revolute arm whose configuration can be described by $q = (\theta_1, \theta_2, \ldots, \theta_n)$. Let $L = \{l_1, \ldots, l_N\}$ be the lengths of each section of the arm and $\mathcal{O} = \{O_1, \ldots, O_M\}$ be the set of polygonal obstacles. For simplicity, we add a constraint $\theta_i \in [-\pi, \pi]\ \forall i \in [n]$. We define the distance metric $d(q, q') = \max_{i \in [n]} |q'_i - q_i|$, where q_i denotes the ith parameter of q. If we assume that each joint has the same bound on its angular velocity, the above metric represents the time cost of going from configuration q to q'. To implement the local planner $\Delta(q, q')$, we use linear interpolation of q and q' and check all intermediate sample configurations for collisions with obstacles using inverse kinematics.

Computing the certificate for this space is more involved. Using the metric of time, we assume a maximum joint angular velocity of $\omega = 1\frac{rad}{sec}$. Considering each length l_i independently, the maximum velocity of any point on each joint is $v_i = \omega l_i = l_i$. Given this maximum velocity per joint, we can provide a loose upper bound on the maximum velocity as follows:

$$v_{max} \leq \sum_{i \in [N]} v_i = \sum_{i \in [N]} l_i$$

We can then calculate a lower bound on the cost to the nearest obstacle by applying the GJK algorithm between all-pairs of lines and obstacles:

$$\frac{\min_{L \times \mathcal{O}} \mathrm{GJK}(l_i, O_i)}{v_{\max}} \geq \frac{\min_{L \times \mathcal{O}} \mathrm{GJK}(l_i, O_i)}{\sum_{i \in [N]} l_i} = d_{\min}$$

In both 2D and 3D cases, the FST* outperformed RRT* in terms of memory cost and path quality, with roughly equivalent time cost. See Table 1 for detailed results. Figure 6 and Figure 7 shows the path obtained from the FST* and RRT* respectively.

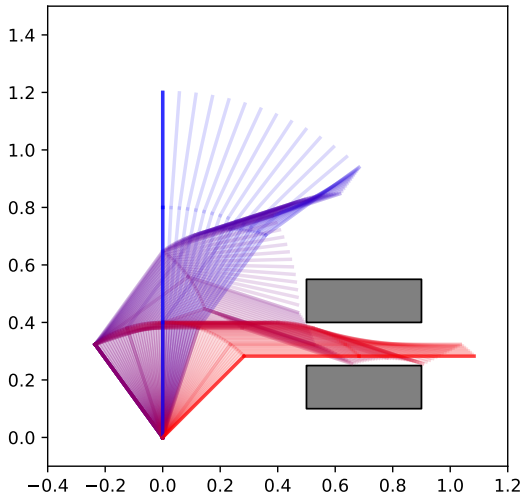


Figure 6: FST* over ARMS-3, path cost 3.822.

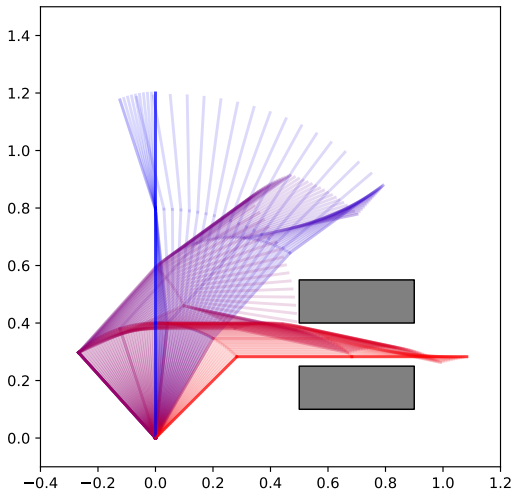


Figure 7: RRT* over ARMS-3, path cost 3.911.

3.3.3 Reeds-Shepp car

We tested the algorithm on a Reeds-Shepp car [25] with obstacles which is shown in Figure 8. Although it is still a very simple model of a car, the Reeds-Shepp car differs from the point robot in 2D in that it is *nonholonomic*, meaning that its motion is constrained by differential constraints. In this case, the car has a direction θ in addition to its position (x, y), and has a constant velocity v and a turning radius r. Thus, even if two configurations are very close to each other, the fact that the car cannot strafe sideways means that the car may need to perform maneuvers to move back and forth. Optimal motion for these cars in the absence of obstacles has been implemented in `SimpleCarModels.jl` [26], which we utilize in our library. The cost function $d(q_1, q_2)$ is directly provided by the library and we implement the local planner $\Delta(q_1, q_2)$ by running a collision check on all waypoints along the optimal path for a fixed time step. We provide a certificate given by the Euclidean distance between the car and the nearest point on the nearest obstacle via GJK divided by the maximum velocity.

Because the local planner requires explicit reconstruction of an optimal path rather than simple linear interpolation, the local planner cost for the Reeds-Shepp car is relatively high, which caused both RRT* and FST* to have longer run times. Because FST* requires more collision checks, this caused FST*'s runtime to be worse than that of RRT* for this space.

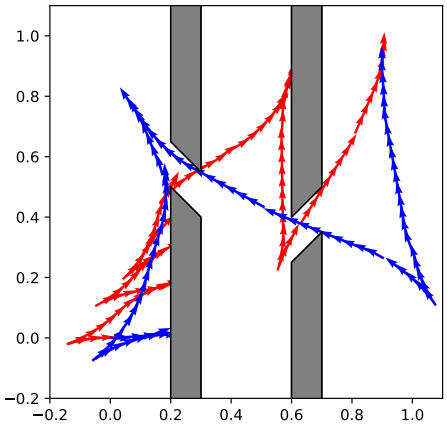


Figure 8: FST* (blue, cost 5.134) and RRT* (red, cost 5.527) over RSCAR-1.

3.4 Limitations and future work

The FST* provides a useful method for constructing a distance function for one-shot planning. However, the requirement for dense coverage of the space via metric balls along with the cost of optimizing over higher dimensional fields makes the approach infeasible for problems beyond three dimensions.

Unlike Field D*, which is limited to two-dimensional planning problems with Euclidean metrics, the FST* algorithm is generalized to work in a configuration space of any dimension. However, an increase in dimensionality exponentially increases the number of fields required, since the fields must cover the entire space. Since the dimensionality of each field also increases the the dimension of the space, optimization over the field becomes more expensive. Optimizations can require up to 20 local planner calls and distance evaluations per field, so our experiments were limited to two- and three-dimensional cases.

We also find that as a function approximation, a cloud of regressed fields kept in a nearest neighbor structure is not a very compact representation. For FST* to be an memory-efficient approximation of the distance function, we propose resampling the space and training a more general purpose regressor such as a neural network or PLR. Instead of keeping the entire tree around and planning based on the tree, perhaps a more compact representation of the distance function could be used with a modified gradient descent approach.

Smarter shaped cells, more accurate certificates, and more sophisticated regression techniques could significantly approve the approach. Finding a useful way to integrate this technique into an all-pairs planner would also improve the applicability of the search method.

3.4.1 Exploration: regressing error functions

Another weakness of the FST* in its current implementation is the incompatibility of linear regressions with intrinsically curved metric functions. Even a workspace without obstacles can have a highly nonlinear configuration space, which is difficult to regress with simple linear approximations. To focus the regression on the error in the distance function caused by obstacles, it may be useful to regress how much the distance has changed due to the obstacles. To this end, define the *metric difference* of a configuration q to be the difference, denoted by $\mathcal{E}(q)$, between the shortest feasible path to the start and the metric distance to the start, i.e. $\mathcal{E}(q) = D(q, q_s) - d(q, q_s)$, where D

represents the true value function of the distance to goal.

Each field is owned by a center configuration q and has a domain defined by the metric ball of radius defined by the certificate of q. As defined before, each configuration has an associated field, denoted by $\mathcal{F}(q)$.

Define the *origin field* to a configuration q, denoted by $O(q)$ to be the field closest to the start that we can connect directly to. Define the *parent field* to a configuration q, denoted by $p(q)$ to be the field that q was discovered from.

First check connection to origin field of $p(q)$, and if possible, connect. If not, get the path of all fields connecting $p(q)$ and $O(P(q))$ in the parent tree. Find the closest field to $O(p(q))$ that q can connect to. This field is $O(q)$.

The metric difference is equal to the difference between the direct path from q to $O(p(q))$ and the obstacle avoiding path, which must go through $d(O(q))$, plus any previously accumulated error getting to $O(p(q))$. Now that we have found $O(q)$, we can calculate the cumulative error of q as follows:

$$\mathcal{E}(q) = d(O(p(q)), O(q)) + d(O(q), q) - d(O(p(q)), q) + \mathcal{E}(O(p(q)))$$

.

This error-function-based approach is not yet entirely feasible, as there are many aspects that remain unclear. In the above formula, the metric d is being applied on field objects (note that $p(q)$ and $O(q)$ are fields), not configurations. The distance between two fields requires optimization along the field boundary, which becomes computationally expensive if it is required often.

3.5 Lessons learned

As hypothesized, using regressions to maintain a better approximation of the optimal distance to goal is much more effective than using discrete graphs. However, this approach is highly susceptible to the curse of dimensionality, because the optimization between fields is expensive. At least in the case of single-goal planning, a more optimized discrete approach such as FMT* is more effective in higher dimensions. Although continuous methods are mathematically elegant and intrinsically a better representation of the planning problem, our current approach is not refined enough to take full advantage of the representation. However, the advantages of regression based approaches, particularly the notion of decomposing a space into regressed subregions,

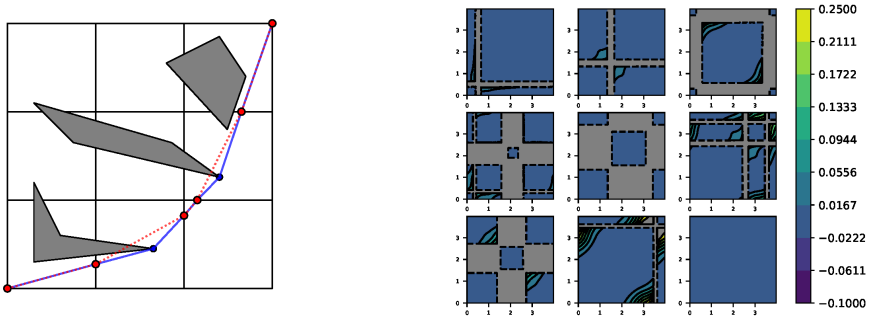
may prove useful for the all-pairs problem, which we explore in the following section.

We also found that when designing a motion planning algorithm, it's easy to guarantee nice properties for the worst-case scenario by using a breadth-first approach. By growing outwards in cost-to-arrive space, we can guarantee that when the goal is found, simple backtracking finds the nearly optimal path. This avoids complex rewiring logic like that found in RRT*, which is both error-prone in implementations and an expensive computation in the inner loop of the algorithm. Unfortunately, this means that it is difficult to find heuristics that are admissible for maintaining this property. Any form of depth-first exploration prioritizing nodes with a minimal heuristic value may require rewiring later, and in an adversarial case, this rewiring will occur often. This becomes a design tradeoff: for a better average case, employing heuristics is useful, but to ensure easy implementations and a good worst-case runtime, relying purely on breadth-first is the preferred approach.

4 Regression complex

In this section, we extend beyond single-start planning solved by FST* to the all-pairs problem and explore the effectiveness of continuous representations in this problem space. Current state-of-the-art sample-based algorithms such as PRM* and its relatives require dense discrete graphs for maintaining optimality. We show that by using a divide-and-conquer method, the configuration space can be decomposed arbitrarily into a complex of cells, where each cell maintains a regression of the all-pairs distance function between any two points on the boundary. This framework decomposes the motion planning problem into a set of smaller, regional planning problems within each cell, which can be done in parallel. When the regression is complete, any intermediate planning results (i.e. samples for the regional roadmaps) can be discarded, providing a useful low-memory data structure summarizing the configuration space.

4.1 A simple example



(a) 2D environment.

(b) Cell distance functions.

Figure 9: Complex with distance calculated from visibility graph.

To illustrate idea of decomposing an all-pairs planning problem into regional planning inside of cells, consider the trivial example of a planar space for a point robot with polygonal obstacles in Figure 9a. We decompose the space into a 3×3 grid of square cells. For visualization purposes, we parameterize the boundary B of each cell with a real number in the range $[0, 4)$.

To calculate this function, we need a regional planner than can connect any two points on the boundary. For this example, we use a visibility graph, although more complicated spaces will require a more general planner such as a PRM*. Let the cost function between points on the boundary obtained from the regional planner be $f_i : B \times B \mapsto \mathbb{R}$, where i is the index of the cell. Figure 9b shows each of the boundary cost functions for the nine cells. Where the regional planner fails to find a path, f_i is undefined. Such undefined regions are shown in gray. We now have have a value function along all boundaries of the cells, and the motion planning problem has been reduced to finding the best points along these boundaries to build a path through. In this case, the best crossing points are those shown in red in Figure 9a, and the blue path shows the feasible path joining the crossing points. The rest of our discussion of the regression complex will be focusing on the details of this procedure: how we represent the all-pairs boundary cost function, how we can use it to find the optimal crossing points, and how we rebuild the paths between the crossing points.

4.2 Approach

In this section, we describe an approach to constructing and querying the regression complex (RC) data structure using a piecewise linear regression to approximate cell-crossing costs.

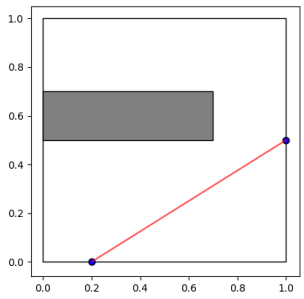
4.2.1 RC construction phase

The construction phase decomposes Q into a set of cells $\mathcal{C}$, where each cell $C \in \mathcal{C}$ is an n-dimensional hypercube. For each cell, the basic approach is to sample distances between pairs of points along the boundary of the cell, and then to use this data to build a regression.

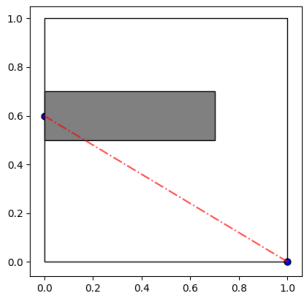
Let $B(C)$ represent the boundary of cell C. We associate with each cell C a regression $R_C : B(C) \times B(C) \mapsto \mathbb{R}_{\geq 0}$ that maps pairs of points on its boundary to the distance between them. A classifier $K_C : B(C) \times B(C) \mapsto \{0, 1\}$ is also constructed to represent connectivity of pairs of boundary points.

Any pair of points on the boundary of the cell can be categorized into one of the four cases shown in Figure 10. We handle each of these cases differently in the representation of the regression complex data structure.

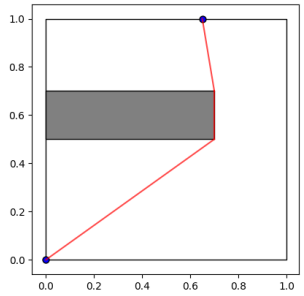
Two points on the boundary of a cell are either connectable by a path that remains within a cell (Figure 10b, Figure 10d), or they are not. Although



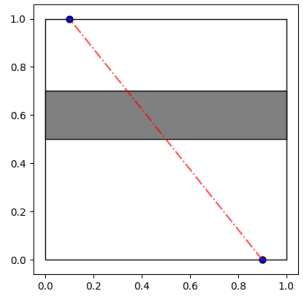
(a) Metric given by local planner.



(b) One point in collision.



(c) Regression used to approximate metric.



(d) Boundary points in separate components.

Figure 10: Pair-wise relationship classes for boundary points.

one could imagine the path cost between unconnectable points as infinite, the resulting discontinuities in the value function are problematic for a smooth regression representation.

We use two approaches to mask out unconnected configurations from the regression. First, we use the collision detector to find if either boundary point is itself in collision with an obstacle, in which case the sample is discarded. Second, if the boundary points themselves are collision-free, but the local planner fails to find a path connecting the points *without leaving the cell*, we record this fact using a classifier, the implementation of which we discuss below.

A key property of this cell decomposition is that we do *not* require that the local planner stays within the cell. As long as the path is free, its distance is used to build the cell's regression. Allowing paths to leave the cell separates

the problem of determining cell size from the geometry of the obstacles, allowing much larger cells than those used in typically cell-approximation methods for motion planning (*e.g.* [1, 2]). This means that a path can leave or enter the cell as many times as it needs to in order to connect two boundary points, and we rely on the query phase procedure to guarantee that paths of this form are indeed optimal.

We also introduce an implementation-level optimization to exploit the fact that for paths that do not contact obstacles (Figure 10a), the local planner already provides an accurate local metric. In other cases, the local planner provides a useful estimate even if a path gently grazes an obstacle. To take advantage of this property, we regress the error function between the cell-crossing distance function and the metric provided by the local planner. Using this approach, if the local planner is accurate over some region without obstacles, the regression needs only to store the constant value 0 over that region. This technique is identical to the method suggested in subsubsection 3.4.1, but applied in the context of the RC.

The training data used for the cost regression and the connectivity classifier can be found using any all-pairs motion planner. For simplicity of analysis and comparison, we make use of a regional PRM* inside the cell.

Only R_C and K_C (if needed) are saved in the regression complex eventually to be further used in query phase; all data from the regional planner can be discarded. The algorithm is model-agnostic, and advantages of various regression models are discussed further in Appendix A. For the purpose of our proofs, we assume the use of PLR (discussed in subsection A.1) as the regression technique, which has useful approximation factor bounds based on the Lipschitz continuity of the value function. For clarity, we denote this specific implementation of the RC framework as PLRC*.

4.3 RC query phase

Given a start configuration $s \in Q_f$ and a goal configuration $t \in Q_f$, a query is performed using a simple graph search with discretization along the boundaries, as described below:

4.3.1 Boundary graph construction

We construct a weighted graph $G_b = (V_b, E_b, w_b)$, whose vertices V_b are samples in Q_f along the boundary of every cell $C \in \mathcal{C}$. The edges E_b are pairs of

points that are in the same cell C and are classified as connectable by K_C, i.e. $E_b = \{(q_1, q_2) \text{ s.t. } q_1, q_2 \in C, K_C(q_1, q_2) = 1 \text{ for some } C\}$. The weight map w_b for each edge is calculated using the regressor, namely $w_b(q_1, q_2) = R_C(q_1, q_2)$. We then run A* search over G_b to return the path P_0, the unrefined path whose waypoints represent the points that the final path P needs to go through.

4.3.2 Path refinement

One can imagine many ways of reconstructing P from P_0. A PRM* could be constructed within each cell that P_0 passes through, and the path between each waypoint could be calculated using A* search. However, this approach requires constructing an incredibly dense PRM*, which is costly in terms of memory and defeats the purpose of using this algorithm. Alternatively, an optimizer could be used to place intermediate points between each waypoint such that the path between the added points is collision-free and the cost is minimized. Our approach uses the optimizer, and incrementally starts by attempting the connection with a single intermediate point. If the connection fails, an additional point is added and the procedure continues until the points are connected. This is guaranteed to terminate because the points on the graph were classified as connectable by their cell's classifier K_C.

4.4 Experiments

This section and its experimental results and figures were made with the help of Luyang Zhao.

This section presents the results of several computational experiments in which PLRC* data structures are constructed and queried. The results are compared against PRM*, demonstrating the memory cost advantage of PLRC*.

To measure the memory cost of each approach, we compute the number of floating points needed to store each data structure; this is a rough but perhaps useful estimate. For the PLRC, the memory cost includes the cost of both the regressor and the classifier. The cost of each is the sum of the number of numbers needed to store the parameters for each linear regression, the bounds of each cell in the BSP, and the number of cells needed, based on the depth of the BSP.

$$\text{FP(plr)} = \text{FP(parameters)} + \text{FP(bsp)} + \text{FP(bounds)}$$

where FP(x) denotes the number of floating points used by component x. In detail, FP(bsp)+FP(bounds) is the floating points of the data structure used to store the parameters of PLR, where FP(bsp) equals $num(leaves) - 1$ and $FP(bounds)$ equals $2 * D$ and D denote the dimension of the configuration space. The memory cost of PRM* is just the memory cost of storing the vertices and graph, which is $N * D + N * k * 3/2$, where N is the number of sample points used in the construction phase, and k equal to $\log_2 N$.

Environment	Algorithm	Memory cost (FP)	Unrefined cost	Query time (sec)	Refinement method	Path cost	Refinement time
WORLD-DOORS	PRM*	215000	-	0.09644	-	6.14225	-
	PLRC*-4	9184	6.04741	0.01806	Optimizer	5.88847	161.16488
					PRM*	6.10913	41.43030
	PLRC*-16	4620	6.01461	0.00956	Optimizer	5.89986	14.75135
					PRM*	6.07112	18.71936
	PLRC*-32	3968	6.04388	0.00933	Optimizer	5.94593	12.99612
					PRM*	6.10998	12.47269
WORLD-MAZE	PRM*	215000	-	0.03401	-	4.09641	-
	PLRC*-4	21224	3.99711	0.00449	Optimizer	3.97747	1.65003
					PRM*	4.07569	19.37482
	PLRC*-16	5058	3.99963	0.00288	Optimizer	3.94168	2.61319
					PRM*	4.07431	7.29743
	PLRC*-32	4874	3.97164	0.00809	Optimizer	3.93096	15.92245
					PRM*	4.06221	9.44786
3D Arm in WORLD2	PRM*	210875	-	0.11388	-	5.44542	-
	PLRC*-8	21554	5.23471	0.04129	PRM*	5.40983	22.37812
RS Car in WLD-DOORS2	PRM*	180000	-	0.083801	-	5.77792	-
	PLRC*-8	25122	5.48602	0.035062	PRM*	5.69049	33.10321

Table 2: Comparison of PRM* and PLRC*.

4.4.1 Planar point robot

We performed our first experiments for a planar point robot with $q = (x, y) \in \mathbb{R}^2$. The robot moves amongst polygonal obstacles from a start configuration to a goal configuration.

We tested several different environments, and present results (Table 2) for two of the more interesting, which we name world-doors (Figure 11) and world-maze (Figure 12). We used different numbers of cells, ranging from 4

to 32 (PLRC*-4 to PLRC*-32).

From Table 2, we can see that the memory required to store the PLRC data structure is about 20 times less than that required for the PRM* for these examples, and query of the PLRC returns a slightly more accurate approximation of the path cost. (The optimal path distance of maze-doors is 5.859918264 and of maze-world is 3.911704702 calculated from visibility graph[19].)

Surprisingly, when the number of cells increases, PLRC* seems to require less memory. When we divide the regions into more small cells, the number of sample points become smaller and the distances of those points has lower variance; the piecewise linear regression subdivides these simpler smaller cells less.

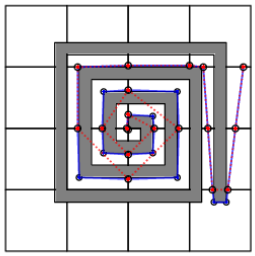
The query phase of PLRC returns only the path cost and some points where the approximation of the optimal path intersects with cell boundaries. To find a higher-resolution path that avoids obstacles within cells, some refinement step is required, as described above.

Figure 11a and Figure 12a show the path after refinement by optimizion techniques and Figure 11b and Figure 12b show a path after refinement using a local PRM* built within each cell along the approximately optimal path. We note that the optimizer returns very good paths indeed, but the comparison to PRM* is not quite fair, since PRM* does not execute a refinement step. Figure 11d and Figure 12d show the path computed by PRM* directly. Figure 11c and Figure 12c show the connectivity of boundary points, as captured by the classifier.

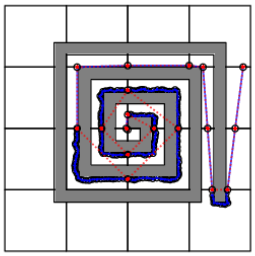
4.4.2 n-joint revolute arm

Our next experiments were performed on an n-joint revolute arm whose configuration can be described by $q = (\theta_1, \theta_2, \ldots, \theta_n)$. As with the same world in the FST* experiments, we add a constraint $\theta_i \in [-\pi, \pi]\ \forall i \in [n]$. For more details on the implementation of the distance metric and local planner, see subsubsection 3.3.2.

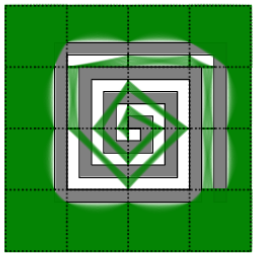
Table 2 shows the comparison results of 3R arms from PRM* and PLRC*-8, where the memory cost of PRM* is 23 times of that of PLRC*-8 while the path accuracy of PLRC*-8 is much higher than that of PRM*. Figure 13 shows path obtained from PLRC*-8.



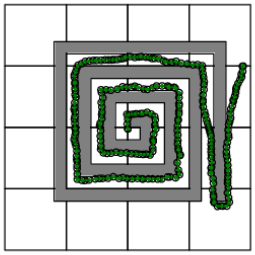
(a) PLRC*-16 with optimizer for reconstruction.



(b) PLRC*-16 with PRM* for reconstruction.



(c) Connection relationship between boundary points.

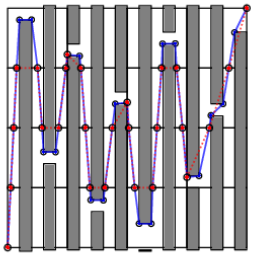


(d) Path from PRM*.

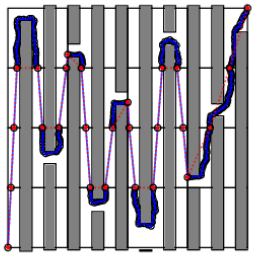
Figure 11: World-maze: PRM* paths, and PLRC*-16 with different refinement techniques.

4.4.3 Reeds-Shepp Car

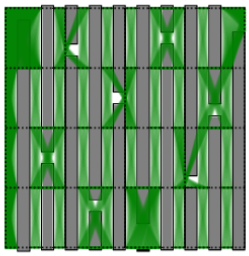
We tested the algorithm on a Reeds-Shepp car [25] with obstacles which is showed in Figure 14. For a detailed description of our implementation of the distance metric and local planner, see subsubsection 3.3.3. The second row of Table 2 shows the comparison results of Reeds-Shepp car from PRM* and PLRC*-8, where the memory cost of PRM* is 7 times of that of PLRC*-8 while the path accuracy of PLRC*-8 is higher than that of PRM*.



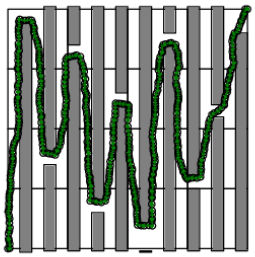
(a) PLRC*-16 with optimizer for reconstruction.



(b) PLRC*-16 with PRM* for reconstruction.



(c) Connection relationship between boundary points.



(d) Path from PRM*.

Figure 12: WORLD-DOORS: finding path with prm* and PLRC*-16 with different reconstruction techniques.

4.4.4 Comparison to spanner algorithms

This section's insights pertaining to graph spanners were provided by Weifu Wang.

Applications of graph spanner algorithms to roadmaps for motion planning can also significantly reduce memory costs while returning *good quality* paths [20, 18, 29]. Most spanning algorithms first construct the entire PRM over the entire space, and then prune the roadmap in a post-processing phase. Streaming spanners [29] that omit edges during construction avoid storing the entire map, but the end result is still a discrete graph which has a much higher memory cost that the continuous representation offered by the regres-

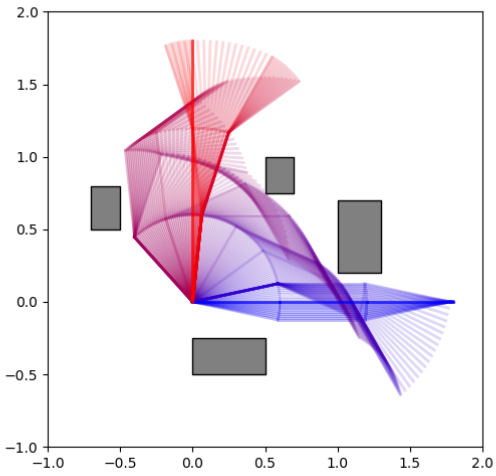


Figure 13: 3-joint revolute arm path obtained by PLRC* from $(0, 0, 0)$ (blue) to $(\frac{\pi}{2}, 0, 0)$ (red) in WORLD2. Total path cost 5.40983, compared to 5.44542 for PRM*.

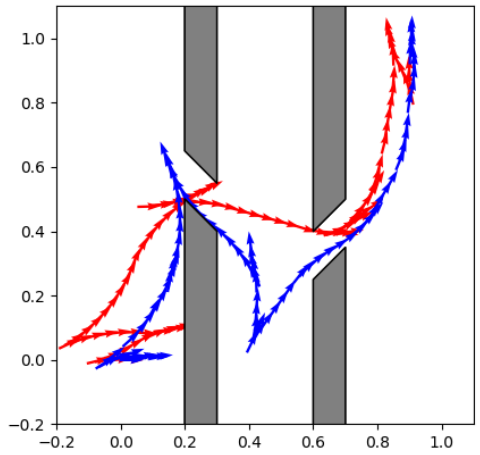


Figure 14: Reeds-Shepp Car path obtained from PRM* (blue, path cost 5.77792) and PLRC* (red, path cost 5.69049).

sion complex. The spanner algorithm presented in [29] (WSS) on average retains 20 to 30% of the edges needed by PRM*, in some cases only 10% if provided with enough stretch. This optimization in memory comes at the cost of optimality, as the average path quality for the WSS is 10 to 20% worse than that of PRM*. The main advantage of the WSS is its runtime, which is 50 to 70% faster compared to PRM$*$, as many collision detections are not needed for the edges discarded by WSS online. However, even with many edges omitted, the WSS usually requires 10 times the data needed compared to the regression-based approach, and still outputs paths with worse quality.

4.5 Use case: high memory cost cell planners

One key advantage of using the regression complex approach is its ability to process the configuration space Q on a cell by cell basis, avoiding the need to store each cell planner for the entire complex. Once a cell C has its all-pairs distance information summarized via the regressor R_C and classifier K_C, all information specific to that cell planner can be discarded. Consider an instance of a motion planning problem where a PRM* is used as the cell planner. If PRM* must be used across the entire space, the entire roadmap must be maintained and searched across. With a regression complex, a PRM* is constructed in each cell, which is then summarized and discarded. When a query is run on the regression complex, cells for which the cell planner is required (see subsection 4.3) can reconstruct the PRM* in that cell alone. Figure 15 depicts the regression complex run on a maze with a PRM* cell planner. Although Figure 15 shows PRM*s for each cell was relevant to the query, there is no need to keep more than one cell's cell planner in memory at any given time – once the path is found, the cell planner may be discarded. This approach allows motion planning for problems where planning over the entire configuration space requires too much memory to be computationally feasible. In the example shown by Figure 15, the regression complex had a maximum memory usage of $163,200$ floating points, whereas a PRM* of equivalent path quality required a maximum memory usage of $460,000$ floating points.

The independence of individual cell construction also means this process can be easily parallelized. This can be useful for systems where memory is not a constraint, or where many processors with associated local memory are available. This allows a highly configurable speed-versus-memory tradeoff that can be tuned on a per-system basis.

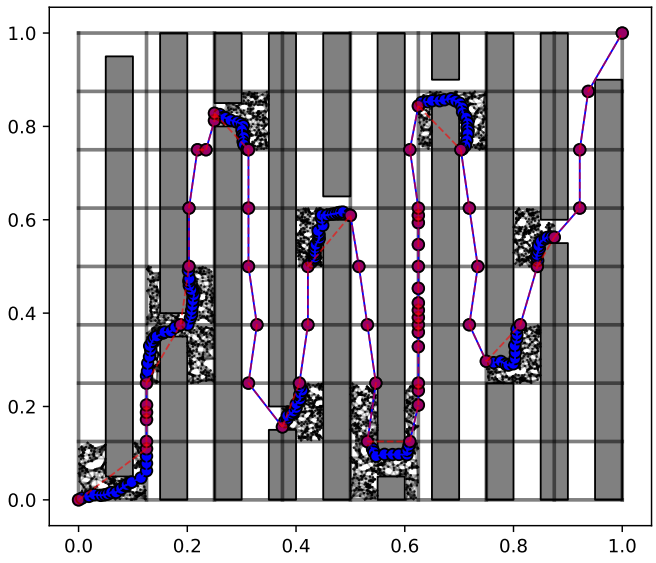


Figure 15: Reconstructed cell planners for a regression complex on for a problem with an intricate configuration space.

4.6 PLRC* Asymptotic Completeness and Optimality

The following proofs refer to a PLRC* used to solve a motion planning problem (Q, q_s, q_t) with a robust solution $\sigma^* : [0, 1] \mapsto Q_f$ with strong δ-clearance.

Let $\mathcal{C}$ denote the complex of regression cells generated by the PLRC*. Let $G_b = (V_b, E_b, w_b)$ be the query graph described in subsubsection 4.3.1. Assume that each cell $C \in \mathcal{C}$ uses PLR with minimum cell edge size ϵ as regressor and classifier, and that boundary samples are placed in a grid with step size ϵ_s.

Because the cells in $\mathcal{C}$ cover all of Q, any path between configurations must trivially cross the boundaries of $\mathcal{C}$. We can define these crossing points as the values $b_1, \ldots, b_n$ of σ^* which corresponds to a path $\pi_b = (\sigma^*(b_1), \ldots, \sigma^*(b_n))$ such that each $\sigma^*(b_i) \in B(\mathcal{C})$.

Lemma 1 (Convergence of boundary graph accuracy)**.** *Let π^* denote the shortest path from q_s to q_t on boundary graph G_b. As $\epsilon \to 0$ and $\epsilon_s \to 0$, the distance between each configuration $\pi^*[i]$ and $\pi_b[i]$ approaches* 0.

Proof. Because the boundary sampling method samples cells with a sampling width of ϵ_s, any configuration $q \in B(\mathcal{C})$ is at most $dim * \epsilon_s$ away from a sample q_V, where dim is the dimensionality of the C-space Q. Thus as $\epsilon_s \to 0$, the distance from q to the nearest configuration $q_V \in V$ approaches 0. To show that the nearest q_V is on the path π^*, we show that the edge weights w_b match the distance function d at the limit of ϵ.

Given a value function $V(\cdot)$ with Lipschitz continuity factor κ, we know from Theorem 2 of [22] that for the approximation $L(\cdot)$ from the PLR, $|V(q) - L(q)| \leq \frac{5}{2}\kappa\epsilon\sqrt{n}\ \forall q \in Q$. Consider the regression R_C over the all-pairs distance function d, which PLRC* uses to calculate the graph weights w_b. By the inequality above, as $\epsilon \to 0, |R_C(q_1, q_2) - d(q_1, q_2)| \to 0$ as long as q_1 and q_2 are correctly classified. The accuracy of K_C with respect to the connectivity function converges to 0 by the same argument.

Because w_b is a good approximate of d in the limit of ϵ and π^* is the shortest path on G, we know that if configurations are sampled close to each boundary waypoint q, the nearest configuration $q_V \in V$ must be on π^*. Therefore as $\epsilon \to 0$ and $\epsilon_s \to 0$, the distance between each configuration $\pi^*[i]$ and $\pi_b[i]$ approaches 0. □ □

Theorem 1 (Resolution completeness of LPRC*)**.** *LPRC* is resolution complete with respect to ϵ and ϵ_s.*

Proof. Let σ denote the path produced by PLRC*. σ is constructed from the waypoints in π^* using the cell planner between each consecutive pair of configurations (q_i, q_{i+1}) on the path π_b. By lemma 1, as $\epsilon \to 0$ and $\epsilon_s \to 0$, a shortest path π_b on G_b must exist connecting q_s to q_t. Because the cell planner PRM* is guaranteed to return a feasible path at high enough resolution, σ must be feasible as long as each consecutive pair of configurations (q_i, q_{i+1}) is connectable. The feasibility of the path between each (q_i, q_{i+1}) depends on the error K_C, which as argued above, converges to 0 as $\epsilon \to 0$. Thus as ϵ and ϵ_s approach 0, PLRC* is guaranteed to return a feasible solution. □ □

Theorem 2 (Asymptotic optimality of PLRC*)**.** *RCRM is asymptotically optimal with respect to ϵ and ϵ_s.*

Proof. Again, let σ denote the path produced by PLRC*. By Theorem 1, we know that the solution σ is feasible, and by lemma 1 the distance between corresponding boundary configurations $\pi^*[i]$ and $\pi_b[i]$ approaches 0. Given a sufficiently dense PRM*, the path returned by the cell planner is the optimal path between each waypoint on π_b. Since each configuration on π_b is

arbitrarily close to π^*, the error of total path length converges to 0 with ϵ and ϵ_s. □

4.7 Limitations and future work

Although the main framework is complete, many questions still remain. How should these regression complexes be constructed efficiently? How should they best be searched for a path? Once a path is found, how can it be refined to avoid obstacles within each cell? For each of these phases of *construction*, *query*, and *refinement*, we have explored only a first approach. For example, in the construction phase, we use a PRM* to measure the underlying cell-crossing metric that the regressions approximate, though we are certain that a PRM* is a bottleneck in the overall approach. We focus on a piecewise linear regression technique for transparency, but can imagine that other techniques (such a neural networks) might be as or more effective. Similarly, in the query phase, we discretize cell-boundaries and search, though the existence of a smooth approximation of the metric strongly suggests an optimization approach like that used by Field D* [12] or in our own development of FST*.

This experimental work, though promising, is limited so far to examples in a few dimensions, due in part to the computational cost of sampling the value functions exhaustively.

The primary contribution of this work is the notion of using spatial decomposition and regressions to construct a hybrid representation of a search space. We compared the performance of the approach against PRM* in various search spaces and provide a model-agnostic algorithm for regressing distances in search spaces for motion planning. We have tested alternative regression teqniques, like xgboost and neural networks, but not deeply.

Computing all-pairs distance using a dense PRM* and repeatedly running A* search is not an elegant approach. Although the regression complex can avoid constructing a dense graph for the entire search space since it only needs roadmaps in local cells, there is still a lot of computation that goes into constructing these local roadmaps.

4.7.1 Hierarchical all-pairs distance computation

Instead of computing all-pairs distances using PRM*, we suggest exploring a divide-and-conquer approach as follows. Start with the entire space as the root cell, and recursively split each cell that is not free up until some maxi-

mum depth. Build approximations for all leaf cells using the local planner. Assuming we have a manner of merging cells' approximations into their parent cell, we can merge all cells in a bottom-up fashion until the desired cell resolution is obtained.

The remaining problem for this approach is that we need a feasible merging technique for combining the all-pairs approximations of the child cells into their parent cells. One can imagine re-sampling points on the boundary of the parent cell, using the child cell's approximations for each pair that does not cross the boundary between the two. However, pairs of points that must cross the boundary are problematic, as they would require optimizing along the boundary between the child cells.

Why should we expect that a divide-and-conquer approach may perform better than existing algorithms like PRM*? When studying chess AI, Shannon's key observation was that for state-space search, inaccurate but inexpensive local information that allows a deeper search is more useful than accurate but expensive local information. This notion suggests that instead of trying to build a small number of incredibly accurate cells using a robust approach such as PRM*, a large number of approximating cells may be more useful for motion planning.

4.8 Lessons learned

Work on the RC revealed two key lessons on divide-and-conquer approaches, both for motion planning and in more general computational problems. First of all, a divide-and-conquer approach is limited by its merge operation. Combining results to return the solution needs to be fast and simple for an algorithm to be effective, no matter what the gains on paralellizability or efficiency for the decomposed problems are. The RC does a great job at reducing the motion planning problem into planning across boundaries of regions with the regressed cost function, but searching along the boundary is difficult. Construction of this decomposition can be easily parallelized and is highly modular, but we are far from satisfied with our current method of merging these results via re-sampling the boundary and running A* search. Before beginning work on this approach, it may have been useful to devise a clever way of planning using the merged results of our decomposition and designed our data structure around that approach, rather than designing the decomposition and then trying to come up with a feasible merge strategy.

Secondly, optimizations of primitive operations are crucial for overall per-

formance. Constructing each cell in the complex requires running a regional planner, and in our current implementation, the regional planner is a dense PRM*. Furthermore, for each regional planner, the local planner must be called many times, so its performance is crucial for the overall runtime as well. Even small optimizations in the local and regional planner helped bring computation costs to a feasible range, and we hope that in the future we will be able to find a better approach for the regional planner altogether.

Appendices

A Regression techniques

In this section we explore various techniques on regressing the all-pairs distance function for regression cells. We demonstrate the performance of linear regressions, piecewise linear regressions, feed-forward neural networks, Gaussian processes, and XGBoost for regressing Euclidean distance functions in the presence of simple obstacles. Table 3 shows the results of our experiments.

Regression	Memory cost (FP)	Time cost (sec)	MSE
Linear Regression	3	0.002	0.051
PLR	74	0.021	0.005
Neural Network	152	2.58	0.011
Gaussian Process	4096	0.231	0.004
XGBoost	300	0.102	0.017

Table 3: Experimental results for FST* and RRT*.

In subsection A.1, subsection A.2, subsection A.3 and subsection A.4 we explore each of the regressions in detail and show plots of their respective error functions. Overall, we find that PLR is the most efficient solution for regressing a Euclidean distance function, although more concise tuning and knowledge of machine learning techniques may make the other approaches more effective.

A.1 Piecewise linear regression

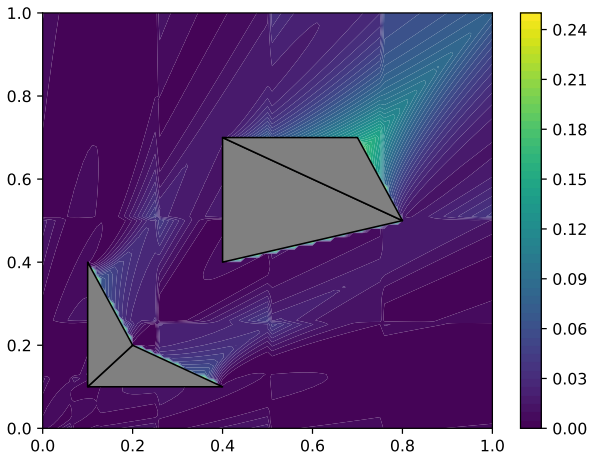


Figure 16: Error function of piecewise linear regression.

The Piecewise Linear Regression (PLR) used to approximate cell-crossing distance functions is discussed in greater detail in a technical report [22], along with proofs of convergence and use cases. The technique is straightforward: we would like to approximate a function from $R^n \mapsto R$ using series of linear regressions. The PLR subdivides the space into cells using a Binary Space Partition (BSP), sampling the function as it goes, and using the samples to compute a linear approximation in each cell of the BSP. Error may then be estimated using further samples; if needed, the BSP is refined further. Out of our experiments, we find that PLR provides the most consistent accuracy for low memory, and due to its simplicity, it provides us with upper bounds on the error based on the target metric's Lipschitz continuity. For this reason, most experiments in the main paper rely on this technique. However, a machine learning engineer would likely be able to beat PLR's performance by clever tuning of standard regression techniques such as the neural network or XGBoost.

A.2 Feed-forward neural network

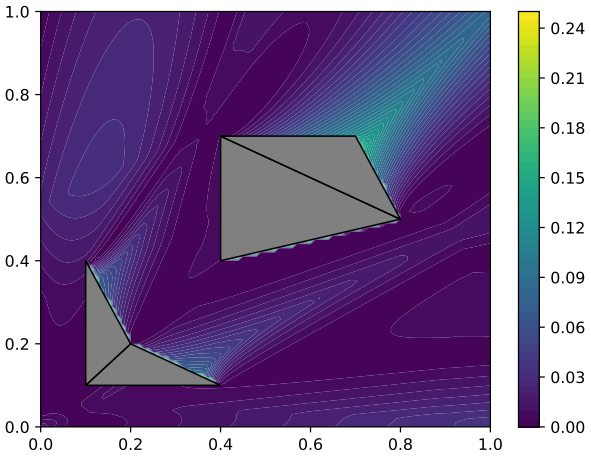


Figure 17: Error function of feed-forward neural network.

We used the `Flux.jl` [14] library for building the neural network. We found that the most cost-effective network for regressing Euclidean distance functions was four layers, each containing 2, 8, 8, and 1 neuron respectively. We found that the most effective activation function was simple ReLU. Although the neural network provided good accuracy for reasonably low memory, training the network required over 100 epochs, which increased the time cost significantly.

A.3 Gaussian process

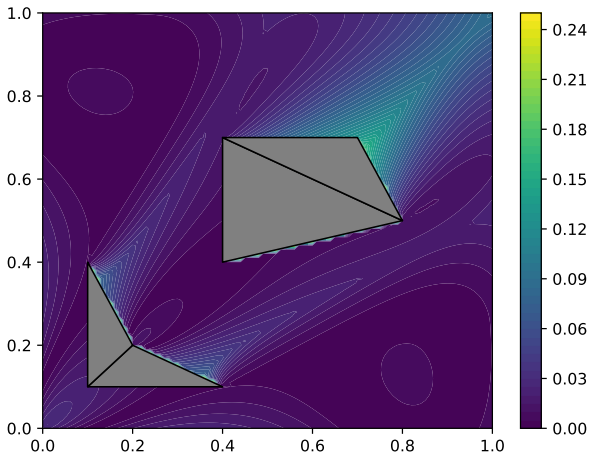


Figure 18: Error function of Gaussian process

To attempt a smoother function approximation, we also tried using a Gaussian Process, which Lisa Oh explored in her thesis research on the same topic. Although this approach gave the lowest mean squared error, its immense memory cost that scales quadratically with sample size makes it infeasible problems with larger training datasets.

A.4 XGBoost regression

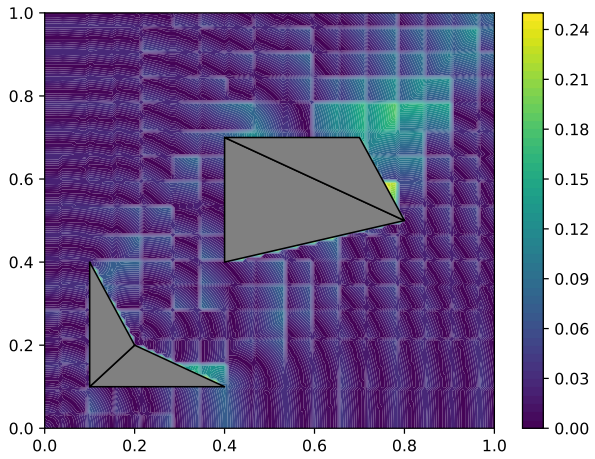


Figure 19: Error function of XGBoost regression

We also tested the XGBoost model described by [6]. We used Julia bindings from `XGBoost.jl` [30]. XGBoost is largely used for deep learning applications, but is also useful as a universal function approximator. We avoid using XGBoost in our formal analysis and experiments due to its lack of provable bounds on the approximation factor, and the unreliability of tuning its parameters. In example, accuracy is affected by both tree depth and the count of trees, both of which have different advantages in different problems, and studying all permutations of parameters could be a paper in itself.

B Recursive Cell Roadmaps (RCRM)

In the spirit of focusing on boundaries of cells for constructing paths around obstacles, we developed an alternative roadmap algorithm to be used for computing all-pairs distances for a given cell in the regression graph. This algorithm takes advantage of large convex regions of Q_f for which samples are unneeded using a recursive cell decomposition of Q. The cells are formed by recursive binary division of Q. Let $\dim(Q)$ represent the dimensionality of Q, and $\text{depth}(C)$ denote the depth of cell C.

Initialize the roadmap to $G = (V = \emptyset, E = \emptyset, w = d)$. Starting at root cell $C = Q$, sample a set of points $X_{B(C)} \subset B(C)$ and set $V = V \cup X_{B(C)}$. C splits along dimension $\dim(Q) \mod \text{depth}(C) + 1$ if $C \subseteq Q_f$, i.e. the entire cell is collision-free. In practice, we can check if $C \subseteq Q_f$ using a certificate based on knowledge of the workspace, or perform brute-force checking using the local planner by checking $\Delta(q_1, q_2) = 1 \; \forall (q_1, q_2) \in B(C)^2$. When a cell no longer needs to split, all-pairs connections are added to the roadmap, setting $E = E \cup \{(q_1, q_2) : q_1, q_2 \in X_{B(C)}\}$. This method of recursive cell division means that samples are focused around obstacles.

Querying the RCRM is simple; given a starting configuration q_s and an ending configuration q_t, find the corresponding cells C_s and C_t such that $q_s \in C_s$ and $q_t \in C_t$. Connect q_s to all sample points in $X_{B(C_s)}$. Then perform A* search to return a path from q_s to q_t.

B.1 RCRM Proofs

The following proofs refer to an RCRM used to solve a motion planning problem (Q, q_s, q_t) with a robust solution σ^* with strong δ-clearance, optimal local planner Δ, and a corresponding metric d.

Lemma 2 (Collision-free cells on optimal path)**.** *Given a minimum cell size ϵ_{cell} such that $d(\vec{0}, \vec{\epsilon_{cell}}) \leq \delta$, q is in a collision-free cell $\forall q \in Q_f^\delta$.*

by contradiction. Consider any configuration $q \in Q_f^\delta$. By definition of strong δ-clearance, $d(q, q_{obst}) > \delta$, where q_{obst} is the nearest configuration in collision. Let C denote the cell containing q. Assume for contradiction that C is in collision, i.e. $\exists q_{obst} \in C$ such that $q_{obst} \notin Q_f$. If $q, q_{obst} \in C$, we know that $d(q_{obst}, q) \leq d(\vec{0}, \vec{\epsilon_{cell}}) \leq \delta$, where $d(\vec{0}, \vec{\epsilon_{cell}})$ denotes the diameter of the cell C under metric d. This contradicts strong δ-clearance, so q must be a collision free cell. □ □

Theorem 3 (Resolution completeness of RCRM)**.** *RCRM is resolution complete with respect to the minimum cell width ϵ_{cell} and the sample dispersion width ϵ_{sample}.*

Proof. To show the existence of a solution σ, we must prove that there exists a path of adjacent cells in the PCRM whose boundary sample points can be connected by Δ to form a feasible path. To show this, we show that the optimal solution σ^* can be expressed as a sequence of configurations on the

boundaries of cells in the RCRM, and that for any point in a collision cell, there exists a nearby boundary point in a free cell.

Because the cells in $\mathcal{C}$ cover all of Q, any path between configurations must trivially cross the boundaries of $\mathcal{C}$. We can define these crossing points as the values $b_1, \ldots, b_n$ of σ^* such that $\sigma^*(b_1), \ldots, \sigma^*(b_n) \in B(\mathcal{C})$.

By lemma 2, each of the cells crossed by $\sigma^*(b_1), \ldots, \sigma^*(b_n)$ are collision-free if ϵ_{cell} is picked small enough such that $d(\vec{0}, \vec{\epsilon_{cell}}) \leq \delta$. By the definition of the boundary sampling method on collision-free cells, for every consecutive pair of boundary waypoints $\sigma^*(b_i), \sigma^*(b_{i+1})$ for $i = 1, \ldots, n-1$, there exists a pair of nearest sample configurations (q_i, q_{i+1}). Because the cell is collision-free, $(q_i, q_{i+1}) \in E$. Since boundaries share sets of samples, the sequence of configurations $q_1, q_2, \ldots, q_n$ forms a contiguous path of edges from q_1 to q_n. Because $(q_s, q_1), (q_n, q_t) \in E$ by definition of the querying procedure, $q_s, q_1, \ldots, q_n, q_t$ is a contiguous path from q_s to q_t, and thus a feasible solution. □ □

Theorem 4 (Asymptotic optimality of RCRM)**.** *RCRM is asymptotically optimal with respect to the minimum cell width ϵ_{cell} and the sample dispersion width ϵ_{sample}.*

Proof. Let σ denote the path produced by the RCRM. Once again, define the crossing points of σ^* as the values $b_1, \ldots, b_n$ of σ^* such that $\sigma^*(b_1), \ldots, \sigma^*(b_n) \in B(\mathcal{C})$. By lemma 2, each of the crossed cells are collision-free. Because the boundary sampling method samples cells with a sampling width of ϵ_{sample}, any configuration $q \in B(\mathcal{C})$ is at most $d(\vec{0}, \vec{\epsilon_{sample}})$ away from a sample q'. Thus increasing the sampling density causes the error from the optimal path to converge to 0. □

C Implementations

We developed a comprehensive library for motion planning using the Julia programming language. This section describes the features and design of the library.

C.1 `MetricTools.jl`

The `MetricTools.jl` sub-library provides the supporting frameworks for all of the other libraries.

This includes `MetricGraphs.jl`, a wrapper around `LightGraphs.jl` specialized for graphs embedded on a metric space. This graph supports automatic calculation of metrics to avoid redundant computation, along with optionally maintaining a KNN Ball-Tree structure for fast nearest-neighbor searches. To support all geometric operations required for collision checking, the library also contains implementations of line-polygon intersections and wrappers around the GJK algorithm for computing minimum distances to convex polyhedra.

Source code is available on GitLab:
gitlab.com/dartmouthrobotics/MetricTools.jl

C.2 `MotionPlanning.jl`

Implementations of PRM*, FST*, Visibility Graphs, RC, and RCRM are provided in `MotionPlanning.jl`.

The library is designed for all configuration spaces and planners to be used on each other by taking advantage of Julia's *multiple dispatch* functionality. This allows for many functions to be generalized for any motion planning algorithm, on any configuration space. The resulting interface is simple, consistent, and without any performance overhead:

```
# Setup space and construct planner
space::Space = PlanarSpace()    # Use any space here
planner::Planner = FST(space)   # Use any planner here

# Find path from start to goal
start::Point = [0, 0]
goal::Point = [1, 1]
path::Array{Point} = plan(planner, start, goal)
display(path)
```

As with `MetricTools.jl`, the package is available on GitLab:
gitlab.com/dartmouthrobotics/MotionPlanning.jl

C.3 `MetricSpaces.jl`

To test motion planning algorithms, space definitions with their corresponding collision checkers and local planners must be implemented. `MetricSpaces.jl` offers these spaces with a common interface usable by the planning algorithms defined in subsection C.2.

Source code is available on GitLab:
gitlab.com/dartmouthrobotics/MetricSpaces.jl

C.3.1 `Planar.jl`

For managing planar spaces, the library provides a simple `PlanarSpace` interface. Obstacles are provided via vertex lists defined in a `.json` file. In order to support complex operations such as the GJK algorithm, all obstacles can be optionally triangulated using the ear-clipping algorithm described in [10]. We provide a custom Julia implementation of this algorithm through `EarClipping.jl`.

C.3.2 `Arms.jl`

For managing the n-revolute arm example, we have a full implementation of revolute arms, inverse kinematics, collision checking, and plotting. For the distance metric, we use `Distances.jl`'s `Chebyshev` metric function. The local planner $\Delta(q_1, q_2)$ uses linear interpolation between q_1 and q_2 with a given step size and runs a collision check on each intermediate configuration.

C.3.3 `ReedsShepp.jl`

For the Reeds-Shepp car, we provide a wrapper around `SimpleCarModels.jl` [26] to implement the local planner and distance metric. The local planner uses the `reedsshepp_waypoints` to acquire a list of configurations along the optimal path and check each configuration for collision. We found that the call to `reedsshepp_waypoints` is relatively expensive compared to other local planners.

C.4 Minimal partition models

For constructing subspaces in configuration space (i.e. for PLR cells), a significant memory cost is associated with explicitly holding the bounds for each cell. The `MotionPlanning.jl` library offers minimal partitioning modules for both binary and grid-based partition schemes.

C.4.1 Binary partitions

Binary partitions can be minimally defined as a tree of partition nodes, where each node stores a splitting threshold, along with some data. In the case of the PLR, this data is simply the coefficients for the local linear regression. The dimension along which the split threshold is applied is inferred by the depth of the tree modulo the dimension of the space. Searching this tree to find the correct leaf node given a valid query vector is trivially a $\log n$ operation, where n is the number of nodes.

C.4.2 Grid partitions

For grid partitions, we construct a grid of equally sized cells along the search space. The grid is stored in a D dimensional matrix, where D is the dimension of the search space. Given a query vector x and D-dimensional vector of lower bounds on the space b_{low} and sizes of each dimension s, indexes into the matrix are calculated along each dimension d as follows:

$$i_d = (v[d] - b_{low}[d] * s[d])$$

This formula allows for fast calculation of indices, providing elegant constant-time lookup.

References

[1] Devin Balkcom et al. “Metric cells: Towards complete search for optimal trajectories”. In: *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2015, pp. 4941–4948.

[2] Jérôme Barraquand and Jean-Claude Latombe. “Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles”. In: *International Conference on Robotics and Automation*. Sacramento, CA, 1991, pp. 2328–2335.

[3] Mukunda Bharatheesha et al. “Distance metric approximation for state-space RRTs using supervised learning”. In: 2014, pp. 252–257.

[4] Joshua Bialkowski et al. “Efficient Collision Checking in Sampling-Based Motion Planning”. In: 2012, pp. 365–380.

[5] Stephen Cameron. “Enhancing GJK: Computing minimum and penetration distances between convex polyhedra”. In: *Proceedings of international conference on robotics and automation*. Vol. 4. IEEE. 1997, pp. 3112–3117.

[6] Tianqi Chen and Carlos Guestrin. “Xgboost: A scalable tree boosting system”. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM. 2016, pp. 785–794.

[7] Hao-Tien Lewis Chiang et al. “RL-RRT: Kinodynamic Motion Planning via Learning Reachability Estimators from RL Policies”. In: *CoRR* abs/1907.04799 (2019).

[8] Peng Cui et al. “A Survey on Network Embedding”. In: *CoRR* abs/1711.08752 (2017).

[9] Robin Deits and Russ Tedrake. “Computing Large Convex Regions of Obstacle-Free Space through Semidefinite Programming”. In: *Workshop on the Algorithmic Foundations of Robotics (WAFR)*. Istanbul, Turkey, Aug. 2014.

[10] David Eberly. “Triangulation by ear clipping”. In: *Geometric Tools* (2008), pp. 2002–2005.

[11] Aleksandra Faust et al. “PRM-RL: Long-range Robotic Navigation Tasks by Combining Reinforcement Learning and Sampling-Based Planning”. In: *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Brisbane, Australia, May 21-25, 2018*. IEEE, 2018, pp. 5113–5120.

[12] David Ferguson and Anthony (Tony) Stentz. *The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-Uniform Cost Environments*. Tech. rep. CMU-RI-TR-05-19. Pittsburgh, PA: Carnegie Mellon University, June 2005.

[13] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. “Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic”. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 2997–3004.

[14] Michael Innes et al. “Fashionable Modelling with Flux”. In: *CoRR* abs/1811.01457 (2018). arXiv: `1811.01457`.

[15] Lucas Janson and Marco Pavone. “Fast Marching Trees: A fast marching sampling-based method for optimal motion planning in many dimensions”. In: *Robotics Research*. Springer, 2016, pp. 667–684.

[16] Sertac Karaman and Emilio Frazzoli. “Sampling-based Algorithms for Optimal Motion Planning”. In: *The International Journal of Robotics Research* 30(7), 846-894 (2011).

[17] Lydia Kavraki et al. “Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces”. In: *IEEE International Conference on Robotics and Automation*. 1996, pp. 566–580.

[18] Anthanasios Krontiis, Andrew Dobson, and Kostas Bekris. “Sparse roadmap spanners”. In: *Proceedings of the workshop on the algorithmic foundations of robotics on Robotics, WAFR 2012* (2012).

[19] Tomás Lozano-Pérez and Michael A. Wesley. “An Algorithm for Planning Collision-free Paths Among Polyhedral Obstacles”. In: *Commun. ACM* 22.10 (Oct. 1979), pp. 560–570.

[20] James D. Marble and Kostas E. Bekris. “Asymptotically Near-Optimal Planning With Probabilistic Roadmap Spanners”. In: *IEEE Transactions on Robotics* 29.2 (2013), pp. 432–444.

[21] Rémi Munos, Leemon C. Baird, and Andrew W. Moore. "Gradient descent approaches to neural-net-based solutions of the Hamilton-Jacobi-Bellman equation". In: *IJCNN*. 1999.

[22] Josiah Putman et al. "Piecewise linear regressions for approximating distance metrics". In: *arXiv* (2020).

[23] Nathan Ratliff et al. "CHOMP: Gradient optimization techniques for efficient motion planning". In: *2009 IEEE International Conference on Robotics and Automation*. IEEE. 2009, pp. 489–494.

[24] D. Chris Rayner, Michael H. Bowling, and Nathan R. Sturtevant. "Euclidean Heuristic Optimization". In: *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. 2011.

[25] J. A. Reeds and L. A. Shepp. "Optimal paths for a car that goes both forwards and backwards." In: *Pacific J. Math.* 145.2 (1990), pp. 367–393.

[26] Edward Schmerling. *SimpleCarModels.jl*. `https://github.com/schmrlng/SimpleCarModels.jl`.

[27] T. Siméon, J.-P. Laumond, and C. Nissoux. "Visibility-based probabilistic roadmaps for motion planning". In: *Advanced Robotics* 14.6 (2000), pp. 477–493.

[28] Russ Tedrake et al. "LQR-trees: Feedback Motion Planning via Sums-of-Squares Verification". In: *I. J. Robotics Res.* 29.8 (2010), pp. 1038–1052.

[29] Weifu Wang, Devin J. Balkcom, and Amit Chakrabarti. "A fast online spanner for roadmap construction". In: *I. J. Robotics Res.* 34.11 (2015), pp. 1418–1432.

[30] *XGBoost.jl*. `https://github.com/dmlc/XGBoost.jl`.