

Topology-inspired Probabilistic Path Replanning in Dynamic Environments

Richard Fisher
558921

Supervised by:
Benjamin Rosman
Vladimir Ivan
Pravesh Ranchod

A thesis submitted to the Faculty of Science, University of the Witwatersrand, in
fulfilment of the requirements for the degree of Master of Science.

School of Computer Science & Applied Mathematics
University of the Witwatersrand
June 18, 2018

Abstract

Path replanning in high dimensional dynamic environments is critical to the success of interactive and reactive robotic agents. State of the art replanning algorithms typically extend sampling-based methods such as rapidly-exploring random trees (RRT) or probabilistic roadmaps (PRM). However, the speed of replanning in complex configuration spaces is relatively slow, which limits the effectiveness of robotic agents in highly dynamic environments.

This thesis proposes DRM-connect, a novel generalisation of the PRM and RRT-connect algorithms, which carries out replanning in dynamic environments by executing graph searches over an underlying graph $\mathcal{G}$, using lazy collision checking. If a path through the graph is not found, DRM-connect will repair the graph using a novel extension to RRT-connect, which we call PRM-connect.

Additionally, we investigate using an approximate Reeb graph as the underlying graph $\mathcal{G}$, which attempts to capture the underlying topology of the task manifold from prior experience. DRM-connect is tested with both a Reeb graph and naïve graph in a 2-D domain and compared to RRT, while DRM-connect with a Reeb graph is tested in three 7-D domains, and compared to RRT-connect. Through simulation we show that the combination of DRM-connect and a Reeb graph typically outperforms both RRT/RRT-connect and DRM-connect with a naïve graph in terms of replanning times, with minimal impact on the length of the solution path.

Declaration

I, Richard Fisher, declare that this Thesis is my own, unaided work. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

A handwritten signature in black ink, appearing to be 'R Fisher', with a long, sweeping horizontal stroke extending to the right.

June 18, 2018

Contents

1	Introduction	3
2	Related Work	5
2.1	Path Planning	5
2.1.1	Probabilistic path planning	6
2.1.2	Rapidly exploring random trees	7
2.1.3	RRT-connect	8
2.1.4	RRT*	9
2.1.5	RRT*-connect	9
2.1.6	Probabilistic roadmap planners	9
2.2	Replanning Approaches	11
2.3	Conclusion	12
3	Graph-based Replanning	13
3.1	Background	13
3.2	Dynamic roadmap planning with lazy collision checking	14
3.3	PRM-connect	14
3.4	DRM-connect	15
3.5	Experiments	16
3.5.1	The DOORS domain	17
3.6	Conclusion	19
4	Planning with Reeb Graphs	20
4.1	Introduction	20
4.2	Background	20
4.2.1	Computational Topology	20
4.2.2	Trajectory classification	22
4.2.3	Reeb graphs	22
4.3	DRM-connect with Reeb graphs	23
4.3.1	Embedding Reeb graphs	24
4.4	Experiments	25
4.4.1	The DOORS domain	26
4.4.2	7-D domains	30
4.4.3	The BAR domain	32
4.4.4	The CUP domain	34
4.4.5	The BOX domain	35
4.5	Comparison of 7-D domains	39
4.6	Reeb graph discussion	39
4.7	Conclusion	41
5	Conclusion and Future Work	43
5.1	Conclusion	43
5.2	Future work	43

Chapter 1

Introduction

Robotic path planning is the operation of converting high-level specifications (start and end positions, locations of obstacles, cost functions, for example) into low-level descriptions of how to move a robot to satisfy these specifications.

This is often posed as the piano mover’s problem, where the goal is generating a collision-free path along which to move a piano from one position to another. A generalisation of this is generating a collision-free path through the configuration space $\mathcal{C}$ of a robot (the set of all positions of the robot’s joint or actuation parameters, usually either the joint angles or physical position of the robot), constrained by its own kinematics [LaValle, 2006].

Once the obstacles or kinematics become more complex, or the number of degrees of freedom grow, elegant analytic solutions to the path planning problem become impossible or infeasible to calculate. These solutions typically involve explicit modeling of the free regions of the configuration space, which is challenging for obstacles with complex shapes or kinematics. For these reasons, sample-based methods were developed to probe the collision-free areas of the configuration space without the need for explicit construction. These methods concern themselves only with the random selection of states to query, and expect that whether these states are in collision can be easily checked. Common sample-based methods build either trees or graphs across the configuration space, and use these to generate viable or optimal paths [LaValle, 2006].

Since the samples are generated randomly, the expected time to solve path planning problems grows exponentially as the number of dimensions increases [Thrun et al., 2005]. Once a path has been computed, and execution has begun, the environment may change. If this results in the current path no longer being viable, the path needs to be replanned from scratch. The time taken for this replanning will be of the same order as the original planning, which could be significant. In any application where the environment changes frequently (e.g. interaction with a human), replanning will have to be carried out as frequently, which will be prohibitively expensive.

This is compounded by the increased complexity of planning in high dimensional environments or those with complex kinematics. This motivates for an improvement of the speed at which high dimensional path replanning is carried out.

Many sample-based methods, and RRT (Rapidly-exploring random trees, see Section 2.1.2) variants particularly, attempt to modify the probabilities with which samples are chosen, as a way to leverage past knowledge of the domain and task [Bruce and Veloso, 2002; Ferguson et al., 2006; Gayle et al., 2007; Zucker et al., 2007].

Alternatively, past experience can be leveraged in a way that directly draws on prior experience of tasks in the same or similar environments. An important question for replanning is whether a change in the environment requires just a slight adaptation of the current solution, or whether a large change is

required. Reasoning about the topological structure of the space, and specifically about how the changes in the environment affect the path-connectivity of the space, can provide this information.

We propose DRM-connect, a bi-directional extension of the DRM algorithm [Leven and Hutchinson, 2000] (dynamic roadmap planning, see Section 3.1), which uses a graph constructed from prior experience from the same or similar tasks to improve the speed of replanning. Further, we investigate one such graph, a Reeb graph, which recovers topological information about the space in which the robot operates from prior experience. The Reeb graph (in conjunction with DRM-connect) should encode local alternatives to solving tasks in dynamic environments, as well as more global alternatives. Thus the hierarchical reasoning about path-connectedness is reduced to a single level by the DRM-connect algorithm.

We expect that this algorithm should replan significantly faster in high dimensional spaces than algorithms which replan from scratch. We test DRM-connect against RRT and RRT-connect in a variety of domains, and show that in many situations the speed of replanning is significantly improved. Additionally, we test DRM-connect with both a Reeb graph and a naïve graph to evaluate how effectively the Reeb graph captures information about the path-connectivity of the task manifold (the manifold formed by the reachable set of states of the robot for a specific task).

The layout of the rest of this document is as follows. Chapter 2 formally poses the path planning and replanning problems and introduces existing methods to solve them. Chapter 3 introduces DRM-connect and tests it with a naïve graph. Chapter 4 proposes a more sophisticated graph to use with DRM-connect, namely a Reeb graph. This draws on several topological approaches to representing the connectivity of the task manifold. Finally, Chapter 5 concludes the thesis and presents avenues for future work.

Chapter 2

Related Work

2.1 Path Planning

The path planning problem is often stated as the *Piano Mover's Problem*: “How should a piano in a cluttered environment be moved from a start position to a final position without colliding with objects?” The goal of path planning can be defined more generally as the task of generating a collision-free path through a robot's configuration space, while respecting its kinematics. General planning has several important definitions [LaValle, 2006]:

1. **State** - The state space encompasses all possible situations which could arise in the environment. These states typically include positions, orientations, velocities and accelerations. The state space may be continuous or discrete, and distinct algorithms usually operate in each type. In this work, the state space will typically represent the degrees of freedom of robots ($\mathbb{R}^n$).
2. **Time** - Time may be implicitly or explicitly represented, although its major use in path planning (in the absence of dynamics) is simply to order actions. In the case of either dynamic environments (unpredictable changes in obstacle positions) or dynamic systems (time derivatives included in states), time is used as a basis for state or environment changes.
3. **Actions** - The plan (which is the solution to the planning problem) describes a set of actions which transform the state from the start state to the end state. These actions attempt to change the state. The state transition model (how the states change under actions) must be embedded in the planning algorithm.
4. **Initial and goal states** - The initial and goal states indicate the states between which the planner attempts to create a path, by specifying actions.
5. **A criterion** - A successful planning solution is one which satisfies the problem criterion. This criterion is either feasibility (when the plan is executed, the final state is the goal state) or optimality (the plan is feasible, and the plan is optimal in some other sense - usually execution cost or solve cost).
6. **A plan** - The plan describes the strategy or set of actions to take to move from the start state to the goal. The plan can depend on the state (a reactive plan) or can simply describe a set of actions to take. When the state transition model is simple, the plan may just be a set of states. This is referred to as trajectory or path planning, where the resulting plan is a trajectory or path, respectively.

Planning algorithms use points 1 - 5 above to generate a plan. An ideal path planning algorithm should be *complete*, which means it should report (in finite time) whether a solution exists, and if so, what that solution is.

Figure 2.1 shows examples of several of the concepts introduced above. The state space (here also the configuration space) is a 2-D one, with initial and goal states of q_s and q_g respectively. The actions and plan are the same in this case, and are a set of nodes and edges connecting the initial and goal states.

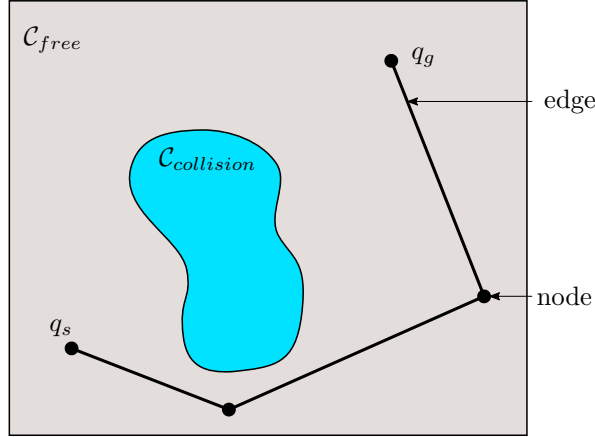


Figure 2.1: An example of a configuration space $\mathcal{C}$, comprised of the union of $\mathcal{C}_{free}$ and $\mathcal{C}_{collision}$. The start and goal point are q_s and q_g respectively. A solution path is created by connecting these with a set of nodes and edges while avoiding $\mathcal{C}_{collision}$.

The criterion here is simply that the path be feasible. The entire configuration space, $\mathcal{C}$, is the union of $\mathcal{C}_{collision}$ (regions of the configuration space which correspond to collisions in the task space) and $\mathcal{C}_{free}$ (regions of the configuration space which correspond to collision free regions of the task space). The task space is the environment in which the robot operates.

Path planning methods fall into two categories [LaValle, 2006]:

- *Single-query* methods aim to find a solution path for a single query.
- *Multiple-query* methods aim to answer multiple queries, so additional effort is usually expended to construct some representation of feasible partial paths, which can be reused.

Early path planning approaches aimed to be complete, either by executing an A* style graph search (an extension of Dijkstra's algorithm which prioritises the nodes to be explored using a heuristic of their distance to the goal) over a discrete domain, or cell decomposition-based graph or tree building over a continuous domain [LaValle, 2006]. These methods require explicit representations of the free configuration space ($\mathcal{C}_{free}$), and are often prohibitively time consuming in higher dimensional spaces.

2.1.1 Probabilistic path planning

Sampling-based methods are a collection of path planning algorithms which avoid explicit representations of the free configuration space. Rather, a black-box collision-detector is assumed to exist, which can reliably determine whether a specific configuration is in collision. The free configuration space is probed using a sampling scheme α which can be random or deterministic. If the sampling scheme follows a probabilistic approach, then the sampling-based method is said to be a probabilistic method [LaValle, 2006].

Probabilistic methods have been shown to perform well for high dimensional systems and those with complex dynamics. For sampling-based and probabilistic methods, the requirement of completeness is relaxed. Rather, two separate completeness criteria are important [LaValle, 2006]:

- A *resolution complete* algorithm will return a solution, if it exists, in finite time. However, if no solution exists, it may run forever.
- A *probabilistically complete* algorithm will return a solution with probability one as time tends to infinity. It has no way of determining whether a solution exists.

Multiple-query planners are recommended when the cost of pre-computation can be amortised over a number of planning episodes. For example, an industrial pick-and-place robot which continuously operates in the same environment (with possibly different tasks) would benefit from the reduced replanning time, at the cost of an expensive pre-computation step. In situations where a single planning task (or several) is expected, the overhead cost of the pre-computation is often too great, and a single-query planner is appropriate.

The most popular single query feasible planner is called Rapidly exploring random trees, a tree-based method which is detailed below.

2.1.2 Rapidly exploring random trees

Rapidly exploring random trees (RRT) is a popular single-query probabilistic path planning algorithm [LaValle, 2006]. A search tree ($\mathcal{G}$) is constructed over the configuration space from the start point (q_s) until the goal point (q_g) is reached. This tree is constructed by selecting points according to some random sequence α , which are then added to the nearest node in the tree (q_n). The procedure for RRT is given in Algorithm 1.

Algorithm 1 RRT(q_s, q_g) [LaValle, 1998]

```

1:  $\mathcal{G}.\text{init}(q_s)$ ;
2: repeat
3:    $q_r \leftarrow \text{UNIFORM-SAMPLE}()$ ;
4:    $q_n \leftarrow \text{NEAREST}(S(\mathcal{G}), q_r)$ ;
5:    $q_s \leftarrow \text{STOPPING-CONFIGURATION}(q_n, q_r)$ ;
6:   if  $q_s \neq q_n$  then
7:      $\mathcal{G}.\text{add\_vertex}(q_s)$ ;
8:      $\mathcal{G}.\text{add\_edge}(q_n, q_s)$ ;
9:   end if
10: until  $q_s = q_g$ 
11: return  $\mathcal{G}$ 

```

Before a newly selected point q_r (line 3) is added to the tree, the line segment between it and the nearest node on the tree q_n (line 4) is first checked for collision (line 5). If there is a collision, the stopping point q_s (line 5), which is the last collision-free point on the line from q_n to q_r , is added to the tree (lines 7 and 8), as long as this results in a new point. $S(\mathcal{G})$ returns the nodes of the graph $\mathcal{G}$.

The STOPPING-CONFIGURATION algorithm (line 5) extends a line from q_n towards q_r . A set of points along this line are tested for collisions. The last collision-free point along this line (or q_n if no collisions exist) is returned. The number of points to test is typically chosen such that the distance between these is smaller than the smallest expected object in the environment. For this reason the line from q_n to q_r is often truncated some fixed distance from q_n (this distance is called the step size, r).

In practice, the goal point (q_g) is periodically used instead of a sample drawn from the sequence or distribution α . This goal-bias is given by p_g . This allows the planner to try connect the tree to the goal point, to complete the path planning query. This goal-bias has to be experimentally set, and will strongly influence the number of samples required to solve a query [LaValle, 2006].

By increasing this bias value, the RRT becomes more greedy, which may result in a quicker solve time, but may also cause the tree to propagate into a local minimum, which will increase the solve time. If $p_g = 1$ (the goal point is always sampled) the tree will propagate directly to the goal. If there is a clear path, this gives an optimal solve time, otherwise, if an obstacle lies in the way, no solution will ever be found. If $p_g = 0$ (the goal point is only sampled when $\alpha(i) = q_g$) then the RRT will almost surely never reach the goal (if the search space is continuous), although it will get arbitrarily close as the number of

samples taken tends to infinity [LaValle, 2006].

The RRT algorithm is probabilistically complete, meaning that if a solution exists, the probability of finding it converges to one as the number of samples tends to infinity.

The RRT algorithm works well for tasks where the approach to the goal point is fairly open. However, for tasks which have narrow gaps (for example the bug trap or channel environments [LaValle, 2006]) the performance of RRT is poor. Bi-directional planning algorithms such as RRT-connect have been developed to deal with this.

2.1.3 RRT-connect

In order to improve path planning efficiency, the RRT algorithm has been extended by Kuffner and LaValle [2000] to a bi-directional search. It works by incrementally building two RRTs, rooted at the start and end points. These trees grow towards randomly selected configurations, and are also biased to growing towards the nodes of the other tree. A simple greedy heuristic guides the trees towards one another, while also exploring the free configuration space. The RRT-connect algorithm has been designed for problems which contain no differential constraints. The RRT algorithm (by rooting the tree at the start point) effectively moves forward in time, for which simulating differential constraints is trivial, whereas RRT-connect also propagates from the end point, which is moving back in time, which would require solving the differential constraints [Kuffner and LaValle, 2000].

Algorithm 2 describes the procedure for building the two RRTs. The task start and end points q_s and q_g are supplied, and a path between them is returned. First, trees $\mathcal{T}_a$ and $\mathcal{T}_b$ are rooted at the start point q_s and the end point q_g (lines 1 and 2). Then, a point is sampled (line 4) and added to $\mathcal{T}_a$ (line 5). The point is added in the same way, but using EXTEND, given in Algorithm 4. If the connection to $\mathcal{T}_a$ was successful, $\mathcal{T}_b$ is then grown towards this point using CONNECT (line 6), given in Algorithm 3. The two trees are then switched (line 10), and the process is repeated until either a path is found between q_s to q_g or K iterations are exceeded.

Algorithm 2 RRT-CONNECT (q_s, q_g) [Kuffner and LaValle, 2000]

```

1:  $\mathcal{T}_a.\text{init}(q_s)$ ;
2:  $\mathcal{T}_b.\text{init}(q_g)$ ;
3: for  $k = 1$  to  $K$  do
4:    $q_r \leftarrow \text{RANDOM-CONFIG}()$ ;
5:   if not ( $\text{EXTEND}(\mathcal{T}_a, q_r) = \text{Trapped}$ ) then
6:     if ( $\text{CONNECT}(\mathcal{T}_b, q_r) = \text{Reached}$ ) then
7:       return  $\text{PATH}(\mathcal{T}_a, \mathcal{T}_b)$ 
8:     end if
9:   end if
10:   $\text{SWAP}(\mathcal{T}_a, \mathcal{T}_b)$ 
11: end for
12: return Failure
```

The CONNECT algorithm takes a tree $\mathcal{T}$ and a state q and repeatedly calls the EXTEND algorithm (line 2) to grow the tree $\mathcal{T}$ towards q until it can no longer advance any closer (line 3). The EXTEND algorithm also takes $\mathcal{T}$ and q as inputs, and takes a step towards q from the point nearest to it on $\mathcal{T}$ (line 1), checking collisions along the segment between q_{near} and q (line 2). EXTEND returns q_{new} , the furthest collision-free point along this segment. This is very similar to the STOPPING-CONFIGURATION step in Algorithm 1.

NEAREST-NEIGHBOUR takes a point q and a tree $\mathcal{T}$ and returns the node on the tree $\mathcal{T}$ closest to q . NEW-CONFIG tests collisions along the trajectory from q_{near} to q . The last collision free state is stored in q_{new} .

Algorithm 3 CONNECT($\mathcal{T}, q$) [Kuffner and LaValle, 2000]

```

1: repeat
2:    $S \leftarrow \text{EXTEND}(\mathcal{T}, q)$ ;
3: until not ( $S = \text{Advanced}$ )
4: return  $S$ 

```

Algorithm 4 EXTEND($\mathcal{T}, q$) [Kuffner and LaValle, 2000]

```

1:  $q_{\text{near}} \leftarrow \text{NEAREST-NEIGHBOUR}(q, \mathcal{T})$ 
2: if NEW-CONFIG( $q, q_{\text{near}}, q_{\text{new}}$ ) then
3:    $\mathcal{G}.\text{add\_vertex}(q_{\text{new}})$ ;
4:    $\mathcal{G}.\text{add\_edge}(q_{\text{near}}, q_{\text{new}})$ ;
5:   if  $q_{\text{new}} = q$  then
6:     return Reached;
7:   else
8:     return Advanced;
9:   end if
10: end if
11: return Trapped;

```

The planning methods discussed above are single-query methods, which excel at single-shot planning. In cases where multiple queries in the same environment are expected, the overhead cost of pre-computation can be amortised over the future planning queries.

2.1.4 RRT*

RRT* is an asymptotically optimal extension to RRT, proposed by Karaman and Frazzoli [2010]. The probability of finding an optimal path tends to 1 as the number of samples tends to infinity. This works similarly to RRT, except that each time an edge is added to the tree, and attempt is made to connect all nodes within a local region to this new point. If the resulting connections give shorter paths to the root node for any of these nodes, this connection is kept and the old connection discarded. The effect of this is that the distance from the root to leaf nodes of the RRT* will in general be shorter than an equivalent RRT. Thus, shorter paths will be returned by RRT*. The trade-off for the improved path-length is that additional collision checks and edge-reordering is required. Since the explored nodes between RRT and RRT* (under the same initial conditions) are always identical, a solution is always found after the same number of samples are drawn. However, RRT* requires additional operations to handle each sample, and so will always be slower. For the 2-D task tested by Karaman and Frazzoli [2010] RRT* required about 30 times more time to solve the task.

2.1.5 RRT*-connect

In the same way that RRT has a bi-directional extension, RRT* has been extended to RRT*-connect [Jordan and Perez, 2013]. Like RRT*, RRT*-connect is probabilistically complete and asymptotically optimal.

RRT*-connect is essentially the same as RRT-connect, except that the two trees are rewired in the same way as RRT*. Again, from the same initial conditions, RRT*-connect will require the same number of samples to solve a query, while requiring additional steps to test for shorter connections and respond to them.

2.1.6 Probabilistic roadmap planners

The most commonly used form of these multiple-query planners is the Probabilistic Roadmap Planner (PRM) [Kavraki et al., 1996]. PRMs consist of two phases. The first phase is an offline phase, where a set

of vertices (corresponding to collision-free configurations) are joined by edges to create a graph of short valid trajectories, which cover the configuration space. The edges are constructed using a local planner. Algorithm 5 details the procedure for constructing a roadmap.

Algorithm 5 BUILD_ROADMAP [LaValle, 2006]

```

1:  $\mathcal{G}.\text{init}()$ ;  $i \leftarrow 0$ ;
2: while  $i < N$  do
3:   if  $\alpha(i) \in \mathcal{C}_{free}$  then
4:      $\mathcal{G}.\text{add\_vertex}(\alpha(i))$ ;  $i \leftarrow i + 1$ 
5:     for all  $q \in \text{NEIGHBOURHOOD}(\alpha(i), \mathcal{G})$  do
6:       if ((not  $\mathcal{G}.\text{same\_component}(\alpha(i), q)$ ) and ( $\text{CONNECT}(\alpha(i), q)$ )) then
7:          $\mathcal{G}.\text{add\_edge}(\alpha(i), q)$ 
8:       end if
9:     end for
10:  end if
11: end while
12: return  $\mathcal{G}$ 

```

Recall that α is a sequence of samples (random, pseudo-random or deterministic) in the configuration space. The parameter N is the number of vertices to add to the graph. Samples are drawn from the sequence α until the sample is in $\mathcal{C}_{free}$ (line 3). This sampled configuration is added as a vertex to graph $\mathcal{G}$ and i is incremented (line 4). Note that i is only incremented if the vertex is added to the graph. This ensures that N vertices are always added to the graph. This new vertex is then connected to all vertices in $\mathcal{G}$ within some neighbourhood (lines 5 to 7). CONNECT is given in Algorithm 3.

The second phase is the query phase. When the roadmap is queried for a new path, the start and end configurations (q_s and q_g respectively) are added as nodes to the graph (if they don't already exist) and connected to the graph using a local planner. A graph search is then conducted over the roadmap to return the shortest path.

The roadmap is constructed in a very similar manner to RRT, in that a graph is constructed incrementally to cover the configuration space. However, RRT grows a tree (only a single connection to the sampled configuration is allowed) while PRM grows a (possibly) cyclical graph, since the sampled point is connected to all points in the existing graph in some neighbourhood. Additionally, RRT terminates when q_g is reached, while PRM adds a predefined number of vertices.

Planning on a chip

An extension to PRMs for realtime high dimensional planning and replanning has been deployed to an FPGA with some optimisations, yielding planning times for a 6 degree of freedom arm of less than 1 millisecond [Murray et al., 2016].

The factors contributing significantly to the fast replanning are precomputation and the use of specialised hardware for collision detection. During a costly precomputation step (which only has to be carried out once for each robot), a standard roadmap is built which covers the $\mathcal{C}$ space. Each edge in this roadmap (which corresponds to some motion of the robot) is mapped to voxels in the task space through which this motion occurs. A logical expression is then constructed for each edge in the PRM which, when supplied with a voxel in the task space, returns true if the swept volume of the edge would cause a collision with that voxel.

At run time, the robot senses the environment and constructs an occupancy grid (which voxels contain obstacles). Each occupied voxel is supplied to the logical expressions of all edges, which are tested in

parallel on the FPGA. The edges which return true for obstacles are discarded from the roadmap, and a standard graph search is carried out over the remaining subgraph to return a trajectory.

2.2 Replanning Approaches

In dynamic systems (systems containing obstacles which appear, disappear and move *unpredictably*), a precomputed viable path may be obstructed by a dynamic obstacle during execution of the plan. This requires a new plan to be generated, which is referred to as *replanning*. The naïve approach is to solve the problem from scratch again, using the same goal, but the current position as the start point. Assuming that the environment changes gradually, much of the structure present when the plan was previously constructed still exists. This naïve approach unnecessarily discards all the discovered knowledge about traversing through this environment.

The RRT algorithm has been extended in many ways to improve the solve time for replanning in dynamic environments by retaining some path information from the previous paths, some of which are discussed below.

Execution extended RRTs (ERRTs) provide a method of drawing on past experience in a particular environment by recording waypoints used in successful prior tasks [Bruce and Veloso, 2002]. When a successful plan is found, all the nodes from the solution plan are added to the waypoint cache (w_{cache}). This cache has a limited size, so new waypoints randomly overwrite existing ones. This assumes that the environment has changed only slightly, so that recent experience gives insight into the new environment.

The RRT algorithm is adjusted so that one of these waypoints is selected at random with probability p_w . This is handled similarly to the sampling of the goal point with probability p_g . The addition of a KD-tree search improves the solve times for the tasks specified, by some margin. In one example, using ERRT instead of a reactive scheme responding to obstacles without lookahead for real time RoboCup F180 obstacle avoidance resulted in a 40% increase in travel speed, indicating improved rates of replanning and optimality of paths [Bruce and Veloso, 2002]. ERRT uses Algorithm 1, but calls ERRT-SAMPLE instead of UNIFORM-SAMPLE, which is shown in Algorithm 6.

Algorithm 6 ERRT-SAMPLE(w_{cache}, q_g, p_g, p_w)

```

1:  $p = \text{UniformRandom in } [0.0, 1.0];$ 
2:  $i = \text{UniformRandom in } [0, \text{LENGTH}(w_{cache}) - 1];$ 
3: if  $0 < p < p_g$  then
4:   return  $q_g$ ;
5: else if  $p_g < p < p_g + p_w$  then
6:   return  $w_{cache}[i];$ 
7: else
8:   return UNIFORM-SAMPLE();
9: end if
```

Using ERRT for replanning means that samples are biased towards the region where the previous tasks succeeded. However, all additional information (in the form of the rest of the tree) is discarded.

Dynamic RRT (DRRT) seeks to improve on this by repairing an RRT tree (rather than discarding it) when any of the nodes become invalid due to changes in the environment [Ferguson et al., 2006]. This is done simply by removing all invalid nodes and all child nodes (including attached edges) from the tree. The remaining tree is used with the standard RRT algorithm to complete the task. Further, the regions around the now invalid points are biased for selection, to attempt to generate a similar path to the previous solution path (which is analogous to the waypoint cache of ERRT).

This approach produces significantly reduced runtimes, compared to ERRT, for a single robot agent in an exploration environment with randomly appearing obstacles. However, since all child nodes of the nodes in collision are discarded, significant information about solving the task is also discarded.

Algorithms to improve on this have been proposed as Lazy Reconfiguration Forests and Multipartite RRTs.

Lazy Reconfiguration Forests (LRF) are an extension to DRRT, designed to maintain knowledge of how to traverse the explored regions of the configuration space for motion planning with multiple tasks in dynamic environments. LRFs maintain a forest of trees whenever an obstacle causes any of the nodes to be in collision by temporarily invalidating the nodes which are in collision. The child nodes of these invalidated nodes are now the roots of trees added to the maintained forest. Further planning works the same as RRT, except that the roots of the forest of trees are biased for sampling. This feature makes them well suited to replanning in dynamic environments [Gayle et al., 2007].

Multipartite RRTs (MP-RRTs) combine the algorithms of ERRT and DRRT to further leverage past experience in dynamic environments [Zucker et al., 2007]. When a path becomes invalid, the invalid nodes and edges are removed, and the now disconnected subtrees are stored. The (possibly) new q_{init} attempts to join to the old root tree. If this fails the old root tree is stored with the other subtrees. MP-RRT then samples points randomly from the roots of the stored subtree with probability p_{forest} . It still samples the goal point with probability p_g , otherwise it samples the space uniformly. MP-RRT is tested in several low dimensional dynamic environments, and significantly outperforms RRT and ERRT in solve time, and marginally outperforms DRRT.

RRTx is the first asymptotically optimal sampling-based replanning algorithm [Otte and Frazzoli, 2016]. It roots a tree at the goal and builds the tree using a similar methodology to RRT*, and rewires the tree when a change in the environment is detected. Edges in the tree which are in collision are temporarily invalidated (like LRF).

Sánchez and Latombe [2003] have extended the PRM algorithm to make it single-query, bi-directional lazy in collision checking (SBL). It also has the property of adaptiveness, where longer steps are taken in more open spaces, and shorter steps in more cluttered areas. Two trees are rooted at the start and end points and expanded. This expansion only places milestones (which are checked for collisions), and adjusts to the density of obstacles in the local region. If the two trees can be joined, then the entire path from start to goal is tested for collisions (using a priority queue which either marks segment safe or in collision, or assigns a probability of it being safe). This priority queue allows the collision checking to focus on edges which are most likely to contain collisions, saving the overall time spent on collision checking. SBL has been shown to handle high dimensional problems with run times of a few seconds to minutes.

2.3 Conclusion

Popular approaches for replanning in dynamic environments extended from RRT have been introduced in this chapter. While many of these methods are effective replanners, they maintain only local replanning information, and are unable to detect whether a topologically different path is required to complete the replanning task. One method of maintaining knowledge of these separate classes of trajectories is to make use of graph-based planning techniques, which are covered in Chapter 3.

Chapter 3

Graph-based Replanning

As discussed in the previous section, many extensions to probabilistic methods exist which cater for replanning in dynamic environments. Many of these extend RRT, and as such work well for single-query replanning. In all of the replanning algorithms discussed, significant amounts of prior experience is discarded whenever replanning is required due to changes in the environment. This chapter introduces a novel approach to graph-based replanning which attempts to overcome this, namely DRM-connect (see Section 3.4).

The background theory required for DRM-connect is covered in Section 3.1. The search and collision check component of DRM-connect (DRM-lazy) is proposed in Section 3.2, while the repair component (PRM-connect) is introduced in Section 3.3. Experimental validation for DRM-connect is carried out for a 2-D domain in Section 3.5.

3.1 Background

The PRM algorithm (see Section 2.1.6) has been extended for dynamic replanning scenarios by maintaining and repairing the roadmap [Leven and Hutchinson, 2000; Jaillet and Siméon, 2004] and is termed dynamic roadmap planning (DRM). Other approaches to extending PRM for dynamic replanning (e.g. Van Den Berg and Overmars [2005]) assume that the motion of the dynamic obstacles is known at planning time. As discussed in Section 2.2, we define dynamic environments to be those where the obstacles move *unpredictably*, so a model of their motion is not available. The approach detailed here is from Leven and Hutchinson [2000], since it is more similar to the algorithms we propose. The work by Jaillet and Siméon [2004] proposes an edge-labeling approach to test for new collisions at each time step more efficiently, rather than the task- to configuration-space mapping used by Leven and Hutchinson [2000].

Like PRM, there are two steps in DRM. The first is a pre-planning phase where a roadmap is built. This probabilistic roadmap is pre-computed, assuming no obstacles. A rectangular cell decomposition of the workspace is then computed, and each cell is mapped to the nodes and edges of the roadmap which correspond to collisions in these decomposed cells. When obstacles in the workspace appear, move or disappear, this mapping makes it efficient to infer which edges and nodes are affected by the change in the environment.

During planning queries (or replanning), nodes and edges are temporarily invalidated when obstacles move through their mapped cells. Queries are then carried out as A*-style graph searches. If there is no valid path through the roadmap, an RRT-connect algorithm is used.

This algorithm has been tested on 4 and 7 DOF manipulators and a 17 DOF humanoid [Kallman and Mataric, 2004]. DRM ranged from 1.1 times faster than RRT-connect (for the 17 DOF humanoid) to 8.6 times faster for the 4 and 7 DOF manipulators. The pre-computed roadmap is constructed without prior knowledge of the kinds of tasks it will be used to solve, and as such, leverages no past experience.

Before introducing DRM-connect, we require two other algorithms, which are presented below. Section 3.2 introduces the DRM-lazy algorithm, an extension to DRM, which carries out the graph searches with lazy collision checking for DRM-connect. When DRM-lazy fails to find a valid path through the graph, PRM-connect, presented in Section 3.3, is responsible for repairing the underlying graph.

3.2 Dynamic roadmap planning with lazy collision checking

We propose a novel extension to DRM, namely dynamic roadmap planning with lazy collision checking (DRM-lazy), a component of our proposed DRM-connect algorithm (see Section 3.4). When provided with start and end points q_s and q_g , this algorithm attempts a graph search on the supplied graph, with lazy collision checking. If a valid path is found, this path is returned, otherwise PRM-connect (see Section 3.3) is called, which attempts to repair the graph.

DRM-lazy (given in Algorithm 7) differs from DRM in that collisions are checked on demand (as each edge is traversed in the solution path), rather than using a mapping between the task space and configuration space. This is done to reduce the amount of computation required to update the mapping when new edges are added to the graph (which is expected to happen frequently in the DRM-connect framework, since the graph is updated online when a valid path through the graph can't be found). DRM-lazy searches a graph for the shortest path (line 2), and checks for collisions on the path found (line 3). If there are invalid edges, these are removed from the graph (line 10), and a new path is found. This is repeated until either a valid path is found, or q_s and q_g lie in disconnected subgraphs. This is conceptually similar to a standard graph search, except that we don't assume that all the edges are collision free, and only test them when they appear on the shortest path.

Algorithm 7 DRM-LAZY ($\mathcal{G}, q_s, q_g$)

```

1: loop
2:    $p \leftarrow \text{SHORTEST-PATH}(\mathcal{G}, q_s, q_g)$ ;
3:    $E_{\text{collisions}} \leftarrow \text{TEST-COLLISIONS}(p)$ ;
4:   if  $p = \emptyset$  then
5:     return false
6:   end if
7:   if  $E_{\text{collisions}} = \emptyset$  then
8:     return true, p
9:   else
10:     $\mathcal{G}.\text{remove\_edges}(E_{\text{collisions}})$ ;
11:  end if
12: end loop

```

The step TEST-COLLISIONS should take a path p and return whether all edges along this path are collision free. In practice, a speed improvement can be achieved by retaining a list of edges which have been checked, since in the present form a single edge may be evaluated for collisions multiple times.

3.3 PRM-connect

DRM-connect also requires the algorithm PRM-connect, detailed in Algorithm 8. This is a novel adaptation of RRT-connect and PRM. PRM-connect is responsible for repairing the graph when DRM-lazy is unable to find a valid path through the graph. This leverages the bi-directional aspects of RRT-connect and applies them to PRM. This allows for subgraphs, disconnected by obstacles, to be rejoined during planning.

This algorithm starts from the disconnected subgraphs containing q_s and q_g respectively, and enlarges the subgraphs until they're joined together. This follows the process of RRT-connect very closely, except

that instead of expanding trees, we’re expanding graphs. All the added nodes will form trees, although the roots will be on the original subgraphs.

Initially, decompose the graph $\mathcal{G}$ into the subgraphs containing q_s , q_g and all nodes which can’t be reached from either point (lines 1 to 3). Then, sample a new point (line 5). The sampling scheme could perhaps be biased - see below. Extend the first subgraph ($\mathcal{G}_a$) towards the sampled point (line 6), using EXTEND (Algorithm 4). If this extension does not hit an obstacle, then attempt to join $\mathcal{G}_b$ to this point (line 7), using the CONNECT algorithm (Algorithm 3). Repeat this until this extension is ‘trapped’ (an obstacle or the target point is reached). If the two graphs are joined, then return the union of the two (line 8). Otherwise, switch $\mathcal{G}_a$ and $\mathcal{G}_b$ and repeat the process (line 12).

CONNECT and EXTEND (Algorithms 3 and 4) are both detailed for an input tree $\mathcal{T}$, and they’re called in PRM-CONNECT with a graph $\mathcal{G}$. However, both CONNECT and EXTEND respectively have identical behaviour for both tree and graph inputs. q_{new} is calculated by EXTEND (see Algorithm 4).

Speed-ups can potentially be achieved by biasing the sampling towards nodes on all other subgraphs, in the algorithm BIASED-SAMPLE. This would be beneficial in scenarios where the majority of the graph is still valid, but q_s and q_g are separated from the rest of the graph. In this case the two graphs containing q_s and q_g ($\mathcal{G}_a$ and $\mathcal{G}_b$) can quickly be reconnected to the graph containing the remaining nodes ($\mathcal{G}_c$).

Algorithm 8 PRM-CONNECT ($\mathcal{G}, q_s, q_g$)

```

1:  $\mathcal{G}_a$ .init(subgraph( $\mathcal{G}, q_s$ ));
2:  $\mathcal{G}_b$ .init(subgraph( $\mathcal{G}, q_g$ ));
3:  $\mathcal{G}_c$ .init(subgraph( $\mathcal{G}, \neg q_s, \neg q_g$ ));
4: for  $k = 1$  to  $K$  do
5:    $q_{rand} \leftarrow$  BIASED-SAMPLE();
6:   if not (EXTEND( $\mathcal{G}_a, q_{rand}$ ) = Trapped) then
7:     if CONNECT( $\mathcal{G}_b, q_{new}$ ) = Reached then
8:       return  $V(\mathcal{G}_a, \mathcal{G}_b)$ ;
9:     end if
10:  end if
11:  SWAP( $\mathcal{G}_a, \mathcal{G}_b$ );
12: end for
```

3.4 DRM-connect

We now propose DRM-connect (Algorithm 9), an online graph-based planning and replanning algorithm, designed for replanning in dynamic environments.

Many of the replanning algorithms discussed in Chapter 2 retain and leverage some prior experience of similar tasks in the same domain, however, all of these discard at least some information which may be valuable in resolving the task in a replanning setting. DRM-connect attempts to address this by providing a framework to encode all valuable prior experience in a graph used for replanning. We explore Reeb graphs as a possible method of encoding this information (see Chapter 4).

The task is to find a path between a start and end point (q_s and q_g respectively) using a graph $\mathcal{G}$ (a possible choice of which is an approximation of a Reeb graph - see Chapter 4). If either the start or end point do not exist in $\mathcal{G}$ (e.g. if the start or end point has moved due to sensor errors, replanning, or if this is a new task), then these are added as unconnected nodes to $\mathcal{G}$ (lines 1 to 6). Then, a search is conducted on the graph for the shortest path (line 7), checking collisions lazily, using the DRM-lazy algorithm (see Section 3.2). If this fails to return a valid path, the graph is repaired (line 9) using PRM-connect (see

Section 3.3), after which DRM-lazy is called again (line 10). This is repeated until a valid path is found.

DRM-connect generalises both RRT-connect and DRM. If no prior knowledge is used ($\mathcal{G}$ is empty), then DRM-connect simplifies to RRT-connect, where the start and end points are used as the roots for the trees which grow to connect to one another. Assuming then that the knowledge provided is useful in solving the task, the performance of DRM-connect should be at worst the same as RRT-connect. If the path repair algorithm (DRM-lazy) is removed or there exists a collision-free path, then DRM-connect is equivalent to DRM. Thus (ignoring the time for collision checking), DRM-connect will match the performance of DRM in cases where DRM would succeed - namely when the start and end points lie on $\mathcal{G}$ and there are no collisions on the shortest path.

DRM-connect differs from DRM in that if a path is not found through the graph, a repair process is carried out to join the subgraphs, rather than using RRT-connect from scratch. It is expected that this (rather than an RRT-connect fallback) should reduce the replanning time if obstacles appear which split the graph into disjoint subgraphs, since the motions encoded by these subgraphs can be reused.

Currently, no workspace-configuration space mapping is used to invalidate edges, but rather lazy collision checking (this will be left as an enhancement). Since collision checking is typically the largest contributor to query time, the mapping (which allows collisions to be checked using a lookup) can significantly improve replanning times, especially when the number of edges to check is large. However, the graphs tested here are fairly sparse. For larger graphs, the mapping may prove very valuable.

Algorithm 9 DRM-CONNECT ($\mathcal{G}, q_s, q_g$)

```

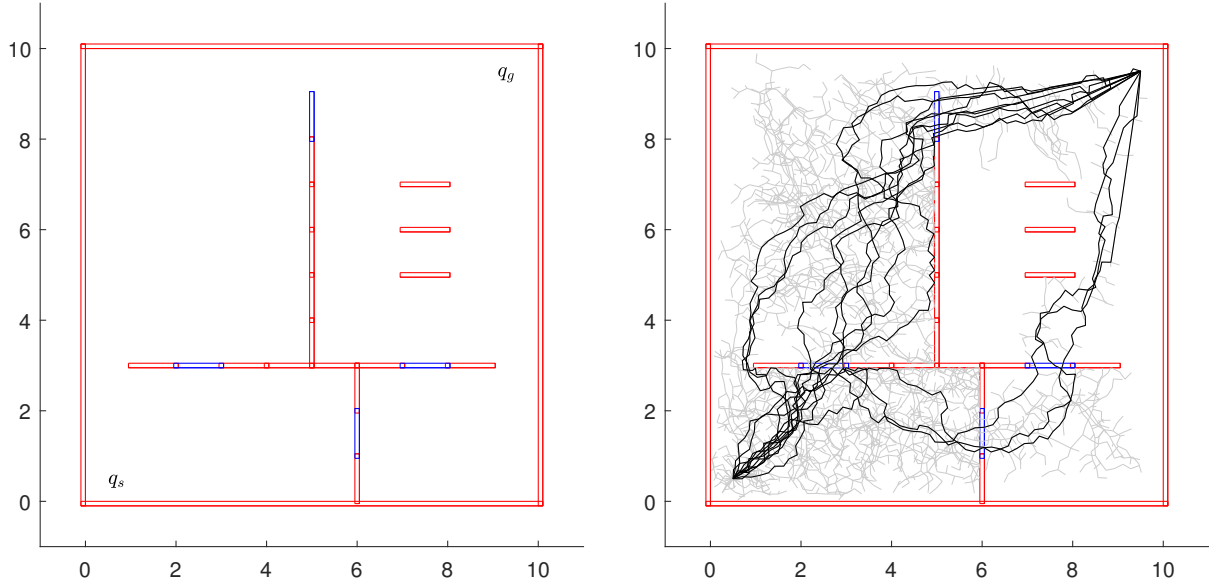
1: if  $q_s \notin V(\mathcal{G})$  then
2:    $\mathcal{G}.\text{add\_node}(q_s)$ 
3: end if
4: if  $q_g \notin V(\mathcal{G})$  then
5:    $\mathcal{G}.\text{add\_node}(q_g)$ 
6: end if
7:  $\text{returnFlag}, p \leftarrow \text{DRM-LAZY}();$ 
8: while not returnFlag do
9:   PRM-CONNECT();
10:   $p \leftarrow \text{DRM-LAZY}();$ 
11: end while
12: return  $p$ 
```

To infer the completeness of DRM-connect, we need to consider all possible outcomes. If a path exists through the graph $\mathcal{G}$, DRM-connect finds a solution and returns it in finite time. If a path doesn't exist through the graph, PRM-connect is called, which behaves materially like RRT-connect (except that graphs are constructed, rather than trees). This has the same completeness guarantee of RRT-connect, which is probabilistically complete. As such, DRM-connect is expected to be probabilistically complete.

3.5 Experiments

To test the DRM-connect algorithm, we provide it with a naïve graph and test how effectively it replans in a 2-D navigation domain against RRT (a planner concerned with generating a feasible path). We test both replanning time and the length of the solution path. We hypothesise that DRM-connect will replan significantly faster than RRT.

This naïve graph is just the union of a set of paths from previous plans of the same task. In Chapter 4 we discuss more sophisticated methods of creating this graph.



(a) 2-D DOORS domain. The red lines indicate fixed obstacles and the blue lines indicate obstacles which may appear and disappear (although not during a task). The task is to navigate from q_s to q_g .

(b) Example of paths generated in the 2-D DOORS domain. The gray lines are the full RRT trees generated, while the black lines are the resulting solution paths.

Figure 3.1: DOORS domain and example paths.

3.5.1 The DOORS domain

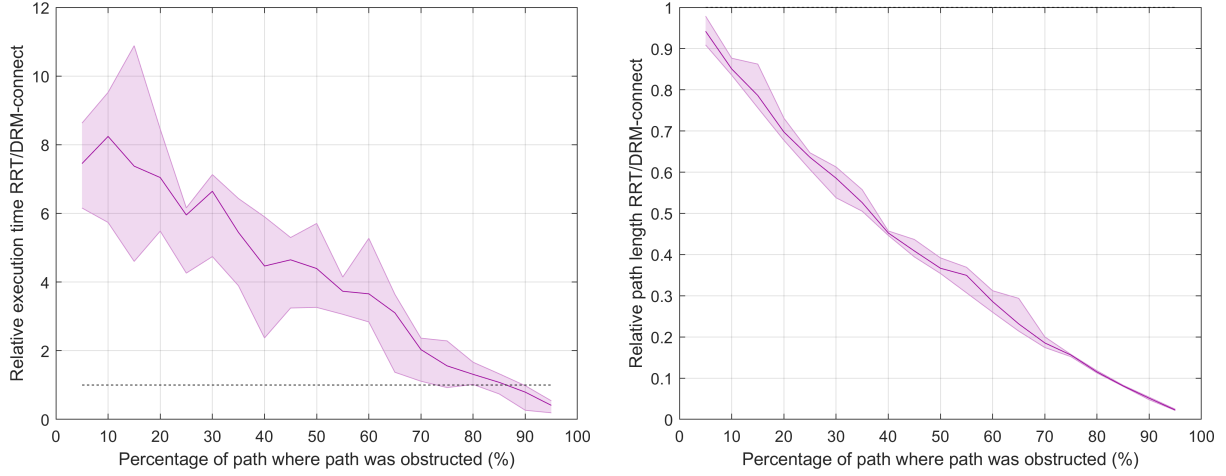
The first domain we test the DRM-connect algorithm with a naïve graph in is the 2-D DOORS domain (shown in Figure 3.1a). The agent in this domain is a holonomic point robot with 2 degrees of freedom. The red lines indicate fixed obstacles, while the blue lines indicate transient obstacles. These transient obstacles represent doors, which may be present (closed) or absent (open) during a planning query. The planning task is to find a path from q_s at (0.5, 0.5) to q_g at (9.5, 9.5).

During the offline learning phase, twelve paths are created for the same task using standard RRT with $r = 0.1$, $maxIter = 10000$ and $p_g = 0.05$. An example of this set of paths is shown in Figure 4.2a. We convert these paths into a graph by connecting all common nodes together. In a continuous space, as all the domains considered here are, the likelihood of this occurring at nodes other than the start and end points is vanishingly small.

All parameters are chosen experimentally. An r value of 0.1 allows the planner to take steps of about 1% of the dimension of the domain, which allows for reasonable solve times. The p_g value chosen increases the speed of RRT without significantly affecting the failure rate. The choice of $maxIter$ is fairly arbitrary in this case, since RRT seldom requires that many nodes to solve tasks in this domain.

Results and discussion

To test the performance of DRM-connect in this domain, we generate a naïve graph as described above, then use DRM-connect to find a valid trajectory from q_s to q_g . A small obstacle is then inserted $n\%$ of the way through the trajectory, and both DRM-connect and RRT are used to replan the trajectory from the point on the path immediately before the obstacle to q_g . Values for n range from 5 to 95 in steps of 5. This process (generating training paths, obstructing the path through the task and testing replanning for all percentages) is repeated 10 times, from which the results in Figures 3.2 and 3.3 are generated.



(a) Relative execution time between RRT and DRM-connect. A larger value indicates better performance of DRM-connect.

(b) Relative path length between RRT and DRM-connect. A larger value indicates a shorter path found by DRM-connect.

Figure 3.2: Relative execution time and path length of RRT/DRM-connect for the 2-D DOORS domain using a naïve graph. The x -axes show how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are the median values over 10 runs, with the shaded region representing the interquartile range (10 sets of example paths, naïve graphs and obstructions). The dotted lines indicate a relative value of 1 (parity of the two algorithms' performance).

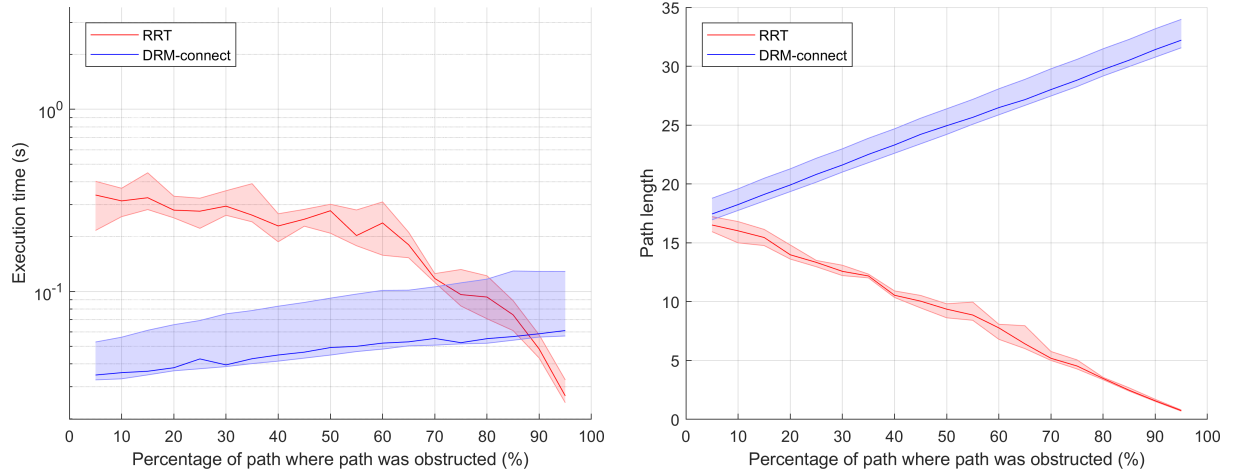
We compare the replanning performance (both execution time and path length) of DRM-connect to RRT. Since both solve times and path length are lower-bounded and not upper-bounded, we don't expect the distribution of these values to be normal. As such, interquartile range (IQR) is used as a measure of the variance. The IQR is given by $Q_3 - Q_1$, the first quartile value (Q_1) subtracted from the third quartile value (Q_3).

Figure 3.3a shows that when the path is interrupted early (close to the start point), DRM-connect replans much faster than RRT (6 - 10 times faster, according to Figure 3.2a). As the path is interrupted later, the performance benefit of using DRM-connect reduces. For interruptions close to the goal (from around 90%) RRT is capable of replanning faster than DRM-connect. To understand why this happens, we need to understand the length of the paths generated by the two algorithms.

Figure 3.3b shows the path lengths returned by the two algorithms. As the distance to replan gets shorter (when the task execution is interrupted later), the path length returned by RRT reduces linearly. The path length returned by DRM-connect, however, increases linearly. This is because DRM-connect will attempt to backtrack along its current trajectory until it finds another arc along which the goal can be reached. Because the naïve graph will usually only have junctions at the start and end points (since the space is continuous), this backtracking will usually require it to return to the start point, and then follow a different path. When the path is interrupted very close to the goal (95%), the path returned by DRM-connect is about twice the length of the paths from near the start (5%). This confirms that DRM-connect is backtracking all the way to the start.

A graph built with better connections between the training paths is expected to fix this large amount of backtracking (see Chapter 4 for an exploration of this).

DRM-connect does lazy collision checking, which means the number of collisions checked will be approximately proportional to the path length it returns (assuming there are no collisions - if there are collisions this will increase). The later we interrupt the path, the more backtracking is required, which lengthens the path along which to check for collisions. This is the reason that the replanning time of DRM-connect increases gradually the later the path execution is interrupted.



(a) Comparison of execution time between RRT and DRM-connect.

(b) Comparison of path length between RRT and DRM-connect.

Figure 3.3: Execution time and path length of RRT and DRM-connect for the 2-D DOORS domain using a naïve graph. The x -axes show how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are the median values over 10 runs, with the shaded region representing the interquartile range (IQR).

3.6 Conclusion

In this chapter we presented a novel graph-based replanning algorithm termed DRM-connect, as well as several additional supporting algorithms. We tested DRM-connect's replanning performance on a navigation task in a 2-D domain with a naïve graph and compared it to the performance of RRT.

It was shown to perform well in certain circumstances (when the path was interrupted far from the goal) and poorly in others (when the path is interrupted close to the goal). The poor performance (both replanning time and path length) is due to the underlying graph (likely) only being connected at the start and end point, which requires DRM-connect to do a significant amount of backtracking. A potential improvement would be to generate a graph which is able to provide shorter local loops, which would reduce the amount of backtracking. This is explored in Chapter 4.

Chapter 4

Planning with Reeb Graphs

4.1 Introduction

In Chapter 3 we defined a novel approach to leverage a graph in a replanning framework. Up to this point we’ve been agnostic to how this graph is generated. We now explore one possible method of creating this graph, namely by approximating a Reeb graph. This approach attempts to leverage topological knowledge of the free configuration space for later replanning.

The naïve graphs of Chapter 3 forced DRM-connect to backtrack to the start point before finding a new path. Each training path was kept separate, with no ability to switch between training paths partway through.

A more sensible approach would be to capture some notion of where it should be possible to switch paths. These training trajectories contain two types of information. Firstly, we have a set of points which are collision-free. Secondly, we have line segments between them, which encode how to move between these valid states to solve the task. We propose a topological approach to connecting and contracting the training paths into a more sensible graph for DRM-connect.

4.2 Background

4.2.1 Computational Topology

Applying the study of topology to path planning provides a formalism of $\mathcal{C}_{free}$ which allows the agent to reason about the path-connectedness of the space. For replanning, this extends to allowing the agent to evaluate whether the changed environment is sufficiently different to warrant searching for an entirely new path, or whether to search for a similar path to the current trajectory. In this section we describe some of the background of topology which we use later.

In topological path planning, the free configuration space is often considered as a manifold. For a topological space to be a manifold M , it must have, for every $x \in M$, an open set $O \subset M$ such that [LaValle, 2006]:

1. $x \in O$
2. O is homeomorphic¹ to $\mathbb{R}^n$ and
3. n is fixed for all $x \in M$

¹Spaces X and Y are *homeomorphic* ($X \cong Y$) if a bijective function $f : X \rightarrow Y$ exists, such that both f and f^{-1} are smooth. Intuitively, this requires that X can be continuously deformed into Y , without creating or destroying any holes.

Intuitively, this requires that the space near any point x must behave like a surface (as if it exists on $\mathbb{R}^n$).

An important notion for topological path planning is topological equivalence. The strong form of equivalence is homotopy, which can be applied to both spaces and paths. Maps $f, g : X \rightarrow Y$ are homotopic if there is a continuous map H which deforms f onto g . Formally, $H : X \times [0, 1] \rightarrow Y$ such that $H(x, 0) = f(x)$ and $H(x, 1) = g(x)$. We can then write $f \simeq g$ [Carlsson, 2009].

Spaces are homotopy equivalent if they have the same “shape”. Intuitively, two spaces are homotopy equivalent if one can be continuously deformed into the other, without creating or merging holes. Additionally, contractions are allowed, so a ball is homotopy equivalent to a point (whereas they are not homeomorphic). Formally, two spaces X and Y are homotopy equivalent if there exists continuous maps $f : X \rightarrow Y$ and $g : Y \rightarrow X$ such that $g \circ f \simeq \text{id}_X$ and $f \circ g \simeq \text{id}_Y$ (the composition of these two maps (f and g) must yield identity maps (id_X and id_Y) [Carlsson, 2009].

A similar notion of homotopy can be applied to paths. Paths τ_1 and τ_2 are homotopic if one can be continuously deformed onto the other. There must exist a continuous function $h : [0, 1] \times [0, 1] \rightarrow X$ which meets the four following criteria [LaValle, 2006]:

1. $h(s, 0) = \tau_1(s) \forall s \in [0, 1]$. When $t = 0$, h must be τ_1 .
2. $h(s, 1) = \tau_2(s) \forall s \in [0, 1]$. When $t = 1$, h must be τ_2 .
3. $h(0, t) = h(1, t) \forall t \in [0, 1]$. The start points should match.
4. $h(1, t) = h(0, t) \forall t \in [0, 1]$. The end points should match.

A weaker form of equivalence is homology, which is based on cell-complexes (see below), rather than mappings. This yields this equivalence relationship generally more useful for computational topology. See Eilenberg [1944] or Hatcher [2002] for a formal definition of homology.

When path planning in any suitable complex or high dimensional space, we seldom have access to the true (continuous) topological manifold of $\mathcal{C}_{free}$. The standard approach to deal with this is to approximate the space by a set of samples. This requires recasting some of the topological concepts presented above in terms of cell complexes, rather than the true smooth spaces. An n -cell is a topological primitive, corresponding to subsets $\mathbb{R}^n$. i.e. a 0-cell is a point, a 1-cell is a line segment, a 2-cell is an open disk etc. We can construct surfaces from sets of n -cells.

When n -cells are connected, they form an n -skeleton. This n -skeleton represents the space X , and is called a cell complex. A 1-dimensional cell complex X^1 is a topological graph. It comprises 0-cells (nodes) connected by 1-cells (edges).

A popular method of representing a topological space is the simplicial complex [Edelsbrunner and Harer, 2010]. A simplicial complex is the representation of a space as the union of points (0-simplices), line segments (1-simplices), triangles (2-simplices) and their higher dimensional equivalents. In the same way as cell complexes comprise n -cells, simplicial complexes comprise simplices [Carlsson, 2009]. A k -simplex is a primitive of dimension k . A simplicial complex can be formed by triangulating data points.

Various types of complexes exist, with different theoretical guarantees and computational trade-offs, which include Čech complexes, Vietoris-Rips complexes, Delaunay complexes and others. For a full treatment, see Edelsbrunner and Harer [2010].

A common feature of many complex types is a filtration or scale parameter ϵ . The filtration parameter controls the maximum distance at which 0-simplices will be connected by a 1-simplex [Carlsson, 2009]. For cases where these samples are drawn from the space X , we have to make some assumption about the sampling density of these points, to tell whether holes in the resulting complex are due to holes in X , or just that we happen not to have sampled there. The resulting homology of the complex can be

evaluated at all values of ϵ simultaneously. This allows us to measure the scale of the topological features. Investigating the persistence of these features as we change ϵ helps to decide whether the feature is present in X , or is just an artifact of noise [Carlsson, 2009].

4.2.2 Trajectory classification

In cases where replanning due to dynamic obstacles is required, the appearance of an obstacle could have one of two effects. We would either require minor changes to the planned path (such as a person stepping around an unexpected chair) or major changes to the path (such as a door unexpectedly being closed, and the person needing to find a path through another room to the goal).

From a topological perspective, this can be treated by reasoning about trajectory equivalence classes. Equivalence classes (or homotopy equivalence classes) are collections of all trajectories through some space, such that no trajectory from one class can be continuously deformed to any trajectory in any other class [Pokorny et al., 2016]. The difference, then, between these two scenarios mentioned above is that the closed door makes an entire homotopy equivalence class of trajectories invalid, while the chair may split an equivalence class, but (probably) leaves many paths through the current trajectory class valid.

For these reasons, understanding the topology of the space in which the agent is operating is valuable. In the examples above, we may be able to update the trajectory around the chair by searching for a new trajectory in the same homotopy class (which may simply involve deforming the current trajectory). In the case of the closed door, however, continuously deforming the trajectory would not work, and planning through a new trajectory class would be required.

Pokorny et al. [2016] propose an approach to reasoning about these trajectory classes for path replanning and trajectory optimisation. Samples from $\mathcal{C}_{free}$ are used to estimate the path-connectedness of the configuration space. This estimation uses filtrations of simplicial complexes. Persistent homology is used to find an optimal scale factor for the filtrations. This approach works as follows:

1. Samples are drawn from $\mathcal{C}_{free}$.
2. From these, multi-scale simplicial complexes are built (specifically Delaunay-Čech complexes), as a description of the topology of the space.
3. The persistence graph of these simplicial complexes is generated to evaluate the optimal or correct filtration value.

In this replanning situation, knowledge of the trajectory classes of the environment is useful as long as the agent knows whether to replan in the same class of trajectories, or switch to a different class. This motivates for a compact representation of both the equivalence classes and the paths through them. This leads into Reeb graphs, and how they can be created over simplicial complexes.

4.2.3 Reeb graphs

A Reeb graph is an abstract graph which describes the evolution of level sets of a continuous function f defined on some volume [Edelsbrunner and Harer, 2010]. This is the 1-skeleton of the volume (and can be thought of as the contraction of the volume onto a set of arcs through the “centre” of it). An example of a volume and its corresponding Reeb graph is shown in Figure 4.1. The function Φ maps the volume X to the Reeb graph $R_f(X)$. This abstract graph can then be embedded back into the volume to create a compact description of it.

The information captured by a Reeb graph is:

1. The critical points of the level sets (points where the volume appears, disappears, splits or merges, as we sweep the height function f). These become nodes of the Reeb graph, and are depicted in Figure 4.1 as clear circles on the volume, and their corresponding locations on the Reeb graph.

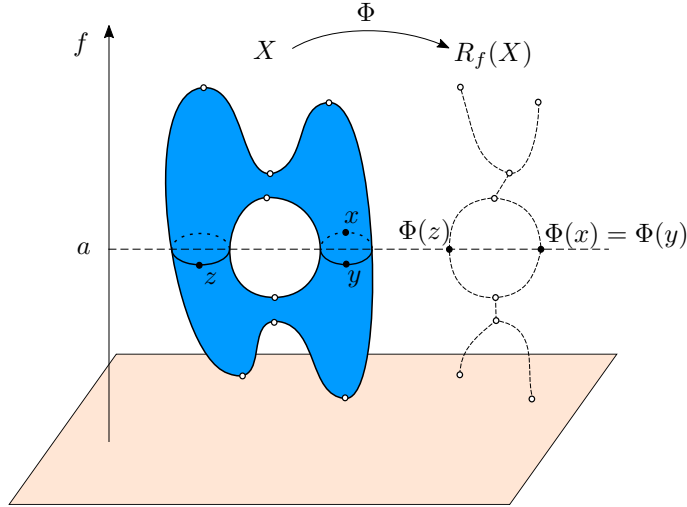


Figure 4.1: An example of a Reeb graph. The blue volume has been contracted to form the 1-skeleton shown on the right. The continuous function f is the height function from which the Reeb graph is generated. This need not map to a physical dimension of the volume. The clear circles on the volume and Reeb graph correspond, and represent critical points. The solid circles also correspond, and are an example of a level set. The Reeb graph is a compact representation of the homology of the volume.

2. The volume connecting these critical points, which become the edges of the Reeb graph.

Given the height function f , the Reeb graph describes the 1-connectivity of the space. Ignoring spurious loops, this is the graph on which the contraction of all trajectory equivalence classes must lie. Thus the Reeb graph could be a compact representation of all trajectory classes.

Edelsbrunner and Harer [2010] provide an approach to generating a Reeb graph for a given manifold and function, assuming the manifold is triangulated and the function is a Morse function (a smooth function on a manifold, $f : \mathbb{M} \rightarrow \mathbb{R}$, which has unique critical points, none of which are degenerate [Edelsbrunner and Harer, 2010]). This has the following high-level characteristics:

1. The manifold is swept in order of increasing function value $f(u_i)$, where u_i is the i th vertex of the triangulation of the manifold.
2. Whenever an appearing, disappearing, bifurcating or merging volume is found, the Reeb graph being constructed is updated accordingly, by creating a node, and joining it to other present nodes, according to the type of critical point encountered.

The full description can be found on page 173 of Edelsbrunner and Harer [2010]. We follow this approach to create our approximate Reeb graphs (see Section 4.3.1). The Reeb graphs we generate are approximate in the sense that we don't enforce that they are a faithful reduction of the complexes, but rather that they contain the useful path connectivity of the space. This is discussed in more detail in the following section, along with how a Reeb graph could be used in conjunction with DRM-connect.

4.3 DRM-connect with Reeb graphs

When introducing DRM-connect in Section 3.4, we didn't prescribe any approach to generating the graphs which underly DRM-connect. Here we present one option of a graph to use with DRM-connect, generated from prior experience of the same task. Our choice of graph should simply be one that encodes information about alternative ways (redundancies) of solving the task when an obstacle blocks the current path. Our requirements for this graph are as follows:

1. The graph should represent the trajectory classes available from the prior experience.
2. It should provide a mechanism for switching between the trajectory classes without significantly affecting the path length.
3. It should be compact, such that replanning times are fast.

Depending on how many training paths are used (and the planning algorithm used to generate them) the collection of sampled points will begin to approximate the free configuration space. This allows us to generate a Reeb graph from our sampled points. It's important to note that we don't intend this Reeb graph necessarily to encode the *true* topology of the space, but rather the *useful* topology of the task manifold.

Since Reeb graphs require a height function f (essentially a height for each sampled node), we need to assign heights to these nodes in such a way that there is a flow from the start point to the end point. Intuitively, the height of a specific node is representative of how far through the task this node is. This height function should be monotonically increasing as we move from the start to the end.

These nodes are joined to all other nodes in a radius ϵ (the filtration or scale parameter) in a triangulation step (as discussed in Section 4.3.1). The topological structure of the space can then be estimated by contracting these simplicial complexes.

Since we're not attempting to recover the true topology of the space, but rather a representation which satisfies our needs for replanning, we can use the filtration parameter ϵ to control the density of our embedded Reeb graphs. This in turn should affect the redundancy and replanning time (since a larger graph could provide replanning solutions to more environment changes, yet a graph search on it will take longer). See Section 4.4.1 for a discussion of this.

4.3.1 Embedding Reeb graphs

A Reeb graph is an abstract graph; for it to be useful with DRM-connect, we need to embed it back into the configuration space. Ge et al. [2011] provide the following steps to generate an embedded Reeb graph:

1. Convert training paths into an abstract Reeb graph.
2. Embed this Reeb graph into the configuration space.

Since we're only interested in the final embedded Reeb graph, it's not necessary to carry these out as two distinct steps. The critical points in our embedded Reeb graph will correspond to nodes in the simplicial complex, so retaining this information between steps 1 and 2 is useful. As such, we can apply a single procedure to convert the nodes from the training paths into the embedded Reeb graph.

The nodes of each of the set of training paths p are extracted and the pairwise distance between these points is calculated. A graph ($\mathcal{G}_{adjacency}$) is constructed by connecting all points with a pairwise distance less than or equal to ϵ . Height values are then assigned to each node in the graph, using GET-HEIGHT-MAP (Algorithm 10). The graph is then contracted into an embedded Reeb graph by BUILD-REEB-GRAPH-FROM-HEIGHT-MAP (Algorithm 11).

The function GET-HEIGHT-MAP is responsible for assigning height values to each node. The requirements we impose for this are simply that the heights along the shortest path from any point to q_g are increasing. Additionally, an advantage would be to have these node heights fairly uniformly spread, so that the sweep of heights during BUILD-REEB-GRAPH-FROM-HEIGHT-MAP (see below) encounters the nodes spaced out evenly across the range of height values.

GET-HEIGHT-MAP takes as input the adjacency graph $\mathcal{G}$, q_s and q_g . It first assigns heights of 0 and 1 to q_s and q_g respectively (lines 1 to 3). Then heights are assigned to all nodes on the shortest path between these two points by ASSIGN-HEIGHTS (lines 8 and 9), as a percentage of the distance between them (this spreads the height values along the path). The graph is then split into all the separate disjoint

Algorithm 10 GET-HEIGHT-MAP($\mathcal{G}, q_s, q_g$)

```
1:  $h(q_s) \leftarrow 0$ 
2:  $h(q_g) \leftarrow 1$ 
3:  $\mathcal{G}_{conn} \leftarrow \mathcal{G}$ 
4: while (any  $h$  unassigned) do
5:   for all  $\mathcal{G}_{sub}$  in SUBGRAPH( $\mathcal{G}_{conn}$ ) do
6:      $q_0, h_0 \leftarrow \text{MIN}(h_{sub})$ 
7:      $q_1, h_1 \leftarrow \text{MAX}(h_{sub})$ 
8:      $p \leftarrow \text{SHORTEST-PATH}(\mathcal{G}_{sub}, q_0, q_1)$ 
9:      $h(q_0 : q_1) \leftarrow \text{ASSIGN-HEIGHTS}(p, h_0, h_1)$ 
10:    REMOVE-EDGES( $\mathcal{G}_{conn}, p$ )
11:   end for
12: end while
13: return  $h$ 
```

Algorithm 11 BUILD-REEB-GRAPH-FROM-HEIGHT-MAP($\mathcal{G}, h$)

```
1:  $b \leftarrow \text{CREATE-BANDS}(h, n)$ 
2: for all  $b_i$  in  $b$  do
3:    $seed\_nodes \leftarrow \text{ASSIGN-SEED-NODES}(\mathcal{G}, h, b_i)$ 
4: end for
5:  $\mathcal{G}_{embedded\_reeb} \leftarrow \text{JOIN-SEEDS}(seed\_nodes, b)$ 
6: return  $\mathcal{G}_{embedded\_reeb}$ 
```

subgraphs (SUBGRAPH - line 5). ASSIGN-HEIGHTS takes the shortest path p and the heights of the start and end nodes (h_0 and h_1 respectively) and assigns heights to the nodes along the path, increasing uniformly with the distance along the path. The edges along this path are then removed from $\mathcal{G}_{conn}$ (line 10). This ensures that all edges remaining should in future form part of a shortest path elsewhere, while retaining the nodes, which are required for future iterations.

Then, for all shortest paths between points with known heights (using the heights h_{sub} of the subgraph $\mathcal{G}_{sub}$ - lines 6 and 7), this process is repeated, spreading the height values from the start and end of the arc to all points along the arc. This mapping between the nodes and heights is returned as h .

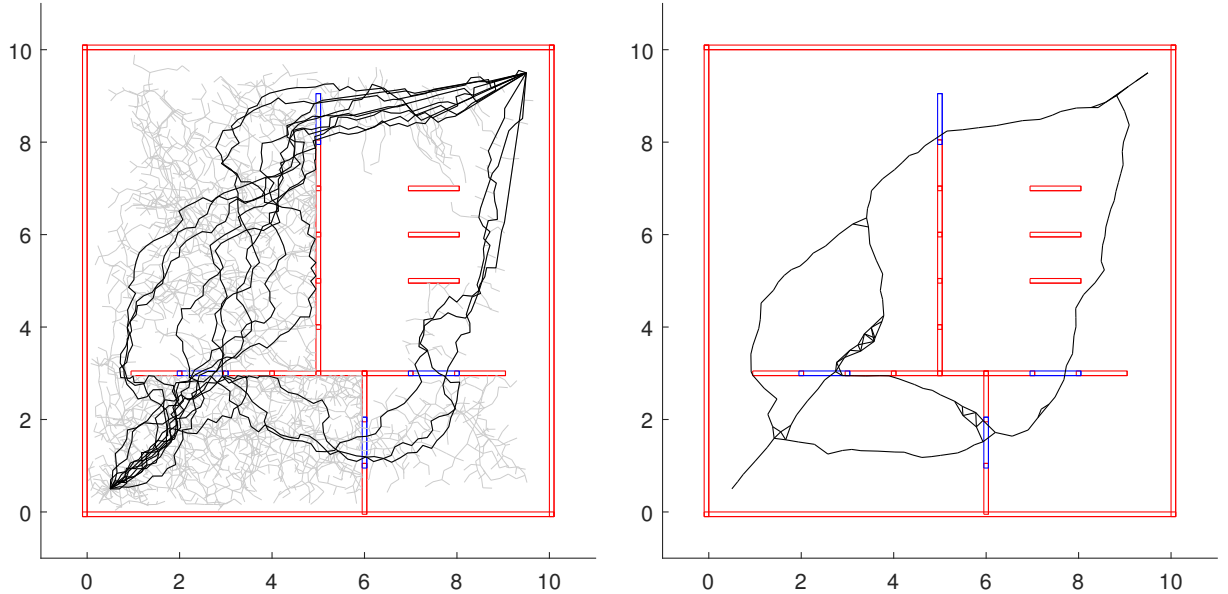
BUILD-REEB-GRAPH-FROM-HEIGHT-MAP takes the same adjacency graph $\mathcal{G}$, and the heights h from GET-HEIGHT-MAP and segments the nodes by height into a set of bands of uniform width (line 1). 7 bands are used here. For each band, ASSIGN-SEED-NODES (line 3) finds the geometric midpoint of each connected subgraph in that band. For each midpoint, the closest node to this is found. These are inserted into the Reeb graph as seed-nodes (line 5).

If possible, JOIN-SEEDS connects the seed-nodes in each adjacent band to one another via the shortest path (adding any nodes along this path), which form the edges of the Reeb graph. This yields an approximate embedded Reeb graph, which is returned as $\mathcal{G}_{embedded_reeb}$.

The number of bands can be varied depending on the volume of data points in the adjacency graph and the computational requirements. The approach used here guarantees that the embedded Reeb graph is a subgraph of $\mathcal{G}_{adjacency}$ above.

4.4 Experiments

The effectiveness of using Reeb graphs as the graphs underpinning DRM-connect is tested in several domains, for various task complexities. One navigation task is tested in a 2-D domain (Section 4.4.1) and three manipulator reaching tasks are tested in 7-D domains (Sections 4.4.3 - 4.4.5).



(a) Example of paths generated in the 2-D DOORS domain. The gray lines are the full RRT trees generated, while the black lines are the resulting solution paths. (b) Example of a Reeb graph generated in the 2-D DOORS domain.

Figure 4.2: Example paths and Reeb graph in the 2-D DOORS domain.

4.4.1 The DOORS domain

We first test the DRM-connect algorithm with a Reeb graph in the 2-D DOORS domain (described in Section 3.5.1 and shown in Figure 3.1a). Through random combinations of these transient obstacle states (the doors), we can directly affect the homology of the space, creating or destroying specific classes of trajectories.

Again, during the offline learning phase, twelve paths are created using standard RRT, using the same parameters as Section 3.5.1. An example of this set of paths is shown in Figure 4.2a.

These paths are then contracted into an approximate Reeb graph, using the approach discussed in Section 4.3.1 with $\epsilon = 2r - 0.01$. Since the paths are generated using a step size of r (the largest extension allowed in a single step), this value is close to the largest we can use without often connecting nodes on a path which have another node between them two edges away, while (hopefully) still connecting to other paths. The Reeb graph created from the paths in Figure 4.2a is shown in Figure 4.2b.

A Reeb graph contains the skeleton of the homology of the space. For our application, using an approximate Reeb graph, we're interested only in capturing an approximation of the true homology of the space (as represented by the samples provided by the training paths) to the extent that it gives us useful information about the space and task to leverage.

A simple requirement might be that we wish to have a graph which encodes several of the major trajectory classes. For the Reeb graph shown, we have a broad range of trajectory classes represented, ones which should give DRM-connect the ability to backtrack and recover, should an obstacle obstruct the trajectory class the robot is currently following. For example, if the robot is near $(7, 2)$ and the door near $(8, 3)$ were to close, DRM-connect would provide a route which passed through the left side of the environment. Note that we would ideally want to recover from the door closing by moving to the right of the obstacle at $(9, 3)$. However, this trajectory class was not present in this particular training set, so this was not encoded in the Reeb graph.

Another similar loss of information is the door near (5, 8). In the training set we can see that trajectories which go both through and around the door are present. Due to the ϵ value used, all of these arcs are collapsed onto a single path which travels through the door. In this case, choosing a larger ϵ value (which allows for faster replanning) may make the replanning less robust in cases where this door closes.

Obviously, providing more paths/samples in the training phase would provide a more accurate representation of all available trajectory classes. The number used here (12) is chosen arbitrarily, and an investigation into the tradeoffs of varying the number of training samples is left for future work.

We evaluate whether the Reeb graphs generated from the training paths satisfy the requirements suggested above (encoding of the major trajectory classes and reducing the amount of backtracking required) by simulation, comparing the time and path length of DRM-connect with a Reeb graph against RRT and RRT-connect. The time taken from replanning is our major criterion, since the aim of this work is to increase the speed of replanning. We evaluate path length as well, to ensure that DRM-connect does not return significantly sub-optimal paths.

Results and discussion

To test the performance of DRM-connect in this domain, we generate a Reeb graph as described above, then use DRM-connect to find a valid trajectory from q_s to q_g . A small obstacle is then inserted $n\%$ of the way through the trajectory, and both DRM-connect and RRT are used to replan the trajectory. Values for n range from 5 to 95 in steps of 5. This process (generating training paths, obstructing the path through the task and testing replanning for all percentages) is repeated 10 times, from which the results in Figures 4.3 and 4.4 are generated. Generating the Reeb graph from the training paths takes approximately 2 seconds.

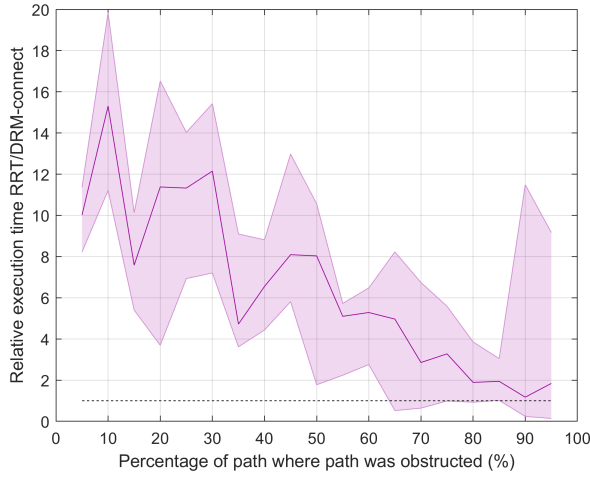
We compare the replanning performance (both execution time and path length) of DRM-connect to RRT. The results from DRM-connect with naïve graphs in Section 3.5.1 are also included. All references here to DRM-connect without specifying the graph type indicate DRM-connect with a Reeb graph.

The main observation from these results is that DRM-connect typically replans significantly faster than RRT, with minimal trade-offs for path length.

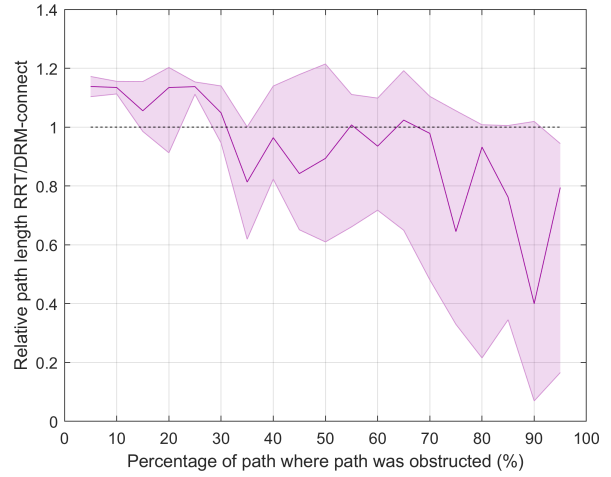
From Figure 4.3a we can see that if the obstruction is early in the task execution (far from the goal), then DRM-connect is able to replan around 10 times faster than RRT. If the obstruction occurs later in the task, then DRM-connect is typically only a few times faster than RRT. In terms of path length, we can see from Figure 4.4 that the path lengths for DRM-connect and RRT are very similar. However, the interquartile range (IQR) of path length for DRM-connect is higher than RRT for obstructions later in the path. A rationalisation for this is an obstruction late in the task execution may obstruct the Reeb graph such that replanning yields a path which is an arc through the Reeb graph, but which requires significant backtracking. In this case replanning will be very rapid, but would also yield a longer solution path. Section 4.6 explores in more detail how well the Reeb graphs capture the important aspects of the topology of the space.

We can see that the path lengths of DRM-connect with a Reeb graph are significantly shorter than DRM-connect with a naïve graph, while the replanning times are similar. This gives an early indication that the information encoded by the Reeb graph is useful (in that it helps mitigate backtracking).

If the obstruction occurs close to the goal, we see two things. Firstly, the IQR of DRM-connect increases. This is the case where the overhead of DRM-connect has the biggest effect. If an alternate route exists through the graph, the replanning will be very quick. Secondly, the median solve time for RRT seems to be lower than that of DRM-connect. This is a case where the slight overhead of searching for the shortest valid path combined perhaps with randomness yields worse performance for DRM-connect than RRT.

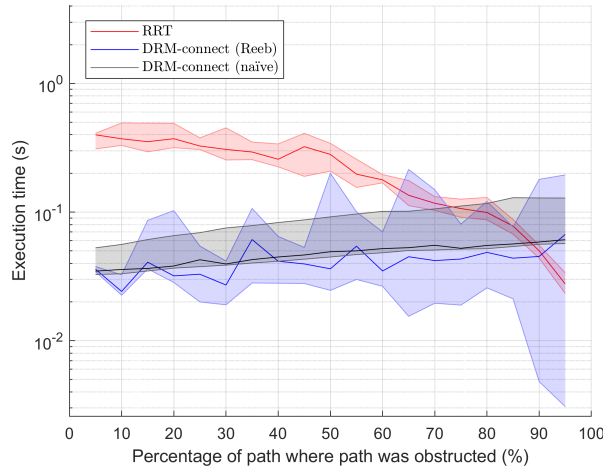


(a) Relative execution time between RRT and DRM-connect. A larger value indicates better performance of DRM-connect.

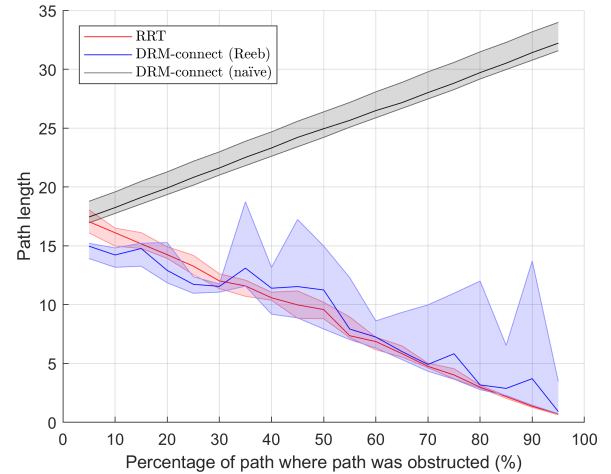


(b) Relative path length between RRT and DRM-connect. A larger value indicates a shorter path found by DRM-connect.

Figure 4.3: Relative execution time and path length of RRT/DRM-connect for the 2-D DOORS domain. The x -axes show how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are the median values over 10 runs, with the shaded region representing the interquartile range (10 sets of example paths, Reeb graphs and obstructions). The dotted lines indicate a relative value of 1 (parity of the two algorithms' performance).



(a) Comparison of execution time between RRT and DRM-connect.



(b) Comparison of path length between RRT and DRM-connect.

Figure 4.4: Execution time and path length of RRT and DRM-connect for the 2-D DOORS domain. The x -axes show how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are the median values over 10 runs, with the shaded region representing the interquartile range (IQR). For both time and path length, DRM-connect is tested with both a Reeb and naïve graph.

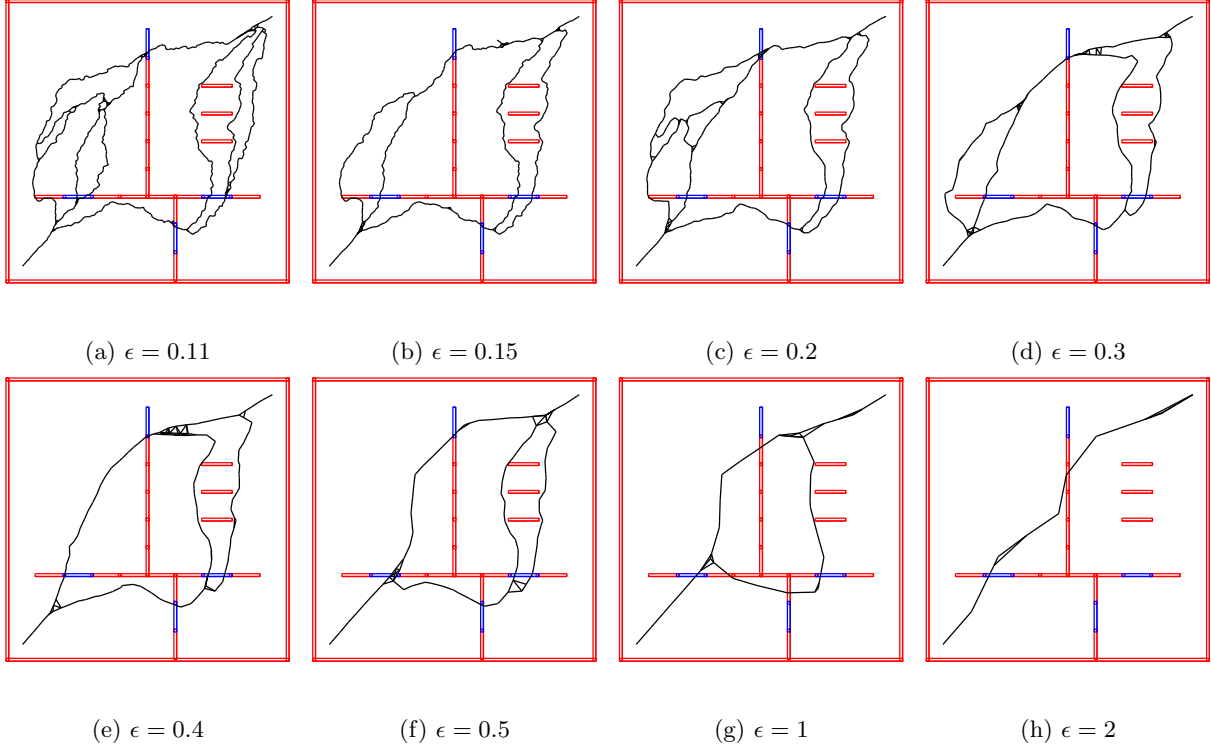


Figure 4.5: Reeb graphs in the DOORS environment for various ϵ values.

The overhead cost of generating the training paths ($\approx 4.8s$) and building the Reeb graph from these ($\approx 2s$), suggests that using DRM-connect in this 2-D environment would be useful if very frequent and fast replanning were necessary, or multiple iterations of the same or similar tasks were to be run. In other cases, the overhead cost is large enough that simply using RRT would be more sensible.

A comparison of the performance of naïve graphs and Reeb graphs for this task is presented in more detail in Section 4.6.

Effect of varying the ϵ value

Figure 4.5 shows the effect of varying ϵ for the Reeb graph constructed from the example paths shown in Figure 4.2a. The smallest ϵ value tested is $\epsilon = 0.11$, which is slightly larger than the minimum size since $r = 0.1$. The discussion below applies to the example paths used in the 2-D DOORS domain. However, the trends are expected to extend to other 2-D domains, and to a lesser extent higher dimensional tasks.

As ϵ increases, the Reeb graph takes longer to construct, since the number of connections in the simplicial complex increases.

The general trend is that as the ϵ value increases, the graph is contracted more and more, and has fewer arcs. From a replanning perspective, this would reduce the robustness of using this Reeb graph with DRM-connect, since there are now fewer local loops to use to recover from an obstacle appearing along the trajectory. The speed that the graph search can be carried out also increases, since the graph has fewer edges. This gives us a useful lever: we can perhaps tune the robustness/efficiency ratio by changing the ϵ value. However, this has some unintended consequences. As the ϵ value increases beyond around 0.4, there are arcs of the graph which are not collision free. This is because a larger ϵ value will allow points which are not adjacent on a training path to be joined, effectively causing the Reeb graph to shorten the path by cutting corners. In some cases this may cause the graph to cross an obstacle. For example, there

are no valid paths through the graphs of $\epsilon = 1$ and $\epsilon = 2$. These effects are all because the graph we generate is an *approximate* Reeb graph.

See Section 4.6 for a more detailed discussion of using Reeb graphs to capture the useful topology of the space in two dimensions and higher.

4.4.2 7-D domains

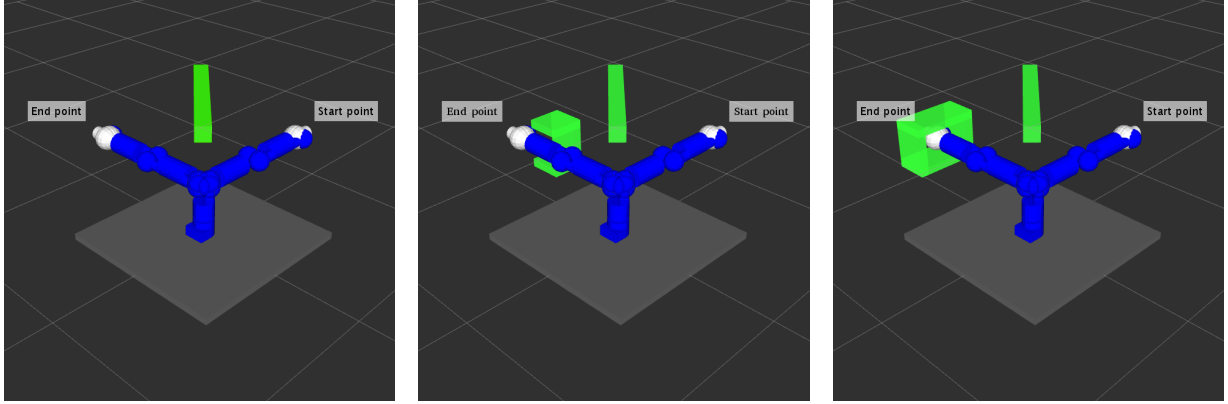
We now test the DRM-connect algorithm in a higher dimensional space, where the curse of dimensionality severely limits the usefulness of RRT and other sample-based path planning variants as on-line planners [Bellman, 1957]. We construct three 7-D domains, shown in Figure 4.6, in increasing domain complexity from left to right. In each, a Kuka arm must move its base joint through 90° , from q_s at $(0, \frac{\pi}{2}, 0, 0, 0, 0, 0)$ to q_g at $(\frac{\pi}{2}, \frac{\pi}{2}, 0, 0, 0, 0, 0)$, while avoiding obstacles placed in the way. The 7-D state q refers to the joint angles of the arm, from the base toward the end-effector. The even-numbered dimensions are the elbow joints, while the odd-numbered dimensions twist the segments between the elbow joints.

The BAR domain (Figure 4.6a) has a single bar around which the arm must navigate. The CUP domain (Figure 4.6b) extends the BAR domain by including a C-shaped bracket around the end point, which increases the complexity of the domain. The BOX domain (Figure 4.6c) also extends the BAR domain, but includes a box around the end point, which significantly increases the domain complexity.

Since RRT becomes prohibitively slow for testing purposes in higher dimensions, we instead compare the performance of DRM-connect against RRT-connect. In the learning phase, twelve paths are generated using RRT-connect.

The implementation of RRT-connect used is the same as DRM-connect, except that an empty graph is supplied. This implementation is inefficient compared to state-of-the-art implementations, but allows us to test the algorithms more fairly. We assume that a more efficient implementation of DRM-connect would provide similar relative results as those below, if compared to an efficient RRT-connect algorithm.

Again, these are contracted into a Reeb graph.



(a) 7-D BAR domain. The green bar is an obstacle which lies in the plane of the swept volume of the arm directly between the start and end poses, requiring the arm to move above, below or otherwise contort to navigate around it.

(b) 7-D CUP domain, an extension of the BAR domain, but with a C-shaped bracket (the cup) increasing the domain complexity.

(c) 7-D BOX domain, an extension of the BAR domain, but including a box around the final end effector position, requiring the arm to contract before extending into the box, vastly increasing the domain complexity.

Figure 4.6: The various 7-D domains. These are in increasing order of task complexity, from left to right. In all tasks, the arm is tasked with moving from the start point to end point, requiring a rotation of 90° of the base joint.

To test the performance of DRM-connect in these domains, we carry out the following experiment in each. We first generate a Reeb graph (generating the training paths with RRT-connect) as described in Section 4.3.1, then use DRM-connect to find a valid trajectory from q_s to q_g . A small obstacle (A sphere of radius $0.1m$, approximately the size of the joints of the robot) is then inserted $n\%$ of the way through the trajectory, and both DRM-connect and RRT-connect are used to replan the trajectory. Values for n range from 15 to 85 in steps of 5. We test a smaller range of percentages at which obstacles are inserted since it not always possible to place the sphere in such a way as to block the path, yet not make planning impossible for the extremes of the percentage range. The path percentage is calculated as:

$$n = (p_c - p_s) \cdot \frac{p_g - p_s}{|p_g - p_s|} \quad (4.1)$$

where p_c , p_s and p_g are the current, start and goal positions of the end effector respectively. This is just the projection of the displacement of the end effector from the start ($p_c - p_s$) onto the vector from q_s to q_g ($p_g - p_s$).

Alternative metrics could be:

1. Fraction of edges executed to the total number of edges in the path.
2. Fraction of the swept length or volume of the arm/end effector.

Both of these approaches depend significantly on whether the path in question is fairly direct, or whether there are spurious movements. The important features of this metric are that it's intuitive and that comparisons between two replanning episodes with the same metric value are fair. This is why path extent is measured as the vector projection in Equation 4.1, which is an intuitive description of how far through the task we are.

However, it has some drawbacks. In any cases where the required motion of the arm extends beyond the end point (for example in the CUP domain), this metric would have values greater than 1 for these positions, which would never be evaluated in the experiments below. This process (generating training paths, obstructing the path through the task and testing replanning for all percentages) is repeated 10

times.

To calculate the break-even point (the number of replanning calls required on average to offset the overhead of generating the training paths and building the Reeb graph) of using DRM-connect instead of RRT-connect, we use the following formula:

$$n_b = \frac{t_R + n_{train} \times t_{RRTc}}{t_{RRTc} - t_{DRMc}} \quad (4.2)$$

where n_b is the number of replanning episodes required to offset the overhead cost of generating the Reeb graphs, t_R is the average time required to generate the Reeb graph from the training paths, n_{train} is the number of training paths (generated using RRT-connect), t_{RRTc} is the average time required by RRT-connect to replan a path and t_{DRMc} is the average time required by DRM-connect to replan a path. For all the experiments below, we assume that replanning will occur for a uniform distribution over the path extent, so t_{RRTc} and t_{DRMc} are simply the average replanning time over the entire path extent.

Generating the Reeb graph from the training paths takes approximately 3.5 seconds (this is t_R in Equation 4.2). All 7-D tests are run on the EXOTica framework [Ivan et al., 2018].

4.4.3 The BAR domain

The first 7-D domain tested is the BAR domain. This domain is shown in Figure 4.6a. The bar is placed horizontally such that it blocks the arm from simply rotating its base joint 90° , as RRT-connect is likely to do. Additionally, it will force the RRT-connect planner to navigate above or below (or contort around) the bar. This serves the purpose of encouraging the planner to generate paths in separate trajectory classes. For this domain, there exists a subspace, containing the two joints closest to the base, on which all planning can happen. The ability of RRT-connect to attempt long, straight connections (in the configuration space) should indirectly exploit this fact. As such, this domain has a fairly low task complexity. Although neither the RRT-connect or DRM-connect planners are specifically capable of identifying redundant degrees of freedom, the CONNECT aspect of each of them will exploit clear bands in the configuration space. For example, if the bar weren't present, it is likely that RRT-connect would plan directly to the goal, moving only the base joint through 90° .

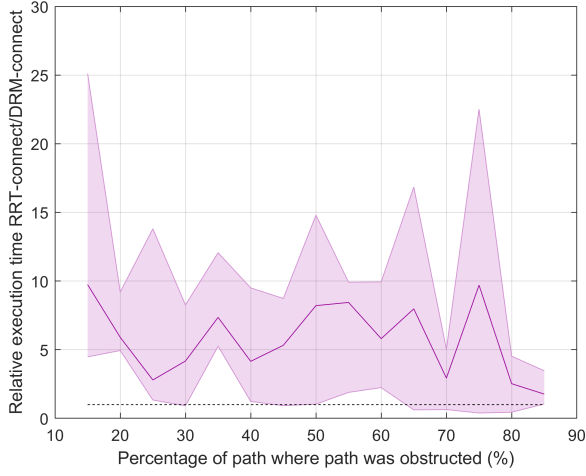
Results and discussion

Figures 4.7 and 4.8 show the results of this experiment.

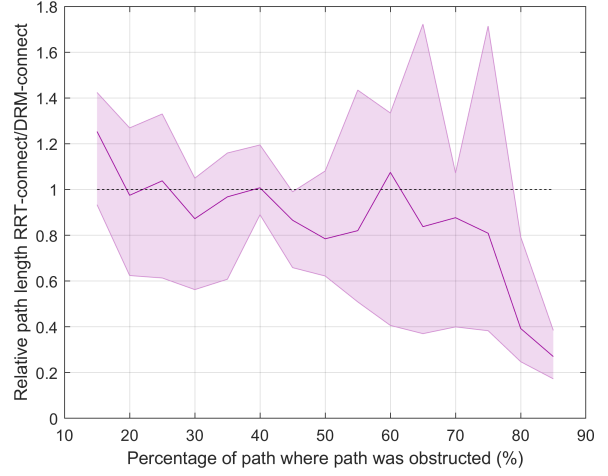
Figure 4.7a shows that, in general, the solve time of DRM-connect is faster than RRT-connect. If the obstruction is early in the task (far from the goal), then DRM-connect replans around 5 or 10 times faster than RRT-connect. As the obstruction occurs later in the task, this performance benefit reduces. However, the IQR is almost always at or above the line $y = 1$, indicating that using DRM-connect should mostly give you equal or better performance than RRT-connect.

Figure 4.8a shows an interesting change at around 50%. The solve times of RRT-connect begin to improve significantly (although the variance also increases). At around this point, the path will be beyond the bar obstruction, and RRT-connect will often be able to plan directly to the goal using its CONNECT step. DRM-connect doesn't have the same advantage, since it still attempts to use the graphs on which it relies, so its performance doesn't really improve for interruptions further through task execution.

The length of paths returned by DRM-connect vary quite a lot compared to RRT-connect (see Figure 4.8b). For obstructions early in the path ($x \leq 50\%$) the paths found by DRM-connect and RRT-connect are around the same length (although DRM-connect has a higher variance). The length of paths returned by DRM-connect gradually increases the later in the path the execution is interrupted. This is because the further through the task we are, the more backtracking will be required to find a new path through the Reeb graph. Section 4.6 contains a more detailed discussion of whether the Reeb graphs have encoded useful information about the space. Additionally, the path length of RRT-connect past 50% begins to reduce dramatically. This is for the same reason that the solve time decreases - once the arm

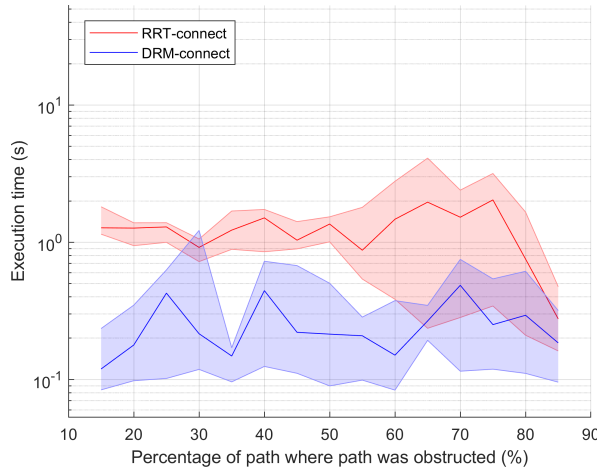


(a) Relative execution time between RRT-connect and DRM-connect. A larger value indicates better performance of DRM-connect.

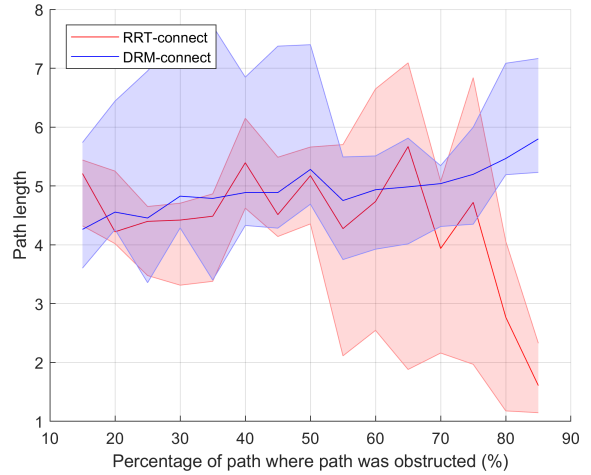


(b) Relative path length between RRT-connect and DRM-connect. A larger value indicates a shorter path found by DRM-connect.

Figure 4.7: Relative execution time and path length of RRT-connect/DRM-connect for the 7-D BAR domain. The x -axis shows how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are the median values from 10 runs, with the shaded region representing the interquartile range (10 sets of example paths, Reeb graphs and obstructions). The dotted lines indicate a relative value of 1 (parity of the two algorithms' performance).

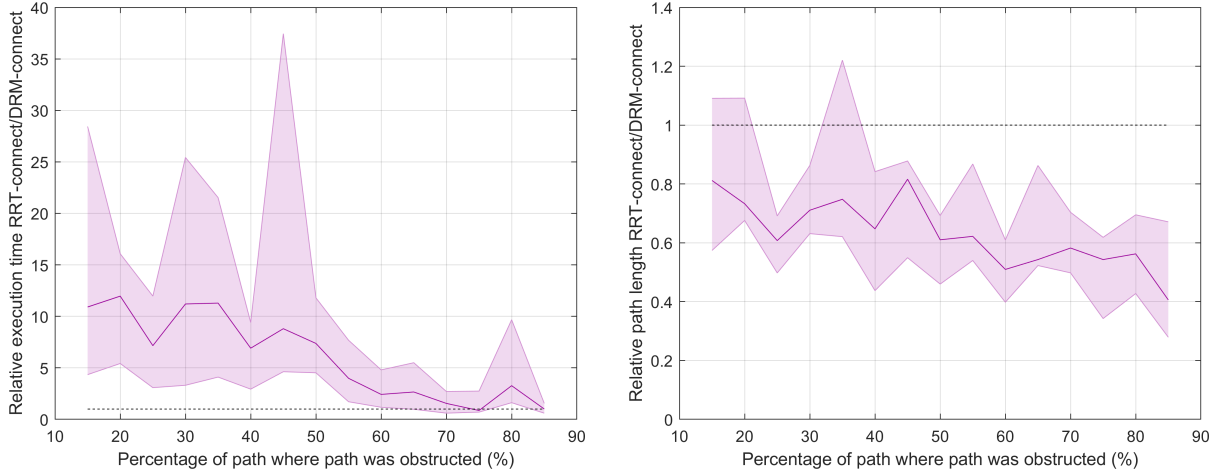


(a) Comparison of execution time between RRT-connect and DRM-connect.



(b) Comparison of path length between RRT-connect and DRM-connect.

Figure 4.8: Execution time and path length of RRT-connect and DRM-connect for the 7-D BAR domain. The x -axes show how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are median values from 10 runs, with the shaded regions representing the interquartile range (10 sets of example paths, Reeb graphs and obstructions).



(a) Comparison of execution time between RRT-connect and DRM-connect. A larger value indicates better performance of DRM-connect.

(b) Comparison of path length between RRT-connect and DRM-connect. A larger value indicates a shorter path found by DRM-connect.

Figure 4.9: Relative execution time and path length of RRT-connect/DRM-connect for the 7-D CUP domain. The x -axis shows how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are the median values from 10 runs, with the shaded region representing the interquartile range (10 sets of example paths, Reeb graphs and obstructions). The dotted lines indicate a relative value of 1 (parity of the two algorithms' performance).

is past the bar, RRT-connect will often plan a direct path to the goal, which will be both short and efficient.

The average replanning time of RRT-connect (t_{RRTc}) is $1.34s$, while the average replanning time of DRM-connect (t_{DRMc}) is $0.52s$. From Equation 4.2, the number of replanning episodes after which the overhead of generating paths and building the Reeb graph is amortised is $n_b = 24.0$. If either the number of replanning episodes is expected to be 24 or greater, or the reduced replanning time is required for a more responsive robot, then DRM-connect is a sensible choice in this domain. These values hold only for this task in this domain, and assume that the Reeb graph has to be rebuilt each time new trajectories are added as training data. The development of an online update method for the underlying graph is left as future work.

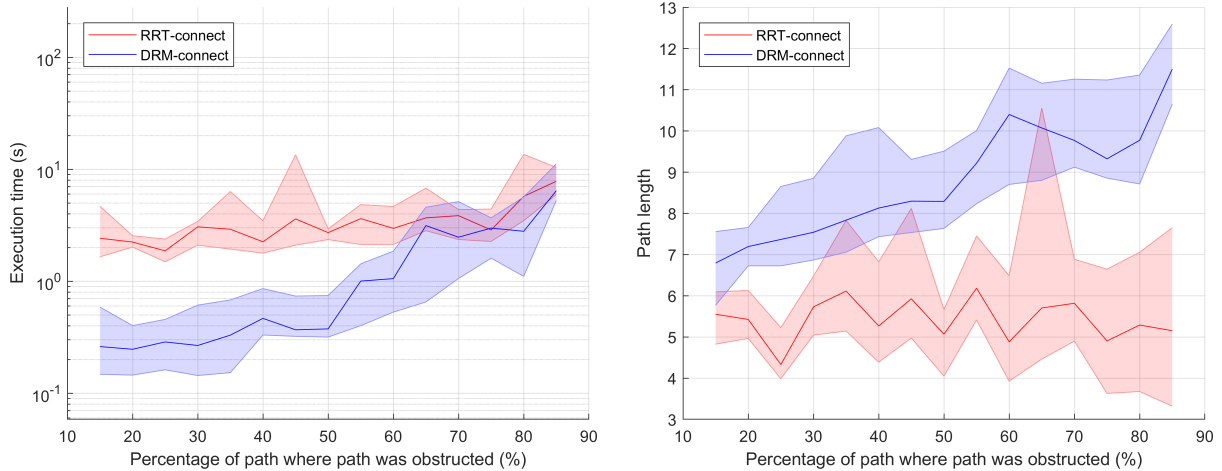
4.4.4 The CUP domain

The next domain tested is the CUP domain. This domain is shown in Figure 4.6b. The cup prevents RRT-connect from sampling randomly until it has line of sight of the goal, and then rapidly moving directly to it. There is still a 2-D subspace on which all planning can happen (the two joints closest to the base), so this domain has a higher task complexity than the BAR domain, yet is still only moderately complex.

Results and discussion

Figures 4.9 and 4.10 show the results of this experiment.

Figure 4.9a indicates that DRM-connect replans around 10 times faster than RRT-connect when the task execution is interrupted early, with the relative benefit of using DRM-connect reducing to around 1 (parity with RRT-connect) when the task is interrupted late. Figure 4.10b shows similar behaviour of DRM-connect compared to the BAR domain, in that the length of the replanned paths increases for tasks interrupted later, again due to backtracking along the graph to find a new path. Even if a valid path through the Reeb graph doesn't exist, the repaired graph is still built from the existing Reeb graph, so the same amount of backtracking may be required, even if PRM-connect is called to repair the graphs. This may indicate that the ϵ value of the Reeb graph is too large, or that insufficient training paths were



(a) Comparison of execution time between RRT-connect and DRM-connect. (b) Comparison of path length between RRT-connect and DRM-connect.

Figure 4.10: Execution time and path length of RRT-connect and DRM-connect for the 7-D CUP domain. The x -axes show how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are median values from 10 runs, with the shaded regions representing the interquartile range (10 sets of example paths, Reeb graphs and obstructions).

used.

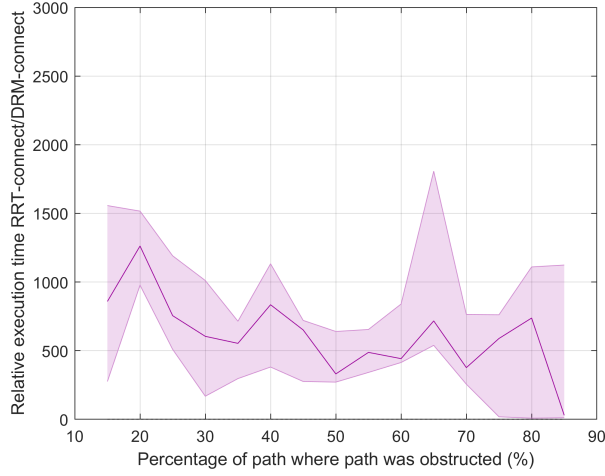
Given that the task is significantly more complex than the BOX domain, DRM-connect doesn't seem to have improved significantly in relation to RRT-connect. Qualitatively these two domains are quite similar. In both cases RRT-connect only has to move past an obstacle before connecting straight to the goal. All that has changed is that the region from which the goal is visible is smaller, and has only been restricted in the configuration space of the first 2 base joints.

The average replanning time of RRT-connect (t_{RRTc}) is $4.5s$, while the average replanning time of DRM-connect (t_{DRMc}) is $3.0s$. From Equation 4.2, the number of replanning episodes after which the overhead of generating paths and building the Reeb graph is amortised is $n_b = 38.9$. If either the number of replanning episodes is expected to be 39 or greater, or the reduced replanning time is required, then DRM-connect is a sensible choice in this domain. The similar performance of RRT-connect and DRM-connect when the path is interrupted close to the goal (likely due to the choice of ϵ and the presence of the 2-D latent subspace) impacts heavily on the break-even point above, making it higher than for the less complex BAR domain. This break-even point would be lower if replanning were expected to be further from the goal, and much higher if replanning were expected to be closer to the goal.

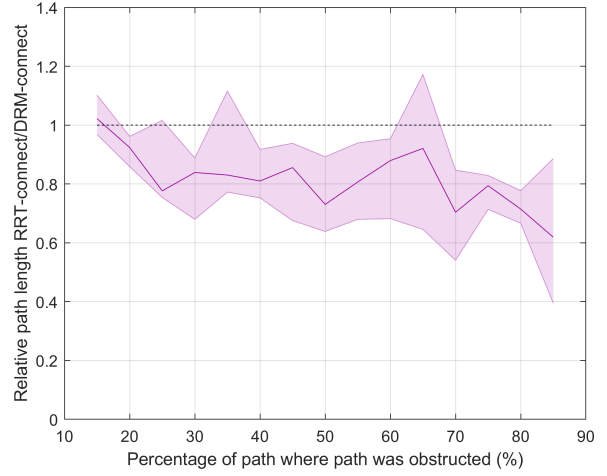
4.4.5 The BOX domain

The next domain tested is the BOX domain. This domain is shown in Figure 4.6b. Both of the previous domains contained a subspace on which planning could be carried out. The BOX domain drastically increases the task complexity by preventing planning from happening on this subspace (there is still a subspace, yet it is higher dimensional). To achieve this, we modify the BAR domain by placing a box around the goal configuration. The result of this is that the arm can no longer navigate from q_s to q_g without using some of its joints further away from the base. The arm must now contort and then reach into the box. We test various ϵ values when constructing the Reeb graphs: 0.11, 0.15, 0.20.

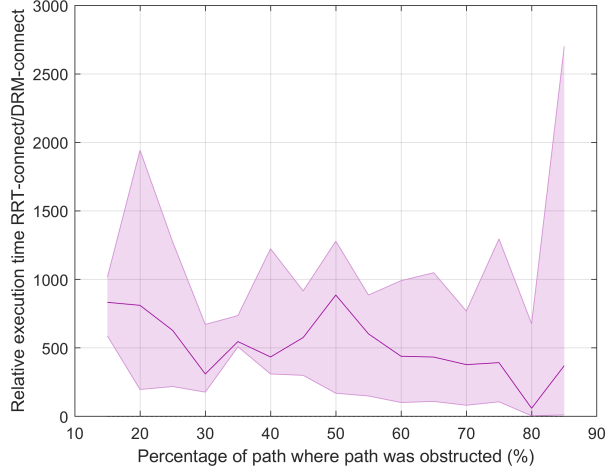
We use 0.11 since this is slightly larger than the minimum value allowed (see Section 4.4.1). Since this is close to the minimum allowable ϵ value ($\epsilon > r$), we chose values to compare which are slightly larger.



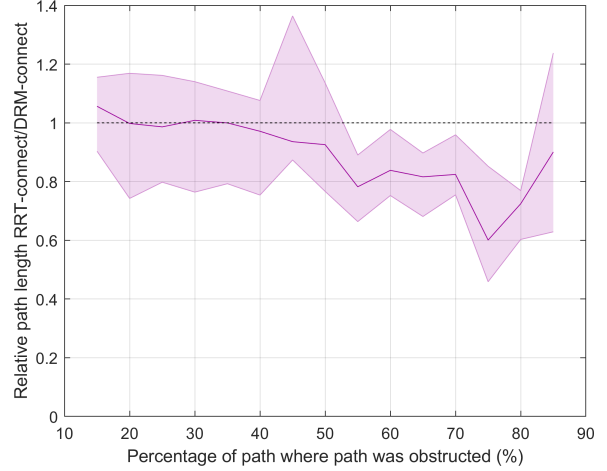
(a) Relative execution time between RRT and DRM-connect for $\epsilon = 0.11$.



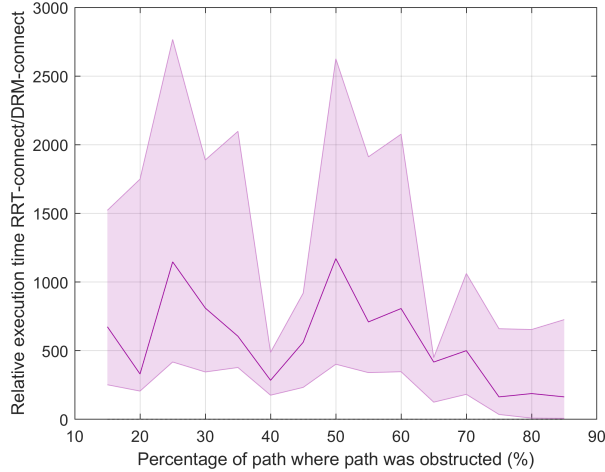
(b) Relative path length between RRT and DRM-connect for $\epsilon = 0.11$.



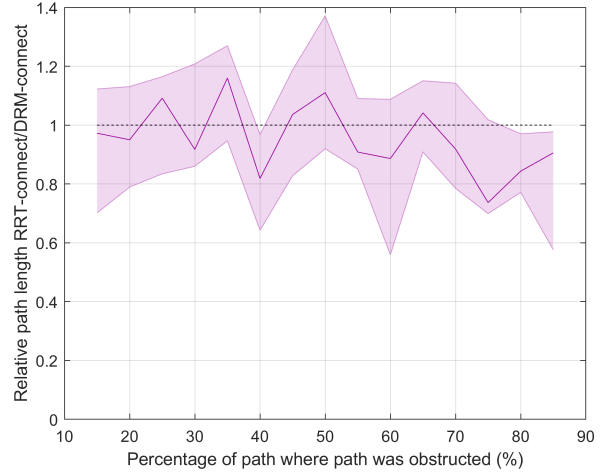
(c) Relative execution time between RRT and DRM-connect for $\epsilon = 0.15$.



(d) Relative path length between RRT and DRM-connect for $\epsilon = 0.15$.

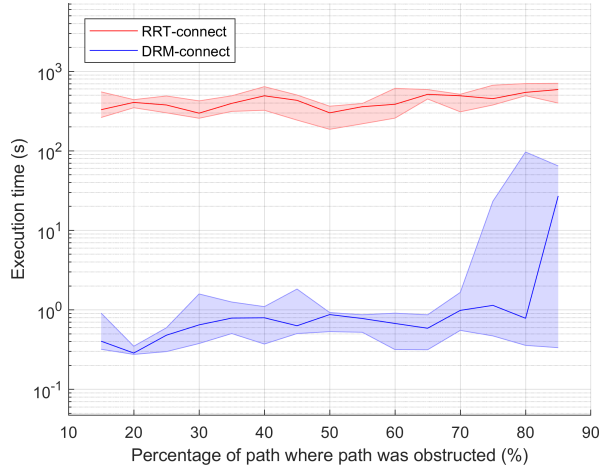


(e) Relative execution time between RRT and DRM-connect for $\epsilon = 0.20$.

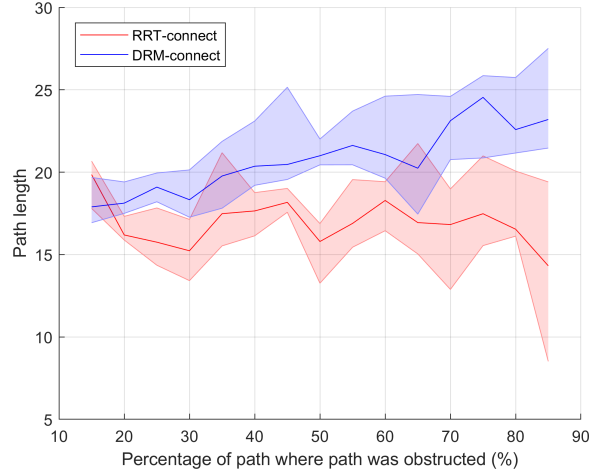


(f) Relative path length between RRT and DRM-connect for $\epsilon = 0.20$.

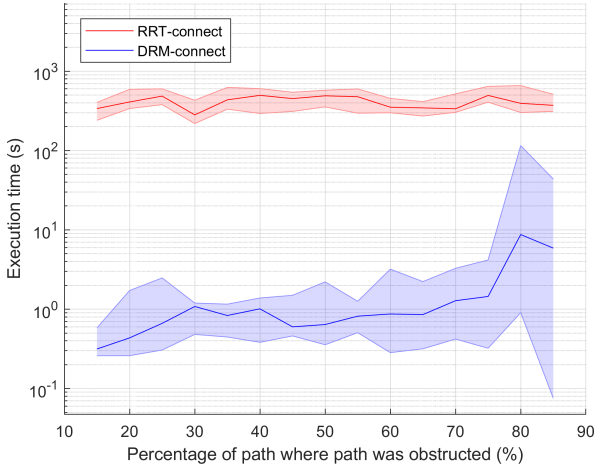
Figure 4.11: Relative execution time and path length of RRT-connect/DRM-connect for the 7-D Box domain. The x -axis shows how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are the median values from 10 runs, with the shaded region representing the interquartile range (10 sets of example paths, Reeb graphs and obstructions). The dotted lines indicate a relative value of 1 (parity of the two algorithms' performance).



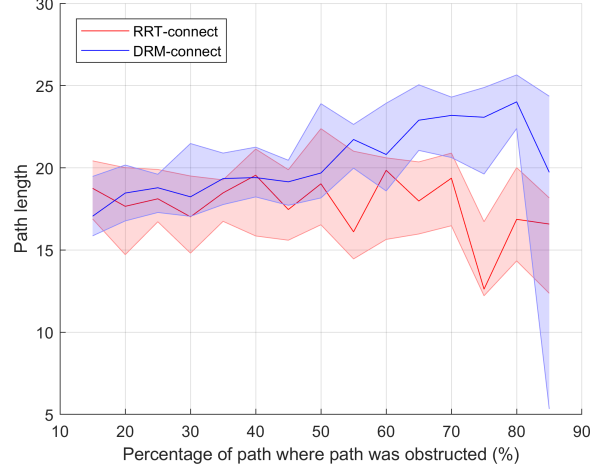
(a) Comparison of execution time between RRT-connect and DRM-connect for $\epsilon = 0.11$.



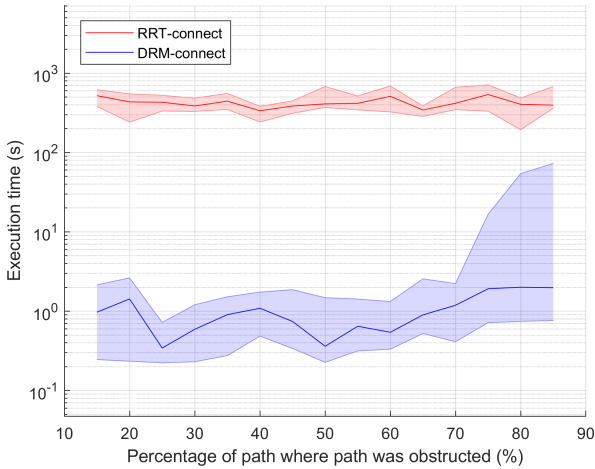
(b) Comparison of path length between RRT-connect and DRM-connect for $\epsilon = 0.11$.



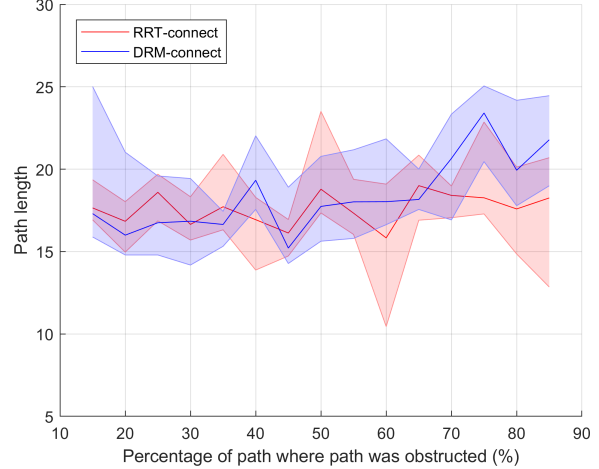
(c) Comparison of execution time between RRT-connect and DRM-connect for $\epsilon = 0.15$.



(d) Comparison of path length between RRT-connect and DRM-connect for $\epsilon = 0.15$.



(e) Comparison of execution time between RRT-connect and DRM-connect for $\epsilon = 0.20$.



(f) Comparison of path length between RRT-connect and DRM-connect for $\epsilon = 0.20$.

Figure 4.12: Execution time and path length of RRT-connect and DRM-connect for the 7-D Box domain. The x -axes show how far into execution of the path the robot is obstructed (as a percentage of the entire path). The results shown here are median values from 10 runs, with the shaded regions representing the interquartile range (10 sets of example paths, Reeb graphs and obstructions).

ϵ	t_{RRTc} [s]	t_{DRMc} [s]	n_b
0.11	440	14.9	12.4
0.15	428	13.7	12.4
0.20	461	9.26	12.3

Table 4.1: Average solve times for DRM-connect and RRT-connect and break-even point in the BOX domain.

Results and discussion

Figures 4.11 and 4.12 show the results of this experiment.

For all three ϵ values, the performance of DRM-connect is approximately 500 times better than RRT-connect when the task execution is interrupted early (see Figures 4.11a, 4.11c and 4.11e), while returned paths are slightly longer (see Figures 4.11b, 4.11d and 4.11f). We see that the relative performance of DRM-connect compared to RRT-connect reduces slightly as the path is interrupted later, in the same way as in the BAR and CUP domains.

When the task is interrupted 80% or 85% of the way through execution (independent of the ϵ value), the execution time of DRM-connect suddenly increases. Since the box requires that all approaches to q_g come from a similar direction, it's likely that the obstacles at these values are entirely severing the Reeb graph. This would require replanning using PRM-connect, which would have an execution time similar to that of RRT-connect (perhaps slightly faster if parts of the Reeb graph are reusable). This is consistent with these results.

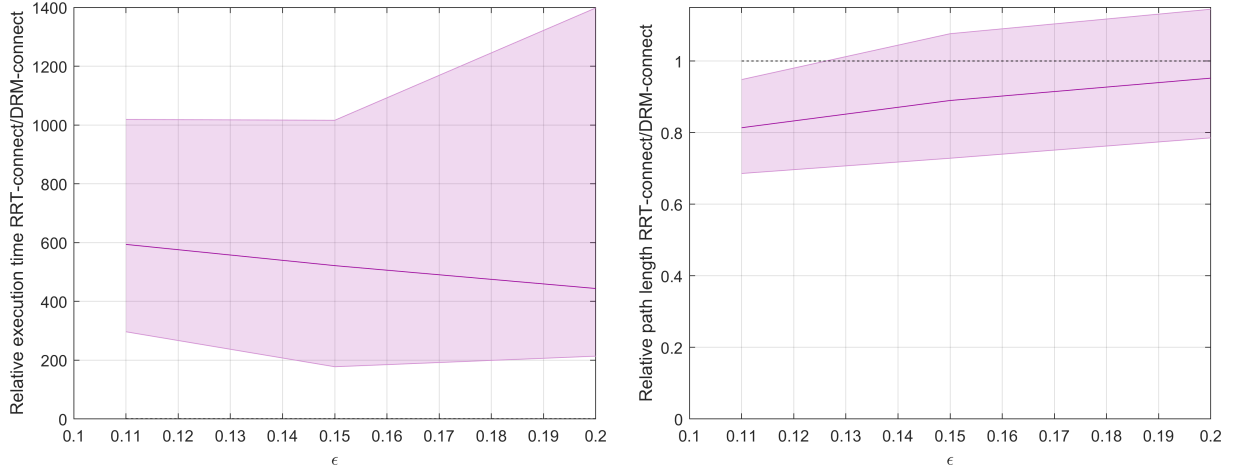
The path length of DRM-connect seems to be slightly affected by the ϵ value used. For $\epsilon = 0.11$, the relative path length of DRM-connect to RRT-connect reduces the later the task execution is interrupted (see Figure 4.11b). As ϵ is increased (see Figures 4.11d and 4.11f) the path length returned by DRM-connect gets closer to the path length returned by RRT-connect. As the ϵ value is increased, we expect the Reeb graph to become denser. This would allow for more efficient backtracking, which would on average reduce the replanned path length.

The ϵ value has little discernible effect on the replanning times. This suggests that of these 3 ϵ values, $\epsilon = 0.20$ is probably the best choice, since we see a significant speed-up, with a minimal effect on the returned path length.

Table 4.1 shows the average time for replanning for RRT-connect and DRM-connect for each tested ϵ value, as well as the number of replanning episodes required to break even. In each case, the number of replanning episodes required to amortise the overhead cost of generating the training paths and building the Reeb graphs is approximately 12. In this domain, the difference in time for replanning between DRM-connect and RRT-connect is very large (around 440s for RRT-connect compared to 14s for DRM-connect). Regardless of the upfront cost, this time saving is significant - it could be the difference between a moderately responsive robot, and an unresponsive one.

Figures 4.13 and 4.14 show the relative and absolute performance of DRM-connect and RRT-connect in the BOX domain for the 3 ϵ values tested. Figures 4.14a and 4.13a indicate that the replanning time of DRM-connect compared to RRT-connect doesn't change significantly for the ϵ values tested. There is a slight downwards trend in the relative execution time as the ϵ value increases, although the IQR also increases, so this trend may not be statistically significant.

There is the opposite trend in path length (see Figure 4.13a), where for small ϵ values the relative path length is around 0.8, but as we increase the ϵ value, this approaches one. As the ϵ value is increased, training paths which are further apart are more likely to be joined, so there may be shorter loops along which DRM-connect can backtrack. Since this may also result in a few number of collision checks being required, we may expect that the replanning time also decreases. However, more connections would also result in a larger graph to search (and more edges added which are not collision free to begin with). It may



(a) Comparison of execution time between RRT-connect and DRM-connect. A larger value indicates better performance of DRM-connect.

(b) Comparison of path length between RRT-connect and DRM-connect. A larger value indicates a shorter path found by DRM-connect.

Figure 4.13: Relative execution time and path length of RRT-connect/DRM-connect for the 7-D Box domain for various ϵ values. The results shown here are median values from 150 runs, interrupted at the path extents in Figure 4.11, with the shaded regions representing the interquartile range. The dotted lines indicate a relative value of 1 (parity of the two algorithms' performance).

be that the additional cost of searching a larger graph (with some edges in collision) does not outweigh the benefit of needing to check fewer collisions.

4.5 Comparison of 7-D domains

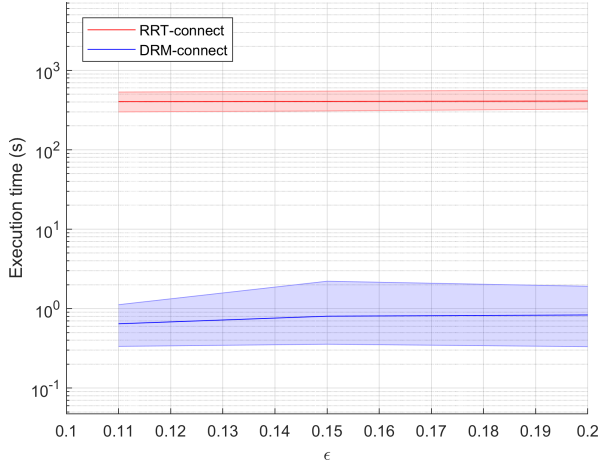
We now compare the performance of DRM-connect to RRT-connect across the 7-D domains. Figure 4.15a shows the relative average replanning time for each 7-D domain tested, in order of increasing domain complexity, from left to right. For each domain, we have 150 samples (450 for the BOX domain). We average over all the path extents at which the path was interrupted. This gives the expected value of the replanning time, assuming that the paths will be interrupted uniformly through execution. Figure 4.15b shows the same summary, but for the path lengths.

For the BAR and CUP domains, the average improvement in replanning times is around 1 order of magnitude, with a larger IQR, while the improvement for the BOX is around 3 orders of magnitude, also with a large IQR. The conclusion from this is the benefits of DRM-connect become more significant as the task complexity increases.

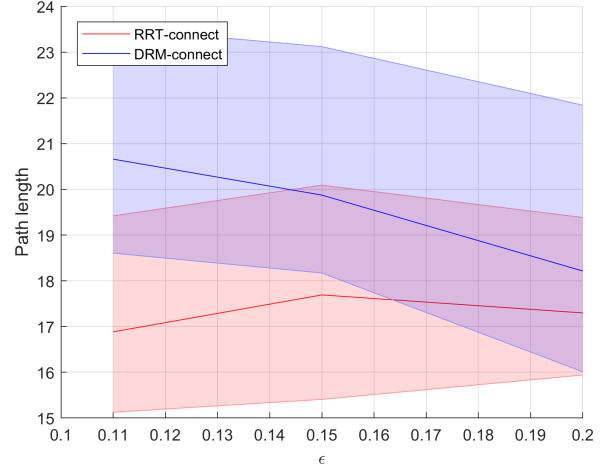
The average relative path length shows no significant relationship between the domain complexity and the relative path lengths between DRM-connect and RRT-connect. Again, the IQR of the path lengths of DRM-connect, relative to RRT-connect, is fairly large.

4.6 Reeb graph discussion

The DRM-connect algorithm has been presented as a novel method of leveraging a graph for replanning in dynamic environments. As discussed in Section 3.4, we prescribe no specific graph to be used with DRM-connect. The choice of graph is left as a choice to complement the specific environment or agent. We have tested DRM-connect with two types of graphs. The first of these is a naïve graph (see Section 3.5.1). The second is a Reeb graph (the focus of this chapter).

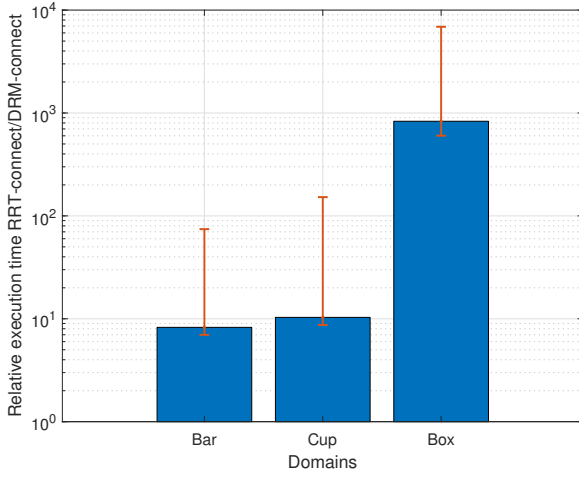


(a) Comparison of execution time between RRT-connect and DRM-connect.

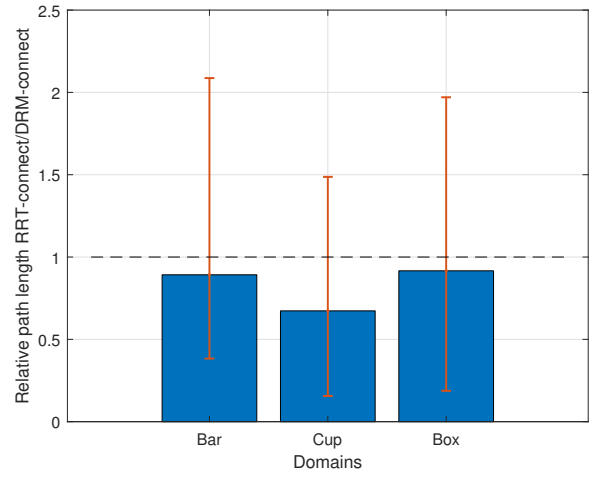


(b) Comparison of path length between RRT-connect and DRM-connect.

Figure 4.14: Execution time and path length of RRT-connect and DRM-connect for the 7-D Box domain for various ϵ values. The results shown here are median values from 150 runs, interrupted at the path extents in Figure 4.12, with the shaded regions representing the interquartile range.



(a) Comparison of execution time between RRT-connect and DRM-connect. A larger value indicates better performance of DRM-connect.



(b) Comparison of path length between RRT-connect and DRM-connect. A larger value indicates a shorter path found by DRM-connect.

Figure 4.15: Average relative execution time and path length of RRT-connect/DRM-connect across the 7-D domains. The domains are placed in order of increasing complexity from left to right. The bars indicate the interquartile range.

We discuss now how effective the use of Reeb graphs have been for the domains we’ve tested, and extend this to a more general discussion of how useful capturing an approximation of the 1-skeleton of $\mathcal{C}_{free}$ is for the types of replanning scenarios presented in this work.

Recall from Section 3.5.1 that using a naïve graph for the DOORS domain caused the replanning time and path length of DRM-connect to increase significantly as the path was interrupted closer to the goal. This was because the individual training paths were only connected at the start and end points. From Section 4.4.1 we saw that the use of a Reeb graph has improved the replanning time (from approximately 8 times faster at 5% using a naïve graph to approximately 10 times faster with a Reeb graph and from approximately 0.5 times at 95% to approximately 2 times faster).

Additionally, the path length returned by DRM-connect with a Reeb graph is more similar to the path length returned by RRT - although occasionally the path is significantly longer.

For the 2-D DOORS domain, using a Reeb graph has definitely offered an improvement. In 2-D spaces, the advantage that a Reeb graph offers is this: If two training paths cross, then the Reeb graph (with an appropriate ϵ value) will create a junction which connects the two training paths (or contractions of them). The advantage of this is that extra loops are created in the graph, which DRM-connect is able to exploit to provide both shorter paths, and because of this, quicker solution times.

The nature of 2-D spaces, however, is that 1-D structures (such as paths and trees) are forced to intersect each other exactly, assuming they’re expansive enough. This raises the question of whether Reeb graphs will still be an effective method of encoding useful information about planning in higher dimensions, since it’s possible (and likely, given randomised path planning) for two paths to start and end at the same point, and never intersect each other elsewhere. More than just the fact that 1-D paths in higher dimensions are very unlikely to intersect exactly is that as the dimensionality increases, the sampled points become more and more sparse.

One of the responsibilities of the ϵ value is to allow paths in higher dimensional spaces to be joined together as a junction in the Reeb graph, to reduce the amount of backtracking required. The trouble is that the ϵ value affects two characteristics of the resulting Reeb graph:

1. Whether a bridge is created between two paths a distance x apart.
2. Which points on a single path are joined together.

As ϵ is increased, points further and further apart on a path are joined together. These edges are used to create the final Reeb graph, so there is an increasing chance that edges will be created which ‘cut corners’ (are always in collision).

In the 7-D domains, we see a mixed response to using a Reeb graph. In general, the performance and path length of DRM-connect increases as the path is interrupted later, which suggests that there is still significant backtracking occurring. However, for none of the 7-D domains is the path length ever twice as long as when paths are interrupted early (which would indicate backtracking to the start point). This indicates that at least some bridges are built between the different trajectories, but the returned paths still require more backtracking than is desirable.

4.7 Conclusion

In this chapter we proposed using Reeb graphs with DRM-connect, as a way of encoding useful topological information about the environment from prior experience. We then tested this approach in the same 2-D navigation task as in Section 3.5.1 and several 7-D reaching tasks. This approach shows better performance (both execution time and path length) than using a naïve graph, and in most cases provides better replanning times than RRT-connect, by several orders of magnitude, with minimal impact on the resulting path lengths.

In general, the execution speed-up is greater when replanning occurs far from the goal, and closer to the performance of RRT-connect when close to the goal. The path length returned by DRM-connect is typically similar to RRT-connect for replanning episodes far from the goal, while it performs worse when replanning occurs close to the goal, due to the need for backtracking.

Using a Reeb graph successfully encoded many of the task relevant trajectories in all domains. The cases where the path length of DRM-connect was significantly worse than RRT-connect indicate that significant backtracking is occurring. This is likely because of the sparsity of the trajectory samples in high dimensions. These cases indicate that the limited samples have prevented the Reeb graph from representing the true topology in this region. An approach to modifying the Reeb graphs by specifically looking to join sparse trajectories may be valuable. This is left to future work.

DRM-connect with Reeb graphs has been shown to work best in high dimensional domains with high task complexity.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Path replanning in dynamic environments with high dimensional configuration spaces remains an unsolved problem, the satisfactory solution of which would be immensely valuable in the field of robotic path planning.

We presented a probabilistically complete, topologically-inspired approach to graph-based replanning, termed DRM-connect, which extends DRM and RRT-connect, by attempting to replan on the current graph, checking collisions lazily. If this fails (a valid path does not exist through the graph), DRM-connect falls back to a modified RRT-connect to repair the graph.

We validated DRM-connect experimentally in 2-D and 7-D domains, using both naïve and Reeb graphs. DRM-connect with naïve graphs was shown to replan faster than RRT in 2-D, except when the interruption was close to the goal (see Section 3.5.1), although it generally returned significantly longer path lengths than RRT. When a Reeb graph was used, the replanning times in 2-D remained fairly similar, although the path lengths returned were more similar to RRT, indicating that the use of a Reeb graph helped to find local loops to exploit, rather than the significant backtracking required when using naïve graphs. See Section 4.6 for a full treatment of this.

For the 7-D tasks, DRM-connect was shown to perform well when the replanning was required far from the goal. When interrupted close to the goal, its performance was shown to be more similar to RRT-connect. The path lengths returned by DRM-connect were typically slightly longer than RRT-connect, especially when the task execution was interrupted near the goal. DRM-connect was shown to work especially well in domains with a higher task complexity, offering an improvement in runtime of 3 orders of magnitude over RRT-connect.

5.2 Future work

All tests executed here used a fixed volume of training data to build the Reeb graphs. The effect of varying the volume of training data supplied to the Reeb graph has not been explored. It would be illuminating to discover what the tradeoff between data volume and execution time is, and how significantly this affects the length of paths returned.

Additionally, another area which would be valuable to explore would be to develop an approach to updating the graph underlying DRM-connect (Reeb or other) online. This would allow for lifelong learning, as well as enabling DRM-connect to cope with environments which change substantially over time.

One of the main reasons for using a Reeb graph instead of a naïve graph is that the Reeb graph is a convenient mechanism to connect separate paths together in local loops, to aid with redundancy. It may

be that a more direct approach to this may be sufficient, rather than using the heavy machinery of Reeb graphs and topological path planning. A simpler approach would be to retain the training paths as is, and then simply connect the paths to one another whenever the distance between them is less than some threshold distance. For the low data examples tested here, this may be sufficient, since the contracting effect of the Reeb graph generation has little impact. However, for lifelong learning (or cases where a large number of paths are available), the ability of the Reeb graph generation to simplify and contract the graph would be valuable.

Further, this research has tested only a small range of domains, tasks and obstacle configurations, so testing DRM-connect on a wider range of these would be very valuable to further validate the algorithm. Additionally, DRM-connect has not been tested in a truly dynamic task (where the obstacles move during planning and execution). This has only been tested by proxy.

Currently, DRM-connect carries out all collision checks lazily (and does not check for collisions along the Reeb graph when reducing it). This motivates for two more future avenues for exploration. The first of these is to implement a mapping between the configuration space and the task space, such that collision checking during DRM-lazy would become a lookup, rather than requiring simulation. This may significantly improve replanning times. The second option then is to test for collisions along the contracted Reeb graph, to ensure that these contractions do not introduce infeasible arcs. However, this may not always be possible, since there may not be a model of the environment which corresponds to the training paths at the time they were generated (since this may be far in the past).

Additionally, further experiments comparing the performance of DRM-connect to other fast path planning algorithms would be very valuable.

Bibliography

- Bellman, R. (1957). *Dynamic programming*. Princeton University Press.
- Bruce, J. and Veloso, M. (2002). Real-time randomized path planning for robot navigation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 3, pages 2383–2388.
- Carlsson, G. (2009). Topology and data. *Bulletin of the American Mathematical Society*, 46(2):255–308.
- Edelsbrunner, H. and Harer, J. (2010). *Computational topology: an introduction*. American Mathematical Society.
- Eilenberg, S. (1944). Singular homology theory. *Annals of Mathematics*, pages 407–447.
- Ferguson, D., Kalra, N., and Stentz, A. (2006). Replanning with RRTs. In *IEEE International Conference on Robotics and Automation*, pages 1243–1248.
- Gayle, R., Klinger, K. R., and Xavier, P. G. (2007). Lazy reconfiguration forest: An approach for planning with multiple tasks in dynamic environments. *IEEE Transactions on Robotics and Automation*, pages 1316–1323.
- Ge, X., Safa, I., Belkin, M., and Wang, Y. (2011). Data skeletonization via Reeb graphs. *Advances in Neural Information Processing Systems*, pages 837 – 845.
- Hatcher, A. (2002). *Algebraic topology*. Cambridge University Press.
- Ivan, V., Yang, Y., Merkt, W., Camilleri, M. P., and Vijayakumar, S. (2018). EXOTica: An Extensible Optimization Toolset for Prototyping and Benchmarking Motion Planning and Control. In *ROS*. Springer, volume 3 edition.
- Jaillet, L. and Siméon, T. (2004). A PRM-based motion planner for dynamically changing environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, pages 1606–1611.
- Jordan, M. and Perez, A. (2013). Optimal bidirectional rapidly-exploring random trees.
- Kallman, M. and Mataric, M. (2004). Motion planning using dynamic roadmaps. In *IEEE International Conference on Robotics and Automation*, volume 5, pages 4399–4404.
- Karaman, S. and Frazzoli, E. (2010). Incremental sampling-based algorithms for optimal motion planning. *Robotics Science and Systems VI*, 104:2.
- Kavraki, L. E., Svestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE International Conference on Robotics and Automation*, volume 12, pages 566–580.
- 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*, volume 2, pages 995–1001.
- LaValle, S. M. (1998). Rapidly-exploring random trees: a new tool for path planning. Technical report, Computer Science Dept., Iowa State University.
- LaValle, S. M. (2006). *Planning algorithms*. Cambridge University Press.

- Leven, P. and Hutchinson, S. (2000). Toward real-time path planning in changing environments. In *International Workshop on the Algorithmic Foundations of Robotics*, number 9, pages 363–376.
- Murray, S., Floyd-Jones, W., Qi, Y., Sorin, D., and Konidaris, G. (2016). Robot motion planning on a chip. In *Proceedings of Robotics: Science and Systems*, volume XII, page 4.
- Otte, M. and Frazzoli, E. (2016). RRTX: Asymptotically optimal single-query sampling-based motion planning with quick replanning. *The International Journal of Robotics Research*, 35:797–822.
- Pokorny, F. T., Hawasly, M., and Ramamoorthy, S. (2016). Topological trajectory classification with filtrations of simplicial complexes and persistent homology. *The International Journal of Robotics Research*, 35(1-3):204–223.
- Sánchez, G. and Latombe, J.-C. (2003). A single-query bi-directional probabilistic roadmap planner with lazy collision checking. *Robotics Research*, pages 403–417.
- Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic robotics*. MIT Press.
- Van Den Berg, J. P. and Overmars, M. H. (2005). Roadmap-based motion planning in dynamic environments. *IEEE Transactions on Robotics*, 21(5):885–897.
- Zucker, M., Kuffner, J., and Branicky, M. (2007). Multipartite RRTs for rapid replanning in dynamic environments. In *IEEE International Conference on Robotics and Automation*, pages 1603–1609.