On learning task-directed motion plans

by

Sarah J. Finney

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

Author .....
Department of Electrical Engineering and Computer Science
January 5, 2009

Certified by .....
Leslie Kaelbling
Professor
Thesis Supervisor

Certified by .....
Tomás Lozano-Pérez
Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

On learning task-directed motion plans

by

Sarah Finney

Submitted to the Department of Electrical Engineering and Computer Science
on January 5, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Robotic motion planning is a hard problem for robots with more than just a few degrees of freedom. Modern probabilistic planners are able to solve many problems very quickly, but for difficult problems, they are still unacceptably slow for many applications. This thesis concerns the use of previous planning experience to allow the agent to generate motion plans very quickly when faced with new but related problems.

We first investigate a technique for learning from previous experience by simply remembering past solutions and applying them where relevant to new problems. We find that this approach is useful in environments with very low variability in obstacle placement and task endpoints, and that it is important to keep the set of stored plans small to improve performance. However, we would like to be able to better generalize our previous experience so we next investigate a technique for learning parameterized motion plans.

A parameterized motion plan is a function from planning problem parameters to a motion plan. In our approach, we learn a set of parameterized subpaths, which we can use as suggestions for a probabilistic planner, leading to substantially reduced planning times. We find that this technique is successful in several standard motion planning domains. However, as the domains get more complex, the technique produces less of an advantage. We discover that the learning problem as we have posed it is likely to be intractible, and that the complexity of the problem is due to the redundancy of the robotics platform. We suggest several possible approaches for addressing this problem as future work.

Thesis Supervisor: Leslie Kaelbling
Title: Professor

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor

Acknowledgments

Thanks to my co-advisors, Leslie Kaelbling and Tomás Lozano-Pérez, not only for their copious help in research matters, but also for allowing me to distract myself from time to time with teaching responsibilities. Thanks, too, to Teresa Cataldo, who has always made the complicated stuff easy. I also owe a great deal to the rest of the LIS research group, past and present, and in particular to Luke Zettlemoyer, Natalia Hernandez Gardiol, James McLurkin and my research buddies, Meg Aycinena and Emma Brunskill.

I'd also like to thank my family for being so supportive throughout this long process, and Jim Frankel for listening to me complain and for picking up the slack, particularly towards the end. I'm make up for it, I promise. Lastly, thanks to my good friends Giovanna Casey and Diana Reed both for being there when I needed them and for being un-offended when I abandoned them.

Contents

Chapter 1

Introduction

In recent decades, autonomous agents have contributed substantially in many areas of modern life. They have allowed us to explore other planets, increased productivity in industrial settings, and provided life-saving tools for law enforcement and military personnel. So far, truly autonomous robots have been most successful in environments that can be carefully constrained, as on a factory floor, or when the tasks to be performed require a small number of actuators, as in planetary exploration. For more complicated tasks, the control is generally done either completely or in part by a human operator.

As the capabilities of autonomous robots increase, we see the promise of robotic elder care, search and rescue agents, and machines to take care of myriad tedious or noxious tasks. However, these tasks are complicated and dynamic, requiring many actuators and the ability to deal with changing environments. Further, even with state-of-the-art hardware, robots face uncertainty in both their sensors and their actuators. Together these properties add up to a very difficult engineering challenge. In this thesis, we investigate the use of machine learning to adapt general-purpose techniques to specific robotic platforms and environments. The goal is to enhance performance so that otherwise time-consuming computations can be done quickly, allowing the agent to keep up with real-time demands.

An important element for achieving success in such complex endeavors is the ability to take a high-level description of a task and generate low-level motor commands

that obey the constraints imposed by the task goals coupled with the agent's environment. Motion planning provides a framework for doing this transformation, and many successful general-purpose motion planning techniques have been developed. We have chosen to focus on the generation of complete motion plans, rather than the use of local or behavioral controllers alone. Local controllers have proven to be robust to environmental complexities, but they are susceptible to local optima, and so are not well suited to the complicated tasks we would like to address. Nonetheless, such local solutions are often an important piece of a global planner.

Motion planning is particularly important and challenging when the agent has a large number of degrees of freedom to control. There is strong evidence that the time required to solve motion planning problems exactly grows exponentially with the number of the agent's actuators (Reif, 1979; Canny, 1988). Even assuming that the environment is completely known, and that it does not change appreciably over time, motion planning for many degrees of freedom is a hard problem, and solving it often takes more time than is available for an agent acting in the world. If the problem is further compounded by the fact that the state of the environment may change while planning takes place, or that the environment may be only partially or imperfectly sensed, slow planning solutions become unacceptable. If we can speed up the planning process, we could hope to plan repeatedly as we observe changes in or new information about the environment.

Since it is known that complete motion planning solutions are hard to find, much of the recent focus in motion planning has been on probabilistic sampling-based techniques. The development of these techniques has greatly increased the complexity of problems that can be solved in reasonable time. Nonetheless, with a large number of actuators, there are still many problems that cannot be solved in real time. It is on these difficult problems that our work is focused. Since existing general-purpose planners are too slow in solving difficult tasks but perfectly usable for many easier tasks, our aim is to use these planners to generate solutions to planning problems for a specific robot, doing particular types of difficult tasks, and then use these solutions as training data to learn to solve similar problems very quickly.

1.1 Motion planning terminology

Motion planning problems are generally described in terms of a robot's actuators, or degrees of freedom. A full specification of each of the robot's d degrees of freedom is known as a configuration, $c = \langle \phi_1, \phi_2, ..., \phi_d \rangle$, where each ϕ_i might be the angle for a joint, for example. Planning problems are given as a start and goal configuration (c_s and c_g, respectively) in a particular environment, which generally includes obstacles: any non-robot objects with which the robot should not collide. The environment is often assumed to be static, meaning that the obstacles do not move, for the duration of a particular planning problem, even if the environment is actually dynamic. We will discuss ways in which dynamic environments are explicitly addressed in section 2.2.1. The space of all possible robot configurations is known as the robot's configuration space $\mathcal{C}$ (Lozano-Pérez & Wesley, 1979). Some configurations are invalid because they would cause the robot to collide with itself, and other are invalid because of collisions with the obstacles. The space of collision-free configurations is $\mathcal{C}_{free}$. If the entire space of collision-free configurations is known, then the planning problem is reduced to path planning for a point. As we will discuss further in chapter 2, computing the configuration-space free space, or an approximation to it, is an important piece in most motion planning algorithms.

A motion plan, or path, is a sequence of configurations $p = \langle c_1, c_2, ..., c_n \rangle$. The plan is meaningful only in conjunction with some controller for interpolating between configurations, $U(c_1, c_2, t) = c_i$ for $t \in [0, 1]$, where $u(c_1, c_2, 0) = c_1$ and $u(c_1, c_2, 1) = c_2$. This controller is typically a linear interpolator, but more complex controllers can be used, so long as they are deterministic. Given a path and a controller, a complete robot motion from the first to the last configuration is completely specified.

The output from a motion planner is a path that is known to be collision-free in the current environment, or failure to find such a path. A path is collision-free if each of the configurations c_i are collision-free ($c_i \in \mathcal{C}_{free}$), as well as the paths between each consecutive configuration, or path segments: $u(c_i, c_{i+1}, t) \in \mathcal{C}_{free}$ for all $t \in [0, 1]$. It would seem that even verifying that a path segment is collision-free

would be computationally intractable since t is continuous, so there are an infinite number of configurations between the two endpoints. However, for a given robot, it is known that when the robot moves linearly between two configurations c and c', no point on the robot moves more than $\rho \max_{i \in [1,n]} |\phi_i - \phi'_i|$, where ρ is a property of the robot and can generally be computed **??**. This, combined with the ability to compute the smallest distance between the robot and the obstacles, allows us to avoid further subdividing a segment that could not possibly move far enough to cause collision. In practice, a small $\Delta\phi$ is often used to approximate the necessary granularity for collision checking on path segments.

1.2 Learning motion plans

As we will discuss in more detail in section 2.2.3, there are a number of classes of planning problems that are still intractable for existing general motion planning algorithms. Our goal is to focus on these difficult problems and use learning techniques to summarize solutions to these problems so that they can be generated quickly when encountered in the future. Our motivation draws from both practical considerations, and some general observations about the way that people seem to develop motion skills.

As we will argue in chapter 3, both an agent's environment and the tasks it is expected to complete tend to be somewhere between completely static and completely random. Static environments with repetitive tasks lend themselves to offline processing or hand-engineered planning solutions. Completely arbitrary environments require the full power of a general planner. In contrast, if the environment changes over time but contains some regularities, a natural solution is one that learns based on previous experience to solve quickly the types of problems it actually needs to solve in environments it is actually likely to see. We'd like to avoid representing the entire space of problems.

For robotics platforms with more than just a few degrees of freedom, there are typically many possible solutions to a given planning problems. We'd like to abvoid

representing the entire space of solutions. Another motivation behind learning from previous experience is that we don't need to be able to solve each problem in every possible way. Required planning time on hard problems grows exponentially with the number of degrees of freedom that a robot has, but as the number of degrees of freedom increases, so does the redundancy of the robot. Once we have found a solution to one problem, if we are able to adapt that solution to novel but related problems, we don't need to concern ourselves with the myriad other ways that we *could* have solved these new problems. Just as people typically maneuver themselves so that their manipulation task is placed comfortably in front of them, we'd like our robot to develop a repertoire of "comfortable" motions that it can use to solve a wide range of difficult tasks.

A different way to avoid solving the general planning problem is to hand-design controllers for particular platforms and particular problems. This has proven quite successful in the case of industrial robots. However in non-industrial settings, often the task and environment are insufficiently known at design time. Learning allows the system to adapt to specific platforms and environments without knowing their details in advance.

There is also evidence that something like motion plan learning is done by humans. While there is a great deal of controversy among neuroscience researchers surrounding the optimization criteria used in generating motion plans, and the space in which humans plan their motor control, it is clear that predictive planning does take place. Furthermore, there is evidence that internal models are used for planning, and that these models are learned over time given experience (Flanagan, Vetter, Johansson, & Wolpert, 2003; Kawato, 1999; Krakauer, Ghilardi, & Ghez, 1999; Wolpert & Flanagan, 2001). This observation is not intended to suggest that our work is meant to model human motion planning; with so little consensus among researchers in the field, it would seem premature. Rather, it is an inspiration for trying to build a system that can learn from experience in a similar way, since humans are so successful at learning to control our complex, many degree-of-freedom manipulators.

1.3 Thesis overview

In this work, we investigate several approaches to learning from previous motion planning experience, and describe the effectiveness of each approach and the lessons learned for future work. In chapter 3 we start with the most basic type of learning: memorization. We propose a technique designed to limit the amount of memorization needed to store necessary motion plans for environments with regularities, both in the obstacle placement, and in the tasks. We find that we are able to provide significant speedup in domains that have low entropy, both in the endpoint configurations, and in the obstacles placement, and that we can enhance the speedup by taking measures to limit the number of stored paths. However we find that the technique is unsatisfyingly inflexible, and does not allow for the level of generalization to a wide variety of tasks that we would like.

Chapter 4 describes an approach that assumes a parameterization of difficult tasks, and learns a function from task parameters to partial motion plans. The technique addresses the non-functional nature of data generated by probabilistic planners for a redundant robot, and is quite successful on a class of problems for which a discrete set of functions can be discerned. However, we discover a distinct class of problems for which the data is less easily partitioned into separate functions, and for which this technique does less well.

In chapter 5 we go on to further investigate the difficulties raised in the previous chapter. We identify several sources of difficulty in the domains that are mo challenging for the clustering algorithm described in chapter 4. We also discuss several standard machine learning techniques that seem applicable to these challenges, and explain why each does not properly address the problem.

Chapter 2 provides an overview of related work, and chapter 6 overviews the conclusions we draw from our investigations, and provides some suggestions for future work in this area, as well as some preliminary results for an implementation of some of this work.

Chapter 2

Related work

Motion planning has a long history of research. This chapter provides a very brief overview of the motion planning techniques that have been developed, with an emphasis on the more recent innovations (section 2.1). In section 2.2 we discuss approaches that particularly address problems that are related to the focus of our work. Section 2.3 reviews work that applies machine learning to the problem of generating robot motion.

2.1 Background

The notion of configuration space was introduced by Lozano-Pérez (1979), and has been widely used since then. Several exact methods for computing solutions to both general and specific motion planning problems were developed (Donald, 1984; Lozano-Pérez, 1983, 1987; Reif & Sharir, 1994). However, in the general case, such exact solutions are intractable, so approximate representations of the collision-free configuration space were developed. There is work involving both sampling-based and deterministic approximations, but sampling-based techniques have met with a great deal of success and have since become widespread. Initially, these sampling-based techniques were primarily intended for static environments, but as sampling techniques have improved and computers have became more powerful, it has become possible to apply similar sampling-based techniques to solve many problems quite quickly, even in changing

environments. We will provide an overview of several of the more prominent techniques for configuration-space free space approximation here, with a focus on more recent work, and work most closely related to ours. For more detailed descriptions, there are a number of excellent survey papers and textbooks available, particularly for earlier work (Latombe, 1991; LaValle, 2006; Hwang & Ahuja, 1992).

2.1.1 Roadmaps

As discussed in section 1.1, the motion planning problem can be transformed into the simpler problem of planning the motion of a point if the configuration-space free space is known. Thus, many motion planning techniques, particularly those that deal with arbitrarily complicated robots, are primarily concerned with computing and representing the configuration-space free space (or, equivalently, the configuration-space obstacles).

A common structure used to represent the approximate free space of the robot is known as a roadmap. A roadmap is a graph of nodes in which each node represents a collision-free configuration of the robot. Given a deterministic controller for moving the robot between configurations, an edge can be placed in the graph between any pair of configurations for which the motion determined by the controller is collision free. Once a roadmap for a particular environment has been built, it may be possible to find a path between the query start and goal configurations c_s and c_g by finding a collision-free connection between each of c_s and c_g and nodes in the graph, and then finding a path through the graph between the two connector nodes. Alternatively, the graph may fail to contain nodes that can be connected to the query nodes.

2.1.2 Multiple-query roadmaps

In completely static environments, a roadmap can be constructed during an offline phase, and then saved for online queries that must be answered quickly. Since the roadmap is built prior to the planning queries, time can be spent to assure that the graph is a good approximation of the free space so that query nodes are likely to

connect to the graph easily. Early roadmap techniques were deterministic, and are briefly reviewed in 2.1.4, but it was found that the general motion planning problem is intractable for many degrees of freedom (Canny, 1988; Reif & Sharir, 1994), and randomized techniques were developed.

These probabilistic techniques tend to solve easy problems very quickly, and are therefore more practical than previous approaches which require planning time exponential in the number of degrees of freedom, even given problems that permit many possible solutions, a property that should make solutions easy to find. One early idea was the probabilistic roadmap. In the initial probabilistic roadmap work, the roadmap was built by choosing a random free configuration and then trying to connect it to the closest nodes already in the graph (Kavraki, Svestka, Latombe, & Overmars, 1996).

Refinements of the probabilistic roadmap approach worked to try to keep the roadmap small while maintaining good coverage of the free space. The intuition behind these modifications is that there may be parts of the configuration space that are both important for connecting the free space, and unlikely to be sampled randomly. If the free space were sampled uniformly with density high enough to ensure coverage of these tricky areas, the number of nodes in the graph might grow large enough to make searching the graph prohibitively slow.

More precisely, any place in the free space that is highly constrained in all directions (in configuration space), sometimes called a narrow passage, has these properties. Several techniques were developed to address this issue, such as visibility-based techniques (Siméon, Laumond, & Nissoux, 2000), bridge sampling (Hsu, Jiang, Reif, & Sun, 2003) and free-space dilation (Hsu, Sánchez-Ante, l. Cheng, & Latombe, 2006). We will discuss these in more detail in section 2.2.3. While these sampling techniques were originally intended to speed up the construction and search time for multiple-query roadmaps, several methods have become competitive with techniques that were designed to be used for single queries.

2.1.3 Single-query planners

Multiple query roadmaps are required to build as complete a representation of the connected free space as possible, since they are built with no information about the particular start and goal queries that the roadmap will face; since the roadmap is built before the time-critical query period, this is a reasonable goal. However, this approach relies on the assumption that the robot's environment remains static between planning queries, which is an unreasonable assumption in many applications. Single-query approaches, on the other hand, do not assume that the environment is static, but instead build a new roadmap for each individual query, using the information about the actual start and goal configuration to guide the exploration of the free space. If there are large regions of the free space which need not be represented at all to answer a particular query, these techniques try to avoid including them in the roadmap.

An early tool for this type of query was the potential field approach (Khatib, 1986; Barraquand, Langois, & Latombe, 1992), in which the robot is pushed by the potential field toward the goal and away from obstacles, and related local control approaches (Faverjon & Tournassoud, 1987a). Control with potential fields alone is extremely susceptible to local optima, but a randomized approach can be used to probabilistically recover from such problematic regions. The potential field can also be used as an alternate sampling technique in the general framework of multiple-query probabilistic roadmaps (Barraquand, Kavraki, Latombe, Li, Motwani, & Raghavan, 1997). Potential fields as a global planning technique are quite sensitive to parameter changes, and so are more frequently used as a local controller that is used by an algorithm with a more global vision(Faverjon & Tournassoud, 1987b).

The Ariadne's Clew algorithm combines a search strategy, which focuses on finding the goal configuration, with an exploration strategy that builds a representation of the free space local to particular landmarks (Mazer, Ahuactzin, Talbi, & Bessiere, 1998). Mazer et al. found that they could significantly improve performance by using obstacle surface information to guide the sampling, which made the algorithm more likely to produce sample trajectories that pass through narrow corridors. Another

early single-query approach was based on the multiple-query probabilistic roadmap, with the primary modification being that collision checking is done lazily (Bohlin & Kavraki, 2000).

Two techniques which have proven very successful on a wide range of problems is the single-query bi-directional lazy (SBL) planner (Sanchez & Latombe, 2001) and the rapidly-exploring random tree (RRT) (LaValle, 1998). Both of these techniques still use a roadmap representation to approximate the free space, and they build the roadmap probabilistically, but rather than building the roadmap to explore the space optimally for multiple queries, the goal is to connect the start and goal queries as quickly as possible. Each method uses a pair of tree-shaped roadmaps, one rooted at each of the start and goal query configurations.

In the SBL planner, developed by Sanchez and Latombe, the tree is grown by sampling near an existing node in the tree. The samples are generated close to one another and assumed to be collision free until a full proposal path between the start and goal configurations has been generated. At that time, the path is checked for collision, and colliding segment, if found, are removed from the tree(s). This process continues until a proposal path is found to be collision free. Our work uses the SBL planner both as a benchmark and as a component of our algorithm, described in chapter 4, so this planner will be discussed in more detail there.

The rapidly-exploring random tree (RRT) was first proposed by Lavalle as a new way to generate a multiple-query roadmap (LaValle, 1998), but then used as the key piece of a successful single-query planner (Kuffner & Lavalle, 2000). The idea behind the tree structure is that it is grown by sampling a random configuration and then taking a step from the closest node in the tree toward the random configuration. This allows the tree to explore the space more rapidly than it would by taking small random steps from existing nodes. However, it also avoids generating many samples that are far from the existing tree and therefore likely to be impossible to connect. The RRT-connect single-query planning algorithm involves again growing a tree from each of the start and goal nodes, and then trying to connect the trees to one another.

RRTs have been used successfully in several different systems, including a system

for controlling a humanoid robot (Asfour, Azad, Vahrenkamp, Regenstein, Bierbaum, Welke, Scröeder, & Dillman, 2008) and a planner for completely deformable environments (Rodriguez, Lien, & Amato, 2005). A comparison of the RRT algorithm and the A* search algorithm for planning on self-reconfigurable robots find that in practice the RRT is much faster (Brandt, 2006). Several variants of the RRT planner have been shown to be resolution complete (meaning that the planner is guaranteed to find a solution that falls within a specified tolerance) (Cheng & Lavalle, 2002), and an RRT variant has been used a subcomponent in a multiple-query planner (Bekris, Chen, Ladd, Plaku, & Kavraki, 2003). A number of systems have been developed to address specific planning difficulties by modifying the RRT-connect algorithm, several of which we discuss in section 2.2.3.

Other single-query planners have also been developed. The main distinction of each one is the sampling technique(s) used to generate the nodes in the roadmap. For example, two systems use entropy, or information gain, to determine the most informative places to sample (Burns & Brock, 2005b; Rodriguez, Thomas, Pearce, & Amato, 2006). Hsu introduced the notion of expansive spaces, and developed a two-tree sampling method that he proves will, with high probability, produce a good approximation of the free space, given assumptions about the expansiveness of the space (Hsu, Latombe, & Motwani, 1999). However, his work points out that when a space is *not* expansive, it tends to produce difficult planning problems that are not solved quickly by probabilistic techniques. As we will cover in detail in section 2.2.3, several different sampling techniques have been developed for specifically addressing the problem of non-expansive spaces, or spaces containing narrow passages. Thomas et al. propose a technique that chains samplers together to produces higher quality samples than the individual samplers would produce alone (Thomas, Morales, Tang, & Amato, 2007). Machine learning techniques have also been employed to choose between sampling strategies; we will see these in detail in section 2.3.

Siméon et al. have extended the notion of roadmap planner to include manipulation planning, which includes interactions with the objects to be manipulated (Siméon, Laumond, Cortés, & Sahbani, 2004). This work builds roadmaps

in extended configuration spaces, sometimes including parameters describing the interaction of the robot and manipulation object. These roadmaps are used to generate plans for moving the robot toward a grasp position, grasping the object, moving it to a new location (re-grasping if necessary), and releasing the object at some target pose. This system shows the flexibility of the roadmap technique, in that it can applied to extended configuration spaces to solve a wider range of problems than collision-free motion planning. However, this technique still requires more time than would be feasible in a changing environment. Nonetheless, such an algorithm could be used to provide training data to a general motion learning algorithm that could then be used to generate plans in real time.

2.1.4 Deterministic approximations

Most of the work described so far has been probabilistic. However, there has also been work done on deterministic approximations of free space. In fact, Lavalle et al. argue that the apparent advantage of sampling-based techniques is in fact not evident in the general case, but only in environments designed to showcase the strengths of particular sampling strategies (Lavalle, 2004).

Early techniques for building roadmap representations of the configuration-space free space were deterministic, and built roadmaps to cover the space completely. Cell decomposition techniques divided the free space into cells, and then placed roadmap nodes along the boundaries between the cells (Schwartz & Sharir, 1983). Generalized Voronoi diagrams were used to generate roadmap nodes along the medial axes of the free space, meaning that they are maximally far from obstacles (Ó'Dúnlaing & Yap, 1985).

A number of early approaches represented the free space by computing the configuration-space obstacles, either exactly or approximately. An algorithm by Lozano-Pérez computes a conservative estimate of the configuration-space obstacles by taking the union of recursively computed lower-dimensional slices of the obstacles (Lozano-Pérez, 1987). Other techniques determine equations for the configuration-space obstacle boundaries (Donald, 1984; Ge & McCarthy, 1989). Boundary equations have been

computed for a number of robot platform types, but they are in general difficult to compute; more importantly, computing the intersections necessary to use the equations to do planning is very complicated. Algorithms for computing configuration-space obstacles use mechanisms such as growing the obstacle according the robot's configuration (Lozano-Pérez & Wesley, 1979), computing swept volume of robot paths, using Jacobian-based information (Paden, Mess, & Fisher, 1989), and others. These techniques were found to be too expensive, and more approximate techniques were developed.

In work by Lindemann and Lavalle (Lindemann & Lavalle, 2004), the rapid exploration property of the RRT is found to be caused by the Voronoi bias of the sampling technique: nodes with larger Voronoi regions are more likely to be expanded. They propose a similar algorithm based on explicit computation of the Voronoi diagram of the tree. Note that this use of the Voronoi diagram (which is constructed with respect to the search tree) is different from the Voronoi diagram used to compute the medial axis of the free space (which is constructed over the free space).

2.2 Related problem areas

In this section, we review motion planning work that relates specifically to the problems we address with our work. Since the primary motivating forces for speeding up planning have to do with dynamic environments and sensor and actuator uncertainty, we first review approaches that have been proposed within the motion planning framework for dealing explicitly with these issues. Then we review techniques intended to focus on the "narrow passage" problem, as this is has been shown to be the primary cause of difficulty in motion planning problems.

2.2.1 Modeling dynamic obstacles

One way to address dynamic obstacles is to generate a complete plan given a particular configuration of obstacles, and then adapt the plan as the obstacles move **??**. Potential fields, or another local approach can be used to adapt the path locally to the changes

in obstacle location. New queries still require a complete plan as a starting place, which may be out of date by the time it is generated.

More recently, another class of flexible roadmap techniques have been proposed which combine the flexible path idea with the roadmap. These roadmaps allow the nodes and edges of the roadmap to move dynamically in response to changes in the environment (Yang & Brock, 2006; Gayle, Sud, Lin, & Minocha, 2007; Sud, Gayle, Guy, Anderson, Lin, & Manocha, 2007). These approaches rely on fairly powerful local controllers that are able to move the nodes and edges of the graph to accommodate changes in obstacle position, without disrupting the global constraints that allow the graph to be useful. Such controllers exist for mobile robots, or even groups of mobile robots, where the relationship between the configuration space of the robot and the obstacle space is fairly simple, but in the more general case, this can be a prohibitively difficult prolem.

Several previous authors have suggested the use of a structure, sometimes called a dynamic roadmap, that is intended to bridge the gap between single- and multiple-query planners (McLean & Laugier, 1996; Leven & Hutchinson, 2002; Kallmann & Matarić, 2004). This structure is essentially the same as our task distribution roadmap (described in chapter 3), though the way we construct it is different from previous work. It is similar to a multiple-query roadmap, with the addition of edge labels that indicate the volume that would be swept out by the robot as it follows the path represented by the edge. This allows for fast collision-checking along the edges of the roadmap so that edges can be invalidated quickly as obstacles move through the environment. Like a multiple-query roadmap, it avoids rebuilding the roadmap from scratch, but it is not limited to completely static environments. Our work with this data structure focuses on a new way of constructing the roadmap in order to take advantage of environmental regularities to avoid growing a roadmap that is exponentially large in the number of the robot's degrees of freedom.

Jaillet and Siméon work with a structure that is very similar to a dynamic roadmap, except that edges are labeled based on the position of dynamic obstacles that interfere with the path corresponding to the edge ((2004)). New obstacle

placements are compared to previously colliding ones and if they are the same, the edge is considered blocked. This requires that all the possible colliding positions of each obstacle be separately considered before the edge is completely labeled. Computing the swept volume directly gets at the same information, but in an obstacle independent way. If the number of possible positions for the obstacles is small, then it is slower to check the entire swept volume than to compare a few obstacle locations.

A common technique for dealing with dynamic obstacles is to incoporate time into the configuration-space roadmap (?). Alternatively, planning can be done with the obstacles represented in the robot's velocity space (Fiorini & Shiller, 1998). These techniques typically require that the trajectory of the obstacles is known or can be predicted. One time-extended roadmap technique by van den Berg and Overmars (2004) was proven effective in real-world applications in which the dynamic obstacles move cyclicly or are controlled by a single planner (as in multiple robot planning problems). If the planner is able to find a solution very quickly, then the measured velocities of the obstacles can be used to predict their immediate trajectories, and then continuous replanning can be used to generate the robot's motion (Hsu, Kindel, Latombe, & Rock, 2002). This is currently feasible only in low-dimensional configuration spaces.

Several techniques have been developed to take advantage of prior partial information about how the dynamic obstacles will move, without requiring that their full trajectories be known in advance. For example, techniques can leverage knowledge about a predetermined set of obstacle placements (LaValle & Sharma, 1997; Nieuwenhuisen, van den Berg, & Overmars, 2007). In a specific scenario involving two manipulators and a conveyor belt carrying parts to be manipulated, the problem can be decomposed into several lower-dimensional problems, each of which can be solved quickly (Li & Latombe, 1997). Again, while such information is sometimes available, there are many examples in which we could have no *a priori* information about the possible or likely placements of the obstacles.

If the only information that is known about the obstacle motion is the maximum speed with which they may move, one approach is to generate plans that are guar-

anteed to be collision free regardless of how the obstacles move (van den Berg & Overmars, 2006). It may be that no such path exists, however. Another possibility is to generate plans that are guaranteed to have sufficient clearance from the moving obstacles that the robot could avoid collision by simply stopping until it is safe to move (Alami, Siméon, & Krishna, 2002). However, these technique are applied only to mobile robots, and it is not clear how to extend them to the more general case.

Li and Shie (2002) operate in environments similar to those we consider, and also leverage the results from previous planning experience. Their approach involves saving the rapidly-exploring random trees (RRTs) from single-query instances in a "forest", and then invalidating nodes according to obstacle position before using the forest to compute a path with the new obstacle locations. The technique is similar in some ways to the DRM, but there is no attempt to make collision-checking for the forest fast, so it is unclear whether it will scale up well with increasing degrees of freedom.

2.2.2 Uncertainty

Another motivation for fast planning is that while motion planning techniques often assume perfect knowledge of the world, real robot systems generally involve uncertainty in the positions of the robot and the obstacles, and often the geometry of the obstacles as well, due to errors in control, sensing, or models used for planning. One way to address this, and the one we have chosen, is continuous replanning. This approach requires that the planning loop be fast enough that a plan can be made in real time with the best information currently available, to be replaced quickly with a new plan when new information is learned that may invalidate the old plan. The advantage of continuous replanning is that the uncertainty need not be explicitly represented or reasoned about; considering all possible outcomes often generates a paralyzingly large number of possible plans in worlds with substantial uncertainty, and acting according to the best plan for the most likely state of the world is successful in many settings. However, representing uncertainty explicitly allows the agent to take actions that are based on that uncertainty, such as information gathering ac-

tions, which may be necessary in certain types of problems. In this section we review the techniques that have been developed in the area of motion planning for dealing explicitly with uncertainty.

Early planning work addressed uncertainty through the use of preimages (Lozano-Pérez, Mason, & Taylor, 1984; Latombe, 1988). The preimage of a region of the configuration space, $\mathcal{R}$ can be informally described as all the regions of the configuration space from which some controller can be guaranteed to reach $\mathcal{R}$, given a model of the uncertainty in the system. Given the ability to "backchain", or produce a region's preimage, a plan can be generated in response to a start/goal query by backchaining from the query's goal until a preimage is produced that contains the start.

Subsequent work found that these techniques, which thoroughly investigate the possibilities raised by the system's uncertainty, are in general trying to solve an intractable problem (Canny, 1989). This indicates that some kind of compromise must be made if a control strategy is to be computed in reasonable time: we must make and exploit assumptions about the robotic platform or the sources of uncertainty in the system, possibly by introducing modifications to the environment, or relax our requirements for soundness and/or completeness. One proposal along these lines involves creating "islands of perfection", or regions of the state space that can be sensed exactly (Lazanas & Latombe, 1995). This compromise allows for successful planning in low degree-of-freedom mobile robots, but it does not seem to scale up to more complicated high degree of freedom manipulators.

A more recent approach frames the manipulation problem as an instance of a partially observable Markov decision process (POMDP) (Hsiao, Kaelbling, & Lozano-Pérez, 2007). An overview of the POMDP framework is outside the scope of this review, but the benefit beyond the approaches described so far is that solving a POMDP provides an control strategy that is optimal with respect to some reward function, taking the uncertainty in its observations into account. For example, if the agent could take a few non-goal-directed information-gathering actions, and then reach the goal quickly given its newfound clarity, it will choose this policy over one that always moves towards the goal, but requires more, smaller steps due to uncertainty. How-

ever, the generality of the approach again comes at a cost; POMDPS are, in general, very difficult to solve. Approximate approaches have significantly increased the size of problem that can be solved in reasonable time, but high degree-of-freedom platforms in complicated environments are still beyond their reach. Note, too, that the work done by Hsiao et al. addressed manipulation problems, rather than a collision free motion planning problem; it is reasonable to expect that uncertainty will need to be modeled fairly carefully when the robot is required to be in close contact with the objects in the world. We believe that collision-free planning is less likely to demand the same degree of precision in reasoning about uncertainty, and therefore lends itself more readily to continuous replanning.

Rather than trying to solve an entire POMDP, Chakravorty and Saha instead treat the motion planning problem as an instance of a fully observed Markov decision process (MDP), meaning that all of the uncertainty is captured in the state transitions, but the state can be sensed accurately by the agent (Chakravorty & Saha, 2007). The technique uses a hierarchy that does high-level planning via dynamic programming and low-level planning with an RRT, and achieves better performance than either of these technique alone. However, the test platform is a low degree-of-freedom mobile robot, and it is less clear how to build such a hierarchy in the general case.

Another recent approach adds uncertainty information to the traditional motion planning roadmap (Burns & Brock, 2007). Each edge in the roadmap has an associated utility as well as a cost which is computed based on the estimated likelihood of failure. Then the A* search algorithm is used to find minimum cost paths through the graph. This technique allows for paths to be found when more traditional probabilistic roadmap approaches would fail due to the noise in the world. However, the additional computation that must be done to find paths that will succeed with high probability makes the approach too slow for high degree-of-freedom robots in dynamic environments.

An enormous amount of research has been done in the area of simultaneous localization and mapping, which is intended to use noisy sensor data to build a map of the robot's surroundings, while also probabilistically situating the robot in the

map (Durrant-Whyte & Bailey, 2006; Bailey & Durrant-Whyte, 2006). This work has been very successful in the realm of mobile robots, allowing them to navigate in noisy, dynamic environments without becoming confused about their location. However, solving the SLAM problem only goes part of the way toward solving the general motion planning problem: it tells the robot where it is, but nothing about how to behave. In the case of mobile robots, the policy is often (though not always) reasonably easy to determine, given an accurate estimate of the robot's pose. This problem becomes much trickier in the case of manipulators, which have a more complicated mapping from workspace to configuration space. Also, as the number of degrees of freedom go up, so do the number of samples needed to have a good estimate of the probability distribution over the robot's state. In fact, if one assumes that the density of samples is what determines the accuracy of the distribution estimate, the number of required samples goes up exponentially in the number of the robot's degrees of freedom.

2.2.3 Narrow passage sampling

It has been argued quite persuasively that the problem of narrow passages in configuration space is the source of difficulty in the most time-consuming planning problems(Hsu et al., 1999). In response to this, there have been a number of techniques proposed for increasing the proportion of samples in the constrained regions of the free space, thereby allowing the graph to remain small but to have high likelihood of containing the necessary connections. This issue exists for both multiple- and single-query planners, and so many of these sampling techniques are relevant in both realms. In some cases the expense of the sampling technique may make it less feasible for single-query settings, since the cost is paid for each query, so the technique may cost more time than it saves. Nonetheless, the general properties desired in a sampling strategy for combating the narrow passage problem are largely shared between the two scenarios. These strategies can be divided into two broad categories: filtering, or rejection sampling methods, in which samples are generated and then discarded, and retraction methods, in which samples are modified.

Rejection sampling

The idea behind rejection sampling in this context is to generate samples normally, but to be selective about which samples are actually used in the roadmap, thereby increasing the proportion of samples with desiable properties. This allows the roadmap to have many samples where they are actually necessary and few redundant samples. However, the time required to generate the samples may be very large, as many samples may need to be rejected for each useful sample.

The notion of visibility can be used to determine whether or not a sample should be included in the roadmap (Siméon et al., 2000). The visibility domain of a configuration is made up of all of the other free configurations to which a collision-free connection exists. Visibility-based techniques ensure that the entire configuration space is visible to at least one node in the graph, while keeping the total number of nodes small, making it as fast as possible to search.

Work by Yershova et al. also uses the notion of visibility to bias sampling, but in a different way (Yershova, Jaillet, Siméon, & LaValle, 2005). Their approach modifies the RRT algorithm so that nodes that are near obstacle boundaries are allowed to expand only to nearby configurations, rather than configurations chosen uniformly at random. In this way, they avoid wasting time trying to expand parts of the tree that are very likely to be in collision, and instead focus on expanding nodes that can expand into free space, which increases the likelihood of sampling in narrow passages.

There are several techniques for narrow passage sampling motivated by the intuition that samples near obstacles are more likely to be in narrow passages. The Gaussian sampling strategy is designed to sample near obstacles by first finding a sample that is in collision, and then sampling some distance d away from the colliding sample, where d is normally distributed (Boor, Overmars, & van der Stappen, 1999). A similarly motivated technique, known as the bridge test, generates samples until two colliding samples are found, with a non-colliding sample in between them (Hsu et al., 2003). In both cases, the non-colliding sample is added to the roadmap.

Another technique uses the notion of *manipulability* to determine which samples

are kept (Leven & Hutchinson, 2003). Manipulability is derived from the end effector Jacobian, and is a measure of the robot's ability to move the end effector from a given configuration. The technique rejects samples that have higher manipulability, motivated by the idea that many low-manipulability configurations are in constrained parts of the configuration space.

Other techniques use features of the workspace (rather than configuration space) to help determine where to sample. Work by van den Berg and Overmars uses a cell decomposition of the workspace to estimate the likelihood that each region in the workspace is part of a narrow passage, and uses that information to choose the positional part of a mobile robot's configuration, while sampling the orientation uniformly (van den Berg & Overmars, 2005). A related approach by Kurniawati and Hsu uses a triangulation of the workspace to bias the sampling of part of the configuration (Kurniawati & Hsu, 2004). In both cases, the technique is applied only to mobile robots and free-flying rigid objects, where the decomposition of the position and the orientation of the robot is easy. It is less clear how these techniques could be applied to complex manipulators.

Sample retraction

The primary motivation behind retraction techniques is that sampling near the medial axis of the free space leads to more narrow passages samples. This is a similar motivation to the approximate free space representations that use the Voronoi decomposition of the free space to choose milestones (Ó'Dúnlaing & Yap, 1985). Wilmarth, Amato, and Stiller (1999) demonstrate the connection between medial axis sampling and narrow passage sampling, and then propose an algorithm that can retract free-space samples, as well as some colliding samples, to the medial axis. However, this technique relies on the ability to compute, for a given configuration which may or may not be in collision, the distance to the closest point that is on the boundary between colliding and non-colliding configurations. These clearance and penetration depth computations are not, in general, tractable. An extension of this work proposes approximations for each of these operations (Lien, Thomas, & Amato, 2003).

Saha and Latombe suggest that configurations that are barely colliding are both easily and fruitfully retracted toward the medial axis (Saha & Latombe, 2005). Thus, they generate roadmaps that are likely to contain such configurations by shrinking the models of either the robot, the obstacles, or both. Once this roadmap is constructed, the colliding configurations can be retracted by just a small amount. Hsu et al. provide a more efficient algorithm for doing this free space dilation (Hsu et al., 2006).

Another retraction method involves the medial axis not of the configuration-space free space, but the workspace free space, which is generally lower dimensional (Holleman & Kavraki, 2003). The idea is to choose some points on the robot (handle points) and try to find configurations in which those points are close to the medial axis of the workspace. This method requires the ability to adjust configurations to move the handle points closer to the workspace medial axis, not an easy problem in general.

2.3 Machine learning

There have been several contributions to the motion planning literature that involve learning in some way. In this section we review the various ways that machine learning has been applied to the task of planning robot motion. We first review techniques that apply machine learning within the context of traditional motion planning techniques like those we have seen so far, and then we discuss learning techniques for generating policies rather than complete motion plans.

2.3.1 Remembering plans

In chapter 3, we describe a simple learning technique that remembers previous solutions to help speed up subsequent queries. Chen proposed an approach that uses a slow planner to to generate plans that can then be adapted by a fast planner when similar problems are encountered again (Chen, 1997). Similar work by Caselli and Reggiani proposes remembering solutions to previous planning problem in a roadmap (Caselli & Reggiani, 2000). However, unlike our work, this work is done in a static environment, and therefore does not need to address the need for general-

ization of old plans to new environments.

Other approaches involve re-using previous planning experience by repairing an RRT roadmap (Ferguson, Kalra, & Stentz, 2006) or a forest of RRTs (Li & Shie, 2002) in real time as the environment changes. These approaches show considerable improvement over the standard RRT in the case of mobile robots. Ferguson et al.'s system is even able to plan effectively for a twenty-degree-of-freedom system of ten mobile robots. However, even with a large number of degrees of freedom, the configuration space of mobile robots has simple relationships to the workspace that do not hold for complicated manipulators, so we believe this technique is likely to be less effective in a manipulator domain.

The planning literature in artificial intelligence includes work on recalling and combining previous plans, in particular (Haigh & Veloso, 1995; Haigh, Shewchuk, & Veloso, 1997) deals with routes through a city map. This work addresses similar issues to the ones addressed in this thesis but the different constraints in the domains lead to solutions that are considerably different.

2.3.2 Biasing sampling

Given the wide variety of sampling methods that have been proposed for generating probabilistic roadmaps, another active research area is centered on determining, based on experience, which sampling method is most appropriate for the current environment and planning problem. Work by Morales et al. computes features of regions of the workspace that are used as input to a classifier. The classifier then learns to predict an appropriate planner type for the particular region (Morales, Tapia, Pearce, Rodriguez, & Amato, 2004). Hsu et al. propose a technique in which the sampling distribution is a weighted sum of several component distributions, and the weights are adapted based on their effectiveness (Hsu, Sánchez-Ante, & Sun, 2005). Another adaptive technique by Kurniawati and Hsu uses both workspace information and sampling history to determine the sampling distribution.

Burns and Brock propose a method that builds a probabilistic model of the free space and then uses that model to bias sampling (Burns & Brock, 2005a). Learned

functions are used to predict whether or not a configuration is in collision, and whether or not the path between two configurations is in collision. Locally weighted regression (LWR) is used for both of these learning problems (Atkeson, Moore, & Schaal, 1997). The input to the learned function is either a configuration or a path between two configurations, and the output in both cases is a prediction, along with a variance on the prediction, for whether the input is in collision. The model is then used to bias sampling by choosing samples that decrease the model's variance. Related work by the same authors uses an estimate of the collision probability of a configuration to compute the expected utility of the sample, and bias sampling in favor of high utility configurations (Burns & Brock, 2005c).

Work by Kalisiak and van de Panne uses a classifier to learn whether or not planning problems are possible (Kalisiak & van de Panne, 2007). The input to the classifier is sensor input from rangefinder sensors, the output is whether or nor the current planning problem is possible in the environment represented by the sensor readings. This function is learned by a support vector machine (SVM) with a radial basis function kernel. This viability oracle is then used to guide sampling away from configurations that are perhaps not in collision, but believed to be impossible. Another approach by Hauser et al. also learns to predict the likelihood that an edge in the roadmap will be collision free, though in the context of manipulation planning for a climbing robot (Hauser, Bretl, & Latombe, 2005). The output from the learning is then used both to bias both the sampling and the roadmap search.

2.3.3 Learning control policies

Reinforcement-learning methods (Sutton & Barto, 1998) have been used for learning a policy, which maps from the robot's current pose to an incremental action, such as joint velocities. The desired behavior of the robot is then expressed as a reward function, based on the state and action. The learning problem is then to find a policy which optimizes the expected (possibly discounted) reward.

Reinforcement learning faces two particular challenges in addressing robot motion planning: the state and action spaces are both continuous and high dimensional.

Many reinforcement learning algorithms learn a value function over the states and actions, and then choose the action by finding the one with the maximal value; both parts of this are more difficult in high-dimensional continuous spaces. In continuous, but low-dimensional spaces, one possible approach is to simply discretize the action space and use a tight control loop. Using this approach, Randlov and Alstrom (1998) produced an algorithm that successfully learned to ride a bicycle. Also using discretized reinforcement learning, Nikovski and Nourbakhsh (2002) were able to learn a policy for balancing an inverted pendulum based on visual feedback.

Other approaches deal with the continuous spaces directly. Smart and Kaelbling (2000) propose estimating the value function using locally weighted regression (Atkeson et al., 1997) to approximate the value function, and an iterative algorithm to find the best action. (Peters & Schaal, 2007) have recently proposed a reinforcement learning technique specifically designed to control manipulator-type robots. This work uses expectation maximization to find the parameters that maximize the expected reward of a stochastic policy.

These policy learning techniques typically assume a static environment, or only a small amount of variability in the obstacles (e.g. doors that might be open or closed). The emphasis is on having an optimal (or near-optimal) control reaction for every input state. In the presence of obstacles, however, it may be less important to have optimal control than to have control that explicitly plans for obstacle locations. Also, when reinforcement learning is used to produce a controller directly, the desired behavior of the system is built into the policy via the reward function, so it is generally not easy to transfer the learned controller to perform a different task. As we will see in the next section, reinforcement learning can also be applied to the problem of learning inverse kinematics, which is one way to address the issue of multiple desired behaviors.

Outside of the reinforcement learning framework, there has been work done on the general problem of learning controllers. Recent work by Grollman and Jenkins (2008) compares the use of two different regression techniques, locally-weighted projection regression and sparse online Gaussian process regression, on a control policy

learning problem on a robotic dog platform. They report success on several problems where a controller can be used to generate training data for the learner. However, for complicated tasks they use human demonstration to generate the training data, which creates problems for the function approximators. This is because the humans may choose to solve the task different in different instances, generating inconsistent training data. We will address this particular problem in more detail in chapter 4.

In the context of animation, Cooper, Hertzmann, and Popović (2007) use data generated by humans doing a particular motion task while wearing a motion capture system, and use this data to learn a control policy for an animated humanoid. This work uses active learning, building a controller based on the data so far, and then querying the human demonstrator for help on tasks at which the controller performs poorly.

2.3.4 Learning inverse kinematics

Learning can also be applied to the problem of inverse kinematics: a transformation from a Cartesian-space description of a desired pose to a set of robot parameters that achieve the pose. A closely related problem is the mapping from Cartesian motions of a point on the robot (often the endeffector) to robot parameter motions. It is sometimes useful to represent the mapping from some transformation of the Cartesian position, such as the apparent position of the endeffector in an image of the robot. In each case, knowing this relationship allows for planning to be done in the workspace (or image space), which is lower dimensional than the robot's configuration space.

When the robot has the same number of degrees of freedom as the Cartesian constraints on the motion, the inverse of the Jacobian matrix can be used to determine the required joint motions. However, when the robot has more degrees of freedom than it needs, the Jacobian is singular. This is a substantial challenge for learning inverse kinematics, and one that we will discuss in our work as well, is the need to resolve redundancies for overactuated robots.

There is a long history of models for learning inverse kinematics. Some approaches involve learning a mapping from action effects, or Cartesian motions of the endeffec-

tor, to joint motions. These techniques all share the weakness that they generally fail to converge when faced with redundant platforms, as two very different joint motions may lead to the same endeffector motion (?) A more robust approach, sometimes called resolved motion rate control (RMRC), learns a mapping from endeffector motion and current robot pose to joint motion (Whitney, 1969; Jordan & Rumelhart, 1992; D'Souza, Vijayakumar, & Schaal, 2001). This approach generates a unique inverse kinematics solution, but that limits the motion of the robot to the one learned solution, so it is unable to adapt to novel task constraints, such as obstacles.

Molina-Vilaplana et al. (2004) developed a system that uses a network of radial basis functions to learn the inverse kinematics for a redundant manipulator. They are able to use the learned inverse kinematics to perform simple reaching motions in obstacle free environments. In the SURE_REACH system proposed by Butz, Herbort, and Hoffmann (2007), a hierarchical system of networks is used to learn multiple inverse kinematics solutions. They then use a heuristic to adapt motions to obstacles at run time.

In work by Lopes and Santos-Victor (2006), the joints of the arm are categorized into either redundant and non-redundant for a particular desired motion of the endeffector (described, in this case, as the perceived motion of the endeffector in images of the arm). The redundant joints are then included in the input to the function, and only the non-redundant joints are predicted. Training data is generated by using gradient descent on a cost function based on the accuracy of the endeffector position and some desired criterion for the redundant degrees of freedom (such as energy minimization), and locally-weighted regression is used to approximate the function.

Reinforcement learning has also been applied to the inverse kinematics problem. As with policy learning, these techniques generally do not address the issue of obstacle avoidance. Planning a path for the endeffector that does not collide with obstacles is relatively easy, and learning the inverse kinematics allows the robot to turn that path into joint motions quickly, but it does not guarantee that the rest of the robot will not collide with the obstacles.

2.4 Summary

This review of previous work in the area of motion planning is far from a comprehensive survey of the field, thanks to the many years of fruitful research that has been done over the last thirty years. It is primarily focused on techniques that strive to address similar concerns as our work: generating collision-free motion plans in environments that require fast planning times, either because the objects in the environment are dynamic, or because their locations are partially or imperfectly sensed. Our work aims to bring machine learning techniques to bear on the problem of motion planning, in a way that is as yet unexplored.

Chapter 3

Motion plan recall

The most straight-forward way to learn from previous experience is simply to remember every successful solution to problems encountered in the past. Given this memory of problems and their solutions, solving a new problem involves finding similar problems stored in memory, and using a previous solution, or combination of previous solutions, to come up with a new, related answer. In general, one expects that this simple learning strategy may require an unreasonably large number of solutions to be stored in memory, making it too slow to find relevant solutions. In this chapter, we investigate ways to make this learning technique tractable for the problem of learning motion plans in regular environments.

Our overall method remembers previously computed solutions by maintaining a data structure, called a *task distribution roadmap* (TDR). The roadmap contains connected paths and their swept volumes. Given a new task instance, the system uses the swept volumes, along with the obstacle locations, to determine what portion of the roadmap is collision-free for this task instance. Ideally, that subset of the roadmap can then be used to quickly find a path, just as a roadmap is typically used in a static environment. In the general case, one expects that such a roadmap will have to be exponentially large, and therefore slow to search, to guarantee that it is sufficiently likely that a solution will be found in the collision-free subset of the roadmap. In this chapter we discuss the assumptions we believe can be reasonably made about some real-world environments, and how to leverage these assumptions to

build a TDR that can be used efficiently.

3.1 Environmental Regularities

If the environment in which the agent operates is truly arbitrary, in that obstacles can occupy any possible region of the space, and the agent may be required to plan motions between any possible configurations, remembering previous solutions exactly will not, on average, help speed up the planning process. We know that the general motion planning problem is PSPACE-hard (Canny (1988)), so without some restriction on the general case, we can not expect to improve matters simply by caching solutions.

However, more often than not, environments contain obstacles that are not completely randomly placed. For example, a given office environment typically contains desks and tables that do not move at all, and chairs that move reasonably freely, but typically stay in one plane (that is, not floating) and are often loosely constrained to a particular region (perhaps in front of a desk). Completely different environments often include similar components that one might hope to interact with in a consistent way, such as tables that have slightly different heights or doorways that are not identical but require a similar motion to go through. Lastly, in dynamic environments obstacles will likely change location smoothly over time. A plan that worked successfully a minute ago is likely to be useful again now.

Further, the tasks that the agent is required to solve may be similar from instance to instance. This is obviously true if the agent is simply performing a task repeatedly, but operating in a changing environment. More interestingly, if the obstacle placement in the environment has regularities, and the agent needs to manipulate the items in the environment, then the tasks will exhibit similar regularities. It is these environmental regularities, both in obstacles placement and in task specification, that we wish to leverage with the task distribution roadmap.

3.2 Using the Task Distribution Roadmap

As mentioned in chapter 2, the TDR data structure has been proposed in previous work (McLean & Laugier, 1996; Leven & Hutchinson, 2002; Kallmann & Matarić, 2004). But in each instance, the goal of the work is ultimately quite different from ours, which leads to a different strategy for constructing the roadmap. We believe that the potential leverage of a memory-based technique relies on discovering and exploiting environment regularities. Thus, our goal is to capture the common cases, and only the common cases, in order to strike an optimal balance between the time taken to search the TDR and the likelihood that a solution will be found in the roadmap. Their goal is to construct, off-line, a complete roadmap that is essentially exhaustive, in that it contains paths that are near every possible path that the robot can execute. Given this structure, they use it to do planning in essentially the same way we do.

3.2.1 Task Distribution Roadmap

Consider a robot with n degrees of freedom, operating in an m-dimensional workspace, $\mathcal{W}$. Let $\mathcal{C}$ be the n-dimensional configuration space of the robot when there are no obstacles present. A task distribution roadmap is a pair $\langle N, E \rangle$, where $N \subseteq \mathcal{C}$ is a set of nodes and E is a set of edges. Each edge, $\langle c_1, c_2, v(c_1, c_2) \rangle$ represents a simple (e.g. straight line) path between two configurations in N: c_1 and c_2. Each edge is annotated with its *swept volume*, $v(c_1, c_2)$, which is the subset of the workspace $\mathcal{W}$ that includes every point occupied by any part of the robot during its traversal of the simple path from c_1 to c_2. This can be represented using a grid in the workspace, and marking any grid cell that is touched during the traversal as being part of the swept volume. This representation is conservative, in that it overestimates the swept volume, but we can correct for the conservatism when it is used. Figure 3-1 shows an illustration of the task distribution roadmap structure for a snakelike robot. Note that a grid in the workspace is feasible, as it has only three dimensions, whereas discretizing the configurations space is, in general, not feasiable.

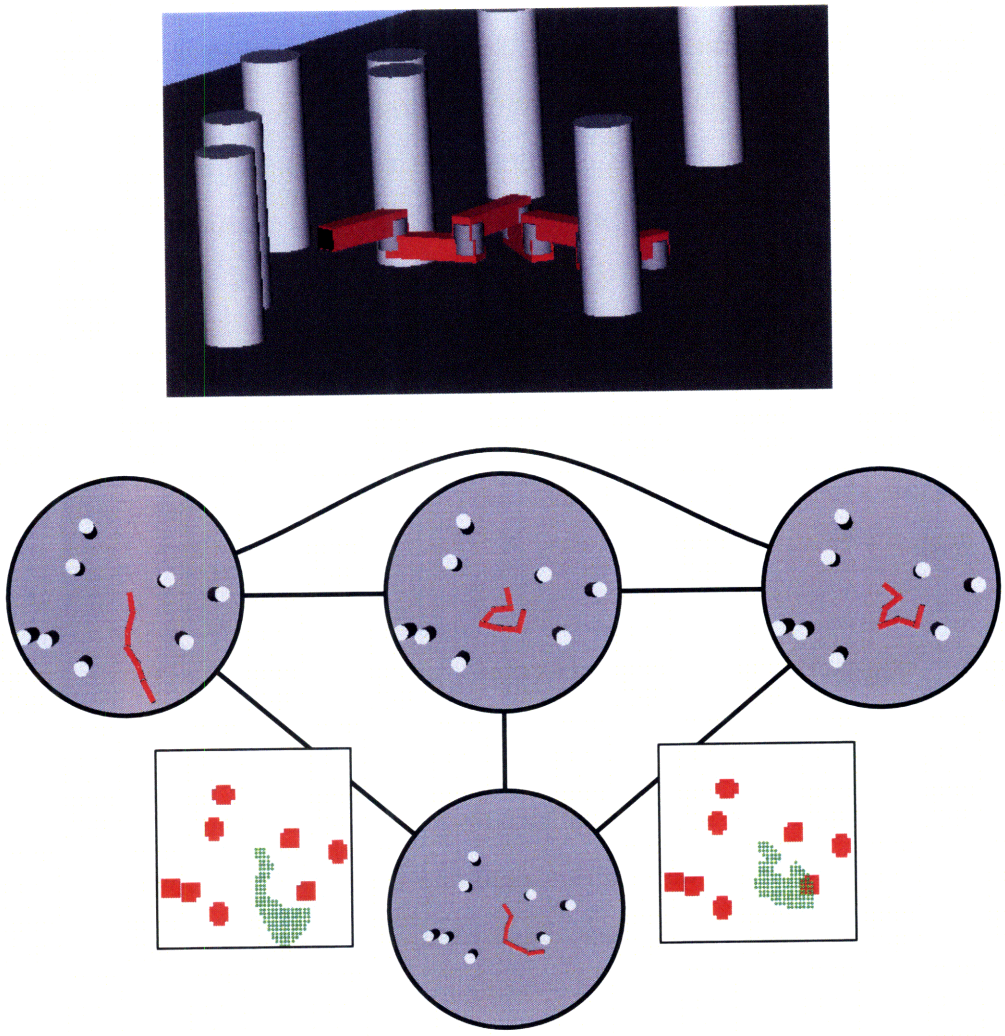


Figure 3-1: Top: A snake-like robot with six degrees of freedom that is anchored to the floor and surrounded by cylindrical obstacles. Bottom: An illustration of the task distribution roadmap structure. Nodes contain overhead views of the robot and obstacles. Two edges are labeled with the swept volumes of the robot as it moves between the configurations (in green) and the grid cells occupied by obstacles (in red). The left edge is collision-free for this configuration of obstacles, and right is not.

3.2.2 Querying the TDR

A planning task instance $t = \langle s, g, O \rangle$ consists of start and goal configurations, s and g in $\mathcal{C}$, and a set of obstacles O, which is a subset of $\mathcal{W}$. A task instance specification t can be given directly to the planner, which will eventually return a solution path in the form of a sequence of configurations $(s, c_1, \ldots, c_n, g)$ such that the simple path between each successive pair of configurations is collision-free in $\mathcal{C}_t$, the subset of $\mathcal{C}$ that is not in collision with any of the obstacles O in task instance t.

We can also attempt to solve a planning problem using the TDR R directly. We will begin by describing the most conceptually straightforward version of the algorithm, then discuss a variation that is much more computationally efficient.

1. Given a set of obstacles O_t, we need to temporarily remove any edge $\langle c_1, c_2, v \rangle \in R$ for which $v \cap O_t \neq \emptyset$, that is, for which the swept volume is overlapping with the obstacles. This test is done in the workspace and can be done efficiently with a grid-based representation of v and O_t. Let R_t be the TDR R with colliding edges and nodes removed. Note that the resulting R_t could be disconnected (in the sense that not every node is reachable from every other node).

2. The next step is to see whether s_t and g_t can be connected, by simple paths, to nodes in R_t. This is done by sorting the nodes in R_t according to distance (in configuration space) from s_t (and similarly for g_t). Starting with the closest node, each simple path between s_t and the node configuration is checked for collisions in O_t. This process ends when a collision-free simple path from s_t to a node in R_t is found. If such a path cannot be found for both s_t and g_t, the process fails, and the planner is called directly on the problem.

3. If nodes n_s connecting to s_t and n_g connecting to g_t are found, then we search R_t for a path between n_s and n_g; if such a path is found, it is returned as a solution. If there is no such path, then the planner is called.

4. If, on the other hand, only one endpoint can be connected to the graph with a simple path, then we take the node n in R that is closest to other endpoint

and invoke the planner to find a path from n to the other endpoint. The idea is that this may be a much simpler planning problem than finding a path all the way from the start to the goal. Again, if the planner fails at this problem, then we simply call the planner on the original task instance.

In fact, rather than computing the entire R_t up front, by invalidating all edges, we perform a process of lazy edge invalidation during the search to see whether n_s and n_g can be connected. During that search, any edge for which v does not overlap O_t is retained; but because the representation of v may be conservative, if it looks like the swept volumes might overlap, we do an actual collision check to verify whether that is the case.

There are a number of other alternative search strategies that might be reasonable, including finding all (or more than one) nodes that can be directly connected to s_t and g_t, and performing a search with multiple initial and goal states. We did not experiment with these in detail.

This algorithm is expected to return a solution in every case that the original planning algorithm would have. In the worst case, when the TDR cannot be used to generate even part of a solution to the problem, it will take longer than the planner would have, because of the added overhead in trying to use the TDR first. However, if the TDR contains all or most of a successful path, and the path is small, then it can be expected to run much more quickly.

3.3 Learning the Task Distribution Roadmap

Previous uses of a data structure that is similar to our TDR (McLean & Laugier, 1996; Leven & Hutchinson, 2002; Kallmann & Matarić, 2004) have been intended to apply to all possible task instances for a particular robot and so they have been built off-line, independently of any distribution over tasks. We build the roadmap as an on-line process while solving instances drawn from a particular task distribution.

3.3.1 Augmenting the TDR

Whenever the planner generates a new path, $(c_1, \ldots, c_n)$, we use it to augment the existing TDR, R. The most straightforward approach is to simply add each configuration on the path, c_i, to N, and add an edge $\langle c_i, c_{i+1}, v(c_i, c_{i+1})\rangle$ to E for each consecutive pair of configurations in the path. All by itself, this does not make the TDR too much more useful: it will contain the new path, and be able to use it in its entirety, but will not be able to use just a part of it very effectively. Instead, we want to connect the new path to R: we must find some collision-free segments between one configuration in the new path and a node in the existing TDR. For each configuration c_i in the new path, we add k additional edges, connecting c_i to other configurations in N. For each c_i, we consider the configurations c in N in order of their distance to c_i, and check to see whether the simple path from c_i to c is collision-free. If so, then an edge $\langle c_i, c, v(c_i, c)\rangle$ is added to E. This done until c_i has k edges, or all of the c have been exhausted.

3.3.2 Decreasing swept volume

Our goal in adding paths to the TDR is to try to increase the "coverage" of the TDR; that is, to make it so that future planning problems are more likely to be solvable within the TDR. So, in adding new nodes and edges, we would like to increase the coverage as much as possible. An edge $\langle c_1, c_2, v\rangle$ can be used in a new planning task instance with obstacles O_t whenever $O_t \cap v = \emptyset$; the smaller we can make v, the more likely it is that this condition will be true for a new task instance. So, the TDR can be improved by decreasing the swept volume of the edges that are added to it.

Ideally, one could call the planner with a criterion of producing paths with low swept volume. This is a very difficult optimization problem, however, and would make the planner completely computationally infeasible. Instead, we perform post-processing on the solution path received from the planner to reduce its swept volume. There are many possible strategies for doing this, including possibly adding new configurations to the plan, but in this paper, we take a very simple approach that

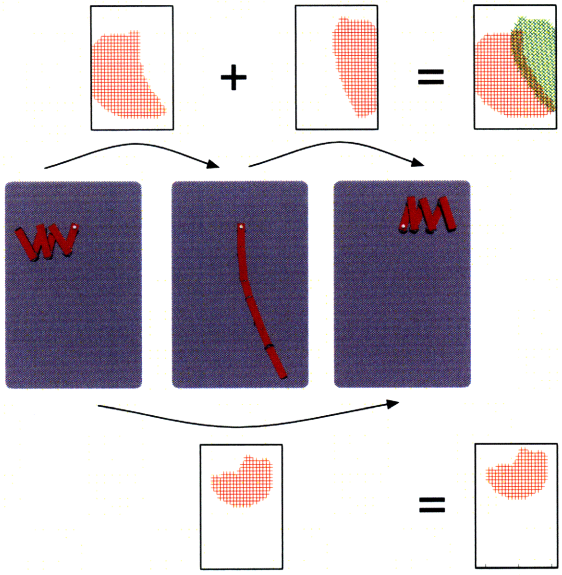


Figure 3-2: Example of decreasing the swept volume of a path by removing a node, in a problem for a robot with a fixed base and five revolute joints in a 2D workspace. The middle row of the figure shows three successive configurations of the robot. The top row shows the swept volume of each linear segment of the path followed by the union of the swept volumes. The bottom row shows the swept volume when the second configuration is omitted. We can see that the swept volume is smaller when the second configuration is removed.

only considers removing configurations from the path.

One simple greedy approach is to remove configurations from the path if doing so decreases the swept volume without compromising the path's validity. Figure 3-2 shows an example of a short path from which one node can be removed to substantially decrease the swept volume of the whole path. However, it is not always the case that removing a node decreases the swept volume. Figure 3-3 shows an example of a path for which removing a node would increase the swept volume. Not only must we examine the swept volume of the entire path every time we are considering removing a node, but there are an exponential number of path subsets that might have the smallest swept volume, and considering them all is clearly not feasible. If we consider removing only one node at a time, we may miss opportunities to remove multiple nodes at once, since removing them individually may not improve the swept volume. Figure 3-4 shows an example path for which this is true. So, a simple top-down

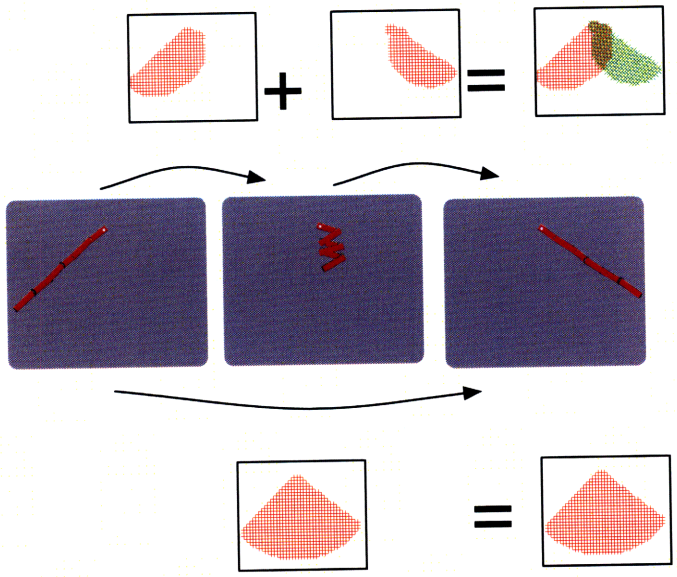


Figure 3-3: Removing the middle node from this path increases the swept volume. Just because removing a node does not cause a collision does not mean that it decreases the swept volume of the path.

strategy can easily fail to find the optimal swept volume. Similarly, if we consider only multiple nodes at once, we may also stray from optimal, as demonstrated by Figure 3-5. So, a simple bottom-up strategy will also typically fail to find the optimal swept volume.

Since looking at the full set of subsets is infeasible for paths longer than about ten configurations, we considered several approximations. By experimenting with each method on paths that were short enough to find near-optimal solutions, we determined empirically that a top-down approach, in which subsets of decreasing length are considered for removal, was a reasonable compromise on running time and amount of swept volume reduction. We also considered a bottom-up case in which individual nodes are first considered for removal, and then increasingly large pieces of the path are looked at. Another method was to choose two random configurations in the path and remove the section in between if it was found to decrease the volume. Figure 3-6 shows the results of our experiments comparing several methods of decreasing the swept volume.

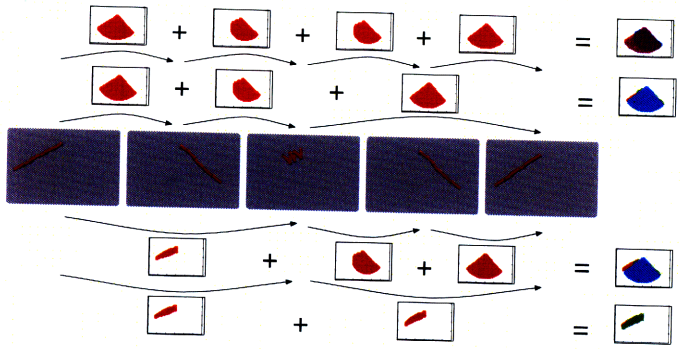


Figure 3-4: This path shows that removing only one node at a time may cause us to miss opportunities to remove multiple nodes. The swept volume for the whole path is shown at the top. Below that is the swept volume if the fourth configuration is skipped, which is slightly higher than leaving the configuration in. Similarly removing only the second node is worse than leaving it in. However, removing both, as shown in the bottom swept volume, gives by far the smallest swept volume.

3.3.3 Deciding when to augment the TDR

The problem of choosing an appropriate size for the TDR, in order to optimize average computation time, is an interesting one. As the size of the TDR increases, its coverage increases, and so the likelihood of finding the solution to a new problem within the TDR increases. At the same time, however, the time taken to search for a solution *within* the TDR increases. If we allow the TDR to grow without bound, the overall algorithm will become slower than calling the planner.

As we will discuss further in section 3.4.3, this is a serious issue in off-line dynamic roadmap (DRM) construction (as in (McLean & Laugier, 1996; Leven & Hutchinson, 2002; Kallmann & Matarić, 2004)), because the DRM needs to be prepared to accelerate any task instance that appears, which argues that it needs to have high coverage, but the associated size risks very slow search times within the roadmap. Because we build the TDR based on a distribution of task instances, we can let the task distribution guide which parts of the configuration space need more dense sampling.

The most naive approach to building the TDR would call the planner on some initial "training" set of task instances drawn from the task distribution and put all of the solutions into the TDR. This will result in a very large TDR with lots of solutions that are relatively redundant.

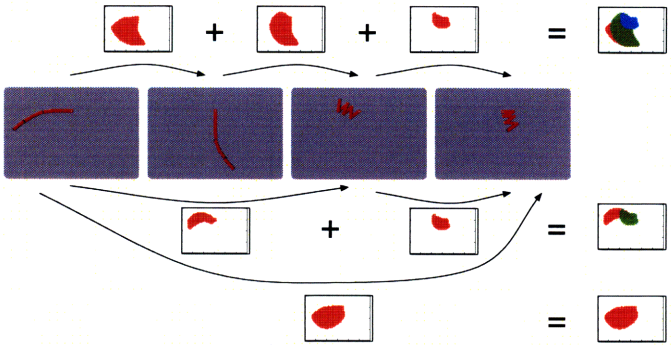


Figure 3-5: This path shows that we do not always want to remove the largest possible subset of nodes. Removing both the second and third configurations does decrease the swept volume (shown on the bottom), but not by nearly as much as removing just the second configuration (shown in the middle).

A more effective approach is to take the task instances sequentially and use a process similar to one frequently used in nearest-neighbor algorithms, in which a solution is only added if the current TDR was unable to solve the problem. This ensures that the structure isn't redundant and significantly decreases the size, and therefore the searching time, of the TDR.

There is a further benefit to this approach. In a domain for which there are many possible solutions to a given task instance or set of task instances (for example, due to redundancy in the robot), a non-sequential approach would tend to add all possible solutions. However, in a sequential approach, once the planner has found a particular solution to a problem, that same solution (or similar ones) will be applied to similar problems, and the rest of the solution space need never be explored or represented.

3.4 Experiments

The goal of our experiments is to test the value of the TDR-based planning (TDRP) approach. We used a relatively simple environment in order to be able to experiment in detail with variations on robot complexity and task distributions.

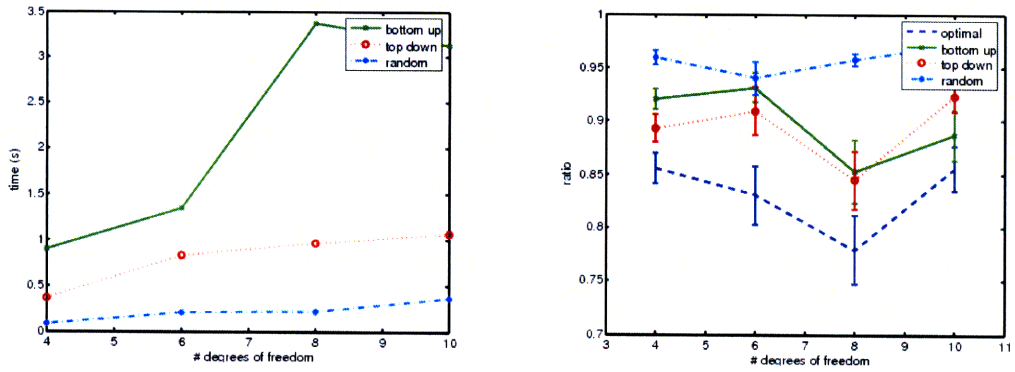


Figure 3-6: Results of experiments for decreasing swept volume of a path. The time for the brute-force optimal algorithm is not shown because it was generally around 15-20 seconds. The bottom graph shows the ratio $\frac{v_s}{v_o}$, where v_o is the swept volume of the original path, and v_s is the swept volume of the path after applying the algorithm. The top-down approach is the best of the non-optimal solutions, since it takes less time and arrives at a path with a smaller swept volume than the others.

3.4.1 Task Distribution

Our experiments were performed with a planar revolute robot arm with a fixed base. We considered four, five, six, and eight degrees of freedom. A six-degree-of-freedom example of the robot with several cylindrical obstacles in the workspace is shown in Figure 3-1. A planning problem consists of a start and goal configuration, and an obstacle placement (they are all of fixed radius). While the obstacles are moved between each planning problem presentation, they are static while the planning is being carried out.

As we discussed in section 3.1, we are interested in looking at task distributions with some variability in task instances (in both start and goal configurations and obstacle placement), but also some regularities. We therefore constructed task distributions that allowed us to control each of these aspects of the distribution individually. We also wanted to test the methods in task distributions with lots of variation and distributions with less variation. The distributions over obstacle positions, starting positions of the end effector, and goal positions of the end effector are all mixtures of

Gaussian distributions in the x and y dimensions of the workspace.

We are primarily interested in handling problems that are typically difficult for the single-query planner, so we construct a process, described below, for creating tasks that is biased in favor of difficult problems. This is less true as the variability of the world increases.

Each experiment is run on multiple task distributions with similar distributional parameters. The experimental protocol, then, is to start by specifying a set of *hyperparameters*, k_o, σ_o, k_e, and σ_e, specifying the number of clusters of obstacles (k_o) and the standard deviation of points generated by a cluster from its mean (σ_o), and the number of clusters of end-points (k_e) and the standard deviation of the endpoint clusters (σ_e). Now, to generate a task distribution, T, with n_o obstacles, we

- Choose k_e cluster means for clusters of end-effector locations, $\mu_e^1, \ldots, \mu_e^{k_e}$; the mean of each endpoint location is chosen to require the arm to be nearly fully extended, and the positions are uniformly spaced around the robot to reduce the percentage of trivial planning problems (in which the start and goal are connected by a simple path)

- Draw k_o cluster means for the location each obstacle: $\mu_o^{11}, \ldots, \mu_o^{1k_o}, \ldots, \mu_o^{n_o 1}, \ldots, \mu_o^{n_o k_o}$, uniformly at random from the x, y workspace.

Given a task distribution, T, to draw a particular task instance t, we

- Draw cluster indices j_s uniformly in $\{1, \ldots, k_e\}$ and j_g uniformly in $\{1, \ldots, k_e\}$ for the start and goal positions; Reject if $j_s = j_g$.

- Draw a start position x_s, y_s from a Gaussian distribution with mean $\mu_e^{j_s}$ and standard deviation σ_e, and goal position x_g, y_g from a Gaussian distribution with mean $\mu_e^{j_g}$ and standard deviation σ_e.

- For each obstacle o, draw a cluster index j_o uniformly in $\{1, \ldots, k_o\}$.

- For each obstacle o, draw an obstacle position x_o, y_o from a Gaussian distribution with mean $\mu_o^{1 j_o}$ and standard deviation σ_o. If the obstacle collides with either

the start or the goal configurations, reject. If sampling with mean $\mu_o^{1j_o}$ fails more than 10 times, resample the mean (previous step).

- If a position that does not collide with the start and goal configurations cannot be found for at least one obstacle after 10 attempts, resample the endpoint configurations.

Figure 4-4 shows four different task distributions. Each distribution is illustrated by overlaying five task instances drawn according to the distribution.

3.4.2 Algorithms

We compare results on a number of different versions of planning algorithms.

Single-query planner The SBL single-query probabilistic roadmap planner (Sanchez & Latombe, 2001) from the Stanford Motion Planning Kit.

Offline DRM The DRM is a TDR generated off-line in the absence of obstacles as in (McLean & Laugier, 1996; Leven & Hutchinson, 2002; Kallmann & Matarić, 2004), with a fixed size chosen experimentally.

All previous plans The TDR is built on-line, keeping all the plans found in previous task instances, up to the same fixed size as the offline DRM.

Necessary previous plans The TDR is built on-line, keeping only plans from tasks in which the existing TDR could not solve the problem.

Reduced swept volume In addition to only keeping the necessary previous plans, the plans are modified to reduce their swept volume. This is the full TDRP as described in Section 3.3.

All the results in this paper are for $k = 10$, that is, we attempt to connect each configuration in the roadmap to 10 other configurations.

3.4.3 Impact of learning

Our first set of experiments is designed to test each TDRP component's ability to take advantage of environmental regularities. Therefore, we fixed each of the variability parameters described above to small values. Our workspace was a 10 by 10 square, with an obstacle and endpoint distribution illustrated in figure 4-4A.

Before performing the experiment to compare each algorithm, we needed to determine an appropriate size for the offline DRM. The number of nodes in any roadmap technique generates a tradeoff between searching the roadmap quickly (small roadmap) and finding many solutions in the roadmap (larger roadmap). We experimented with many sizes for the offline roadmap, and found that for our problems, there was no clear best size: if the roadmap is large enough to have low probability of giving up and calling the planner, it takes so long to search the graph that it is not a clear win over calling the planner in the first place. Figure 3-7 shows this tradeoff for our eight degree-of-freedom robot. Since there did not appear to be a size that was best, we chose a fairly small size (see above) so that our roadmaps could be built in reasonable time.

The results are shown in Figure 3-8. The results for each number of degrees of freedom is the average of 10 trials with different obstacle and start/goal distributions, drawn from the same hyperparameters; each trial involved 100 planning queries. The roadmaps (both off-line and on-line) were built anew for each trial. In each of the TDR cases, the learning phase was run "to convergence", that is, until a large number of consecutive problems were solved by the TDR.

Note that building the TDR on-line and remembering all the paths substantially reduces the average time to solve a planning query, but the worst-case time is still high. The roadmap contains many redundant plans, and so therefore fails, at its predetermined maximum size, to contain solutions to a high enough percentage of new problems, so it must resort to the single-query planner for many problems.

On the other hand, when we keep only the necessary paths, we see that both the average time and the worst-case time are substantially reduced. Interestingly, if we

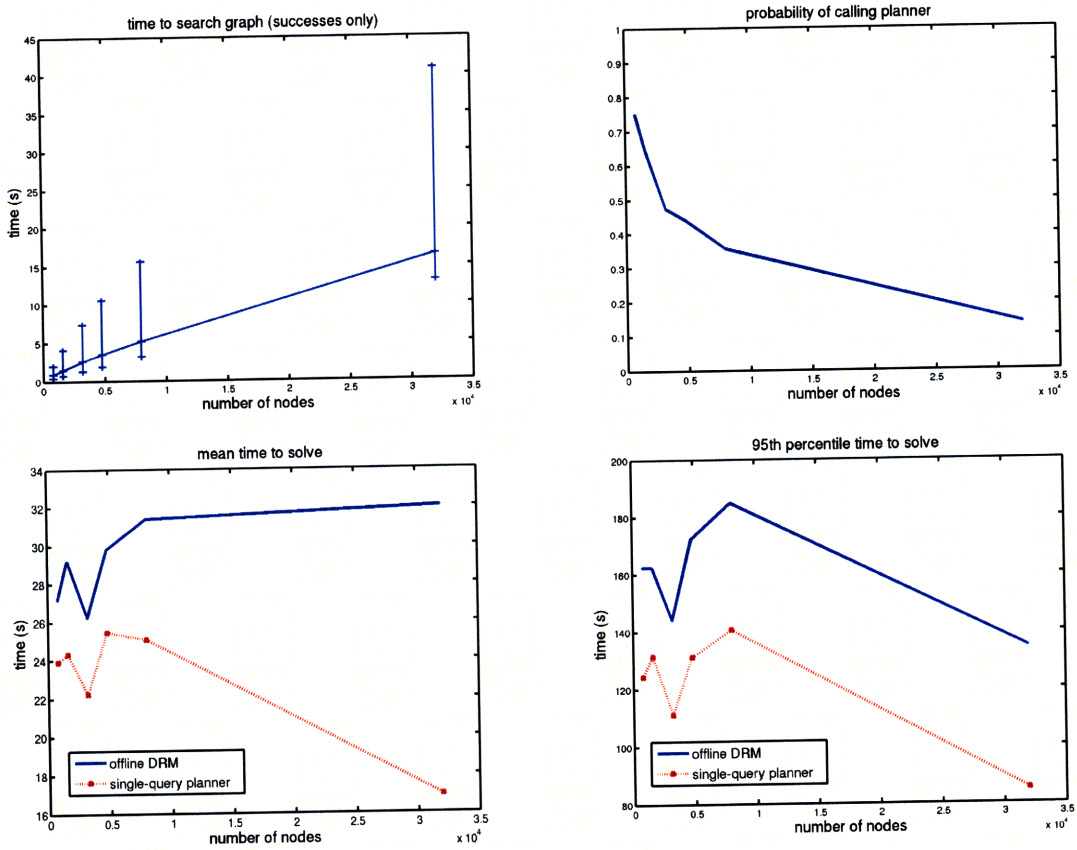


Figure 3-7: On our planning problems, there was no size for the offline DRM that gained an advantage over the single-query planner. The top two plots show the tradeoff between query time (left) and failing to find a solution (right). The bottom two plots show the mean planning time (left) and 95th-percentile planning time (right) for both the offline DRM and the single-query planner.

look at the size of the roadmap that is built, we find that while the average size is smaller than the predetermined size chosen for the roadmap of all previous plans, there are some roadmaps that actually grow to be larger. Thus we conclude that the performance advantage is due both to keeping the graph small (and therefore fast to search) when small size is adequate, and to growing the graph sufficiently to cover the space of planning problems when the environment requires it.

Lastly, we consider the effect of reducing the swept volume of each path added to the roadmap. The figure shows a substantial reduction in both the average and worst-case planning times. Both of these outcomes are due to the fact that decreasing the swept volume of a path makes it more likely that the path will be useful in future planning problems, and therefore allows the graph to stay small while still containing many relevant solutions.

3.4.4 Dealing with Disorder

Our second set of experiments is designed to see how the TDR copes with increasing variability in both the motion endpoints and the obstacle placements. We therefore tried increasing each while holding the other at a fixed, small value, and then tried increasing both simultaneously – first a little, and then a great deal, as seen in Figure 4-4.

We find that the TDR is very robust to increasing variability of start and goal locations, and is still advantageous in the face of high varying obstacle positions. Figure 3-10 shows, for the task distributions in figure 4-4A through D, the average ratio of the single-query planner time and the TDR planning time as a function of the number of planning problems that have been seen. Figure 3-11 shows the percentage of the time that the solution is found in the roadmap, rather than giving up and calling the planner.

In addition we carried out an experiment with a very highly varying task distribution (shown in figure 4-4E), because we antipcated that a roadmap approach, even one that is carefully constructed, would fail to show any advantage when the world was very random, and might in fact hurt. What we find is that since the robot is

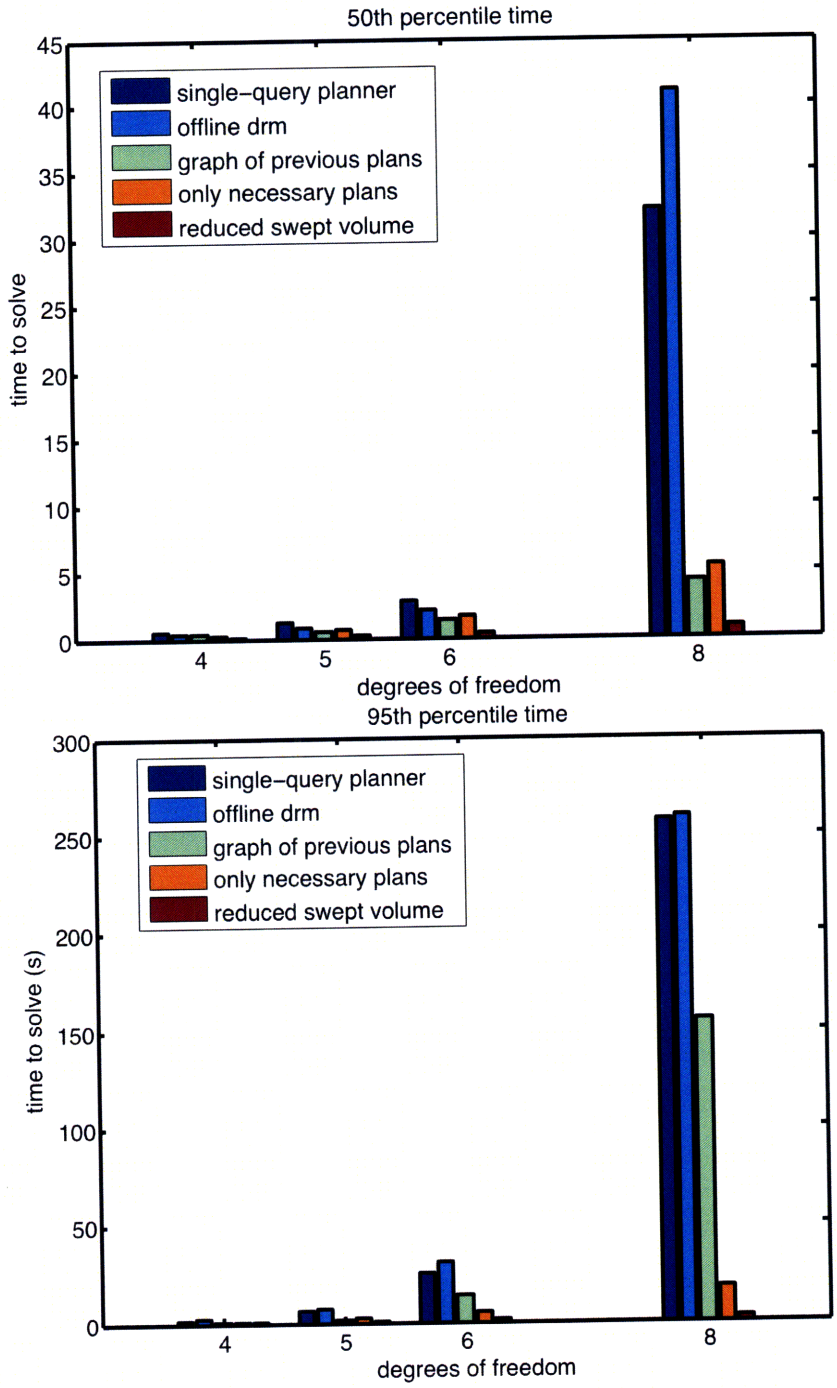


Figure 3-8: 50th percentile (top) and 95th percentile (bottom) times for single-query planner, off-line task roadmap and several versions of the TDR as a function of the number of degrees of freedom

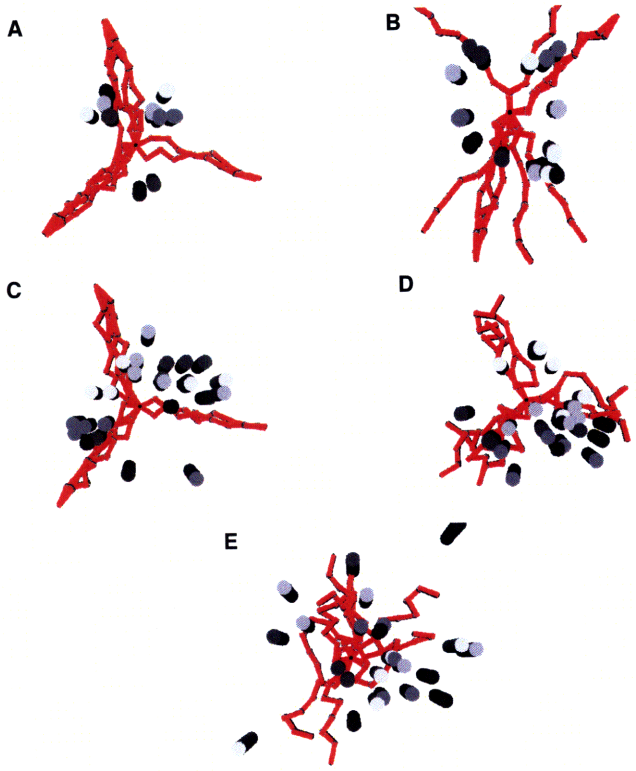


Figure 3-9: Example task distributions: A. Low variability B. Increased start/goal variability. C. Increased ostacle variability. D. Increased obstacle and endpoing variability. E. High variability

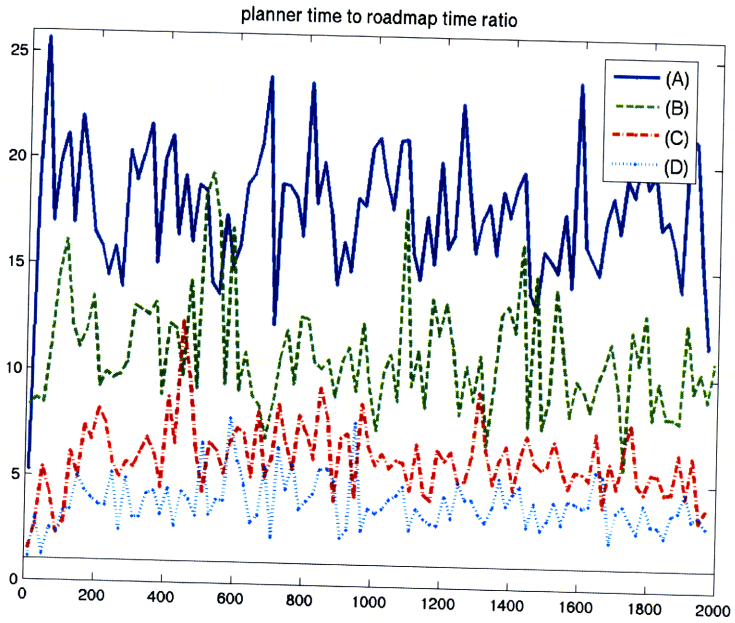


Figure 3-10: Ratio t_{sqp}/t_{tdr} of the planning time required by the single-query planner, t_{sqp} and that required by the TDR t_{tdr} as a function of the number of planning problems seen by the TDR. The thin black line shows a ratio of 1.0 for reference. The letters in the legend refer to the task distributions shown in figurefig:envs.

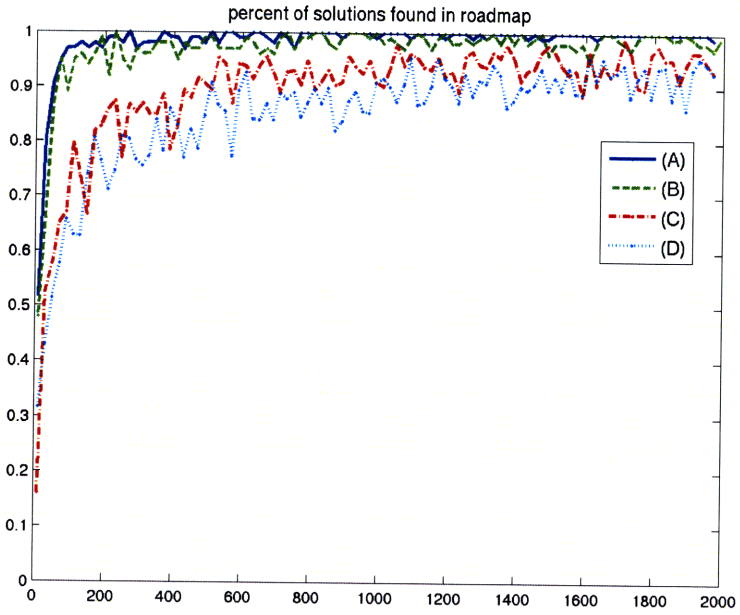


Figure 3-11: The probability that the solution is found in the TDR as a function of the number of planning problems seen.

redundant, meaning many problems have multiple possible solutions, the roadmap is able to capture at least one solution to enough problems that it achieves a 2X speedup over the SBL planner.

As expected, we see that the TDR is able to provide more substantial speedup when the tasks have lower entropy. It is interesting to note that the performance of the TDR relative to the planner is more flat than the probability of finding solutions in the graph, as a function of the number of training examples, particularly in the low-entropy cases. This is because there is a lot of speedup when the graph is small, but contains a reasonably high percentage of answers. As more training data is seen, the contribution of each point is less evident, as it makes it incrementally more likely that a similar problem will now have a solution in the graph, but makes the graph larger and therefore slower to search.

3.5 Conclusions

Our experiments show that it is possible to get significant improvements in the running time of a single-query motion planner in related tasks by building a task-distribution roadmap, which captures a relatively small set of paths that are useful for the tasks. There are a number of directions to extend this work and we briefly discuss some of these here.

First, there are a number of simple improvements that could be made to the TDR to decrease the time needed to find an answer if it is in the graph. Since the time required to get an answer from the TDR depends on the size of the graph, a number of techniques could be used to keep the number of nodes and edges small. One idea is to prune nodes and edges that are not found to be useful after additional experience. In addition, the time required to find nearest neighbors in the graph can be decreased by storing the nodes in a KD-tree or some other appropriate data structure.

The most interesting aspect of this work was the demonstration that some attempt to generalize a path before committing it to memory can decrease the number of solutions that must be remembered in order to answer queries quickly. However, there remain some questions as to the best way to perform this operation. First, since decreasing the swept volume of paths before inserting them was very useful both in decreasing the overall size of the graph, and in increasing the likelihood that the graph will be useful for new planning problems, it might prove even more useful to simply *generate* plans with smaller swept volume to start with. Thus, working to bias the planner in favor of small paths might prove a fruitful direction. In fact, we should bias the paths to have small swept volume in areas where obstacles frequently appear.

Our approach for decreasing the swept volume could also be improved upon. For example, because we were interested in running many experiments quickly, we considered only removing nodes from the path to decrease its volume. Paths could almost certainly be improved more by also considering the judicious addition of nodes. Platform-based heuristics could also be used to improve the process. Lastly, swept

volume may not be the only worthwhile property to consider when working to generalize solutions; something like manipulability might also be used in this way.

As our results showed, the TDR can solve some very hard problems very quickly, but when the agent is faced with hard problems only occasionally, the overhead of the TDR may be a liability in terms of average time needed to solve the problems faced. One method for alleviating this problem might be to develop a fast way to tell whether a planning problem is likely to be difficult for the single-query planner. We might then hope to get the best of both worlds.

A clear weakness of the TDR is that there is no abiltiy to modify a path from the graph that doesn't quite work in the current circumstances, but would require only small modifications. One possible way to address this within the task distribution roadmap framework would be allowing pieces of the roadmap to move locally in response to changes in the environment, since local changes are often reasonably easy to compute. Since our work on the task distribution roadmap presented in this chapter, this idea of a flexible roadmap has been explored by others (Yang & Brock, 2006; Gayle et al., 2007). Rather than extending the task distribution roadmap to deal with the lack of flexibility, however, we have chosen to address the issue with a technique that combines the desire to focus on the difficult problems and the notion of a parameterized path. This technique is described in the next chapter.

Chapter 4

Predicting partial paths

The primary weakness of the memorization technique described in the previous chapter was that a small change in the environment can cause very large regions of the solution memory to be invalid. To address this, we would instead like to learn a parameterization of the old solutions that allows us to adapt them quickly to similar but novel problems. Also, since existing probabilistic techniques already deal quite effectively with easy planning problems (those that do not contain narrow passages in configuration space), we would like to focus not just on the difficult problems, but on the difficult parts of the difficult problems. Thus a natural parameterization would be one that characterizes the narrow passage in input space.

In general, such parameters may not be easy to find: seemingly harmless regions of the workspace can correspond to narrow passages in configuration space. However, there are common difficult problems that do have identifiable features in the workspace. A small hole that the robot is requires to move or reach through, or a pre-grasp configuration that has an object enclosed in the robot's hand are examples of such configuration-space narrow passages. Given a set of tasks such as these, we will use the notion of a "task template", or a parameteric description (in the workspace) of a particular instance of the task.

Even given a small set of narrow-passage tasks, it will still be difficult in many real-world applications to identify the appropriate parameters from the robot's sensor input. One possible scenario for obtaining task template parameters might involve

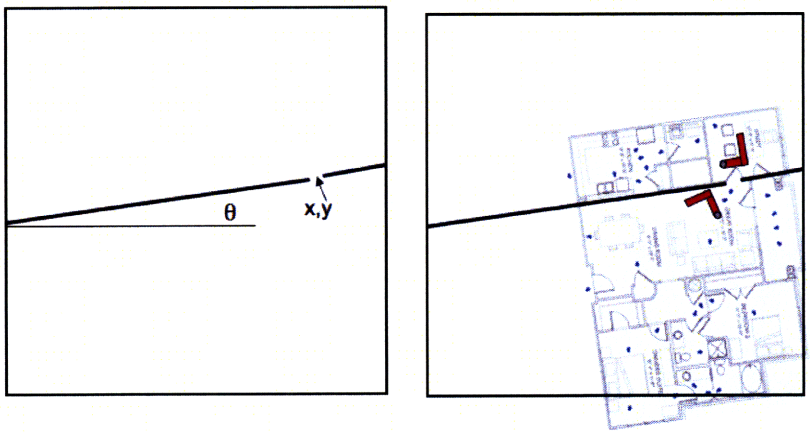


Figure 4-1: A task template for a doorway task. Left: Parameterization of the template. Right: Application of task tmplate to a complex environment. Start and goal configurations for a mobile 2-degree of freedom robot are shown below and above the door (in red).

teleoperation of the robot, in which a human operator could fit the template to a particular instance. In general, we do not expect that extracting task parameters from sensor data automatically will be easy. This thesis does not address this issue; instead we assume the ability to derive problem parameters in some way, and investigate the possible utility of such parameters in solving the motion planning problem. Figure 4-1 shows an example of how a task template that describes a tight doorway might be instantiated.

Our approach involves learning a function from task template parameters to a partial motion plan that is relevant to solving the overall task. This plan segment is then passed to a modified version of the SBL planner (Sanchez & Latombe, 2001) which is then able to solve the full planning problem more quickly. The training data for the learning algorithm is generated by using the unmodified SBL planner to solve task template instances, and then extracting the difficult part of the plan.

4.1 Generating training data

For each task template, we assume a source of training task instances of that template. These instances might be synthetically generated at random from some plausible

distribution, or be drawn from some source of problems encountered in the world in which the robot will be operating.

We begin with a set of task instances, $t^1, \ldots, t^n$, where each task instance is a specification of the detailed problem, including start and goal configurations and a description of the workspace obstacles. These descriptions will generally be non-parametric in the sense of not having a fixed size. We will convert each of these tasks and their solution paths into a training example, $\langle x^i, y^i \rangle$, where x^i is a parametric representation of t^i and y^i is a path segment. Intuitively, to get x^i we instantiate the task template for the particular task at hand. In the case of the doorway example this means identifying the door, and finding its position and orientation, x_{door}, y_{door} and θ_{door}.

We assume that each t^i in the training set is "labeled" with a parametric description x^i. To generate the y^i values, we begin by calling the single-query planner on task instance t^i. If the planner succeeds, it returns a path $p = \langle c_1, ..., c_r \rangle$ where the c's are configurations of the robot, connected by straight-line paths.

We do not, however, want to use the entire path to train our learning algorithm. Instead we want to focus in on the parts of the path that were most difficult for the planner to find, so we will extract the most constrained portion of the path, and use just this segment as our training data. Algorithm 4.1 shows an outline of the way in which the most constrained path segments are extracted. We describe the algorithm below.

Before we analyze the path, we would like to reduce the number of unnecessary configurations generated by the probabilistic planner, so we begin by smoothing the path. This process involves looking for non-colliding straight-line replacements for randomly selected parts of the path, and using them in place of the original segment.

Since we are interested in judging the extent to which each configuration in the path is constrained, we need to be careful when smoothing the path. In general, replacing segments with straight-line segments whenever possible tends to generate paths that go very close to obstacles, even when it is not necessary. This may make it appear that some configurations are more constrained than they need to be. Thus, we

Algorithm 4.1 ExtractConstrainedSegments(*path*, *length*) : *subpaths*

```
1: smoothPath = SmoothPathCheckClearance(path)
2: resampledPath = ResamplePath(smoothPath, d)
3: constrainedList = MeasureConstrained(resampledPath)
4: t = min(constrainedList)*p + max(constrainedList)*(1.0-p)
5: constrainedSegs = SegmentsAboveThresh(constrainedList,
   resampledPath, t)
6: for all segment ∈ constrainedSegs do
7:    subpath = AdjustPathLength(segment, length)
8:    if CollisionFree(subpath) and
      (PlannerTime() > 10*PlannerTime(segment)) then
9:       subpaths.Add(subpath)
10:   end if
11: end for
```

are careful, while smoothing, to replace segments only with staight-line paths that do not decrease the clearance, or minimum distance between the robot and an obstacle.

We also want to sample the paths at uniform distance intervals, so that paths that follow similar trajectories through the workspace are similar when represented as sample sequences. Thus, we first resample the smoothed path at a higher resolution, generating samples by interpolating along p at some fixed distance between samples, resulting in a more finely sampled path.

Given this new, finely sampled path p', we now want to extract the segment that was most constrained, and therefore difficult for the planner to find. In fact, if there is more than one such segment, we would like to find them all. For each configuration in the path, we draw some number of samples from a normal distribution centered on that configuration (sampling each degree of freedom independently). Each sample is checked for collision, and the ratio of the number of colliding samples to the total number of samples is used as a measure of how constrained the configuration is.

We then set a threshold to determine which configurations are tight, and find the segments of contiguous tight configurations. Our outputs need to have a fixed length l, so for all segments less than l, we resample the path at appropriate granularity. For segments with length greater than l, we skip configurations as appropriate.

Lastly, we want to avoid adding points to our data set that are not actually helpful.

We can judge this by calling the planner with the proposed path segment, and adding the segment to our list only if the planning time is significantly improved. In this way, we extract each of the constrained path segments from the solution path. Each of these, paired with the parametric description of the task, becomes a separate training example.

4.2 Learning from successful plans

Each of our n training instances $\langle x^i, y^i \rangle$ consists of a task-template description of the world, x^i, described in terms of m parameters, and the constrained segment from a successful path, y^i, as described above. The configurations have consistent length d (the number of degrees of freedom that the robot has), and each constrained segment has been constructed to have consistent length l, therefore y^i is a vector of length ld.

At first glance, this seems like a relatively straightforward non-linear regression problem, learning some function f^* from an m-dimensional vector x to an ld-dimensional vector y, so that the average distance between the actual output and the predicted output is minimized. However, although it would be useful to learn a single-valued function that generates one predicted y vector given an input x, our source of data does not necessarily have that form. In general, for any given task instance, there may be a large set of valid plans, some of which are qualitatively different from one another.

For instance, members of different homotopy classes should never be averaged together. Consider a mobile robot going around an obstacle. It could choose to go around either to the left of the obstacle, or to the right. In calling a planner to solve such problems, there is no general way to bias it toward one solution or the other; and in some problems there may be many such solutions. If we simply took the partial paths from all of these plans and tried to solve for a single regression function f from x to y, it would have the effect of "averaging" the outputs, and potentially end up suggesting partial paths that go through the obstacle.

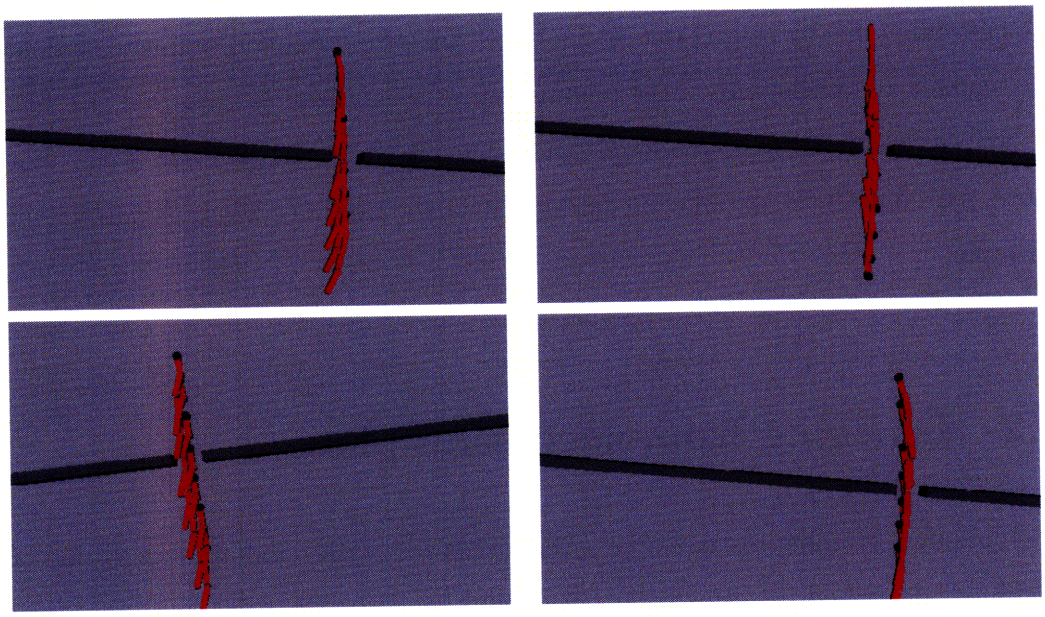


Figure 4-2: Top: Different strategies proposed for the same environment. The strategy on the left has the arm trailing the base (indicated by the black circle), while the strategy on the right goes through arm first. Bottom: Suggested partial paths generated by the same strategy for different environments.

4.2.1 Mixture model

To handle this problem, we have to construct a more sophisticated regression model, in which we assume that data are actually drawn from a mixture of regression functions, representing qualitatively different "strategies" for negotiating the environment. Figure 4.2.1 shows strageies relevant for solving our doorway example with a 6 degree-of-freedom mobile arm. Note that the two suggestions generated by different strategies for the same environment are incompatible: the average of the two strategies produces a colliding path.

We learn a generative model for the selection of strategies and for the regression function given the strategy, in the form of a probabilistic mixture of h regression models, so that the conditional distribution of an output y given an input x is

$$\Pr(y|x) = \sum_{k=1}^{h} \Pr(y|x, s = k) \Pr(s = k|x) \quad .$$

where s is the mixture component responsible for this example.

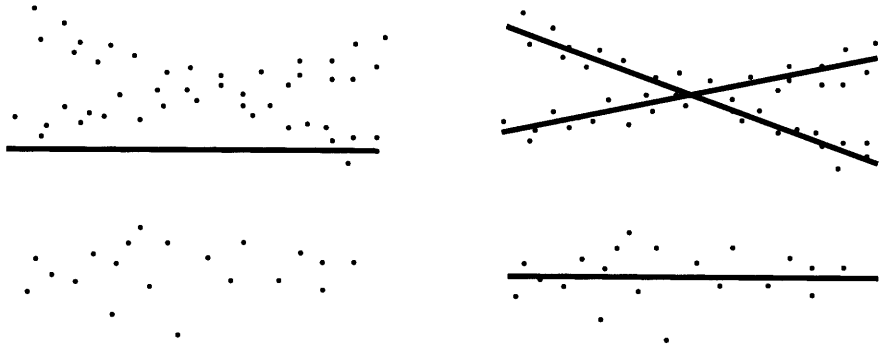


Figure 4-3: Data with linear regression (left) and mixture regression (right) fits.

Figure 4-3 illustrates a data set with this non-functional property in which the x and y are one dimensional. In the first frame, we show the best single linear regression line, and in the second frame, a mixture of linear regression models. It is clear that a single linear (or non-linear, for that matter) regression model is inappropriate for this data.

For each strategy, we assume that the components of the output vector are generated independently, conditioned on x; that is, that

$$\Pr(y|x, s = k) = \prod_{j=1}^{ld} \Pr(y_j|x, s = k) \quad .$$

Note that this applies to each configuration on the path as well as each coordinate of each configuration. Thus, the whole partial path is being treated as a point in an ld-dimensional space and not a sequence of points in the configuration space. The parameter-estimation model will cluster paths such that this independence assumption is satisfied, to the degree possible, in the model.

Then, we assume that each y_j has a linear dependence on x with Gaussian noise, so that

$$\Pr(y_j|x, s = k) = \frac{1}{\sigma_{jk}\sqrt{2\pi}} \exp\left(-\frac{(y_j - w_{jk} \cdot x)^2}{2\sigma_{jk}^2}\right) \quad ,$$

where σ_{jk} is the standard deviation of the data in output dimension j of strategy k from the nominal line, and w_{jk} is an m-dimensional vector of weights specifying the dependence of the mean of the distribution of y_j in strategy k on x as the dot product of w_{jk} and x.

It would be possible to extend this model to contain non-linear regression models for each mixture component, and it might be useful to do so in the future. However, the current model can approximate a non-linear strategy function by using multiple linear components.

In our current model, we assume that $\Pr(s = k|x)$ is actually independent of x (we may wish to relax this in future), and define $\pi_k = \Pr(s = k)$. So, finally, we can write the log likelihood of the entire training set $LL(\sigma, w, \pi)$, as a function of the parameters σ, w, and π, (note that each of these is a vector or matrix of values), as

$$\sum_{i=1}^{n} \log \sum_{k=1}^{h} \pi_k \prod_{j=1}^{ld} \frac{1}{\sigma_{jk}\sqrt{2\pi}} \exp\left(-\frac{(y_j^i - w_{jk} \cdot x^i)^2}{2\sigma_{jk}^2}\right) .$$

We will take a simple maximum-likelihood approach and attempt to find values of σ, w, and π that maximize this likelihood, and use that parameterization of the model to predict outputs for previously unseen values of x.

The model described in this section is essentially identical to one used for clustering trajectories in video streams (Gaffney & Smyth, 1999), though their objective is primarily clustering, where ours is primarily regression.

4.2.2 Parameter estimation

If we knew which training points to assign to which mixture component, then the maximum likelihood parameter estimation problem would be a simple matter of counting to estimate the π_k and linear regression to estimate w and σ. Because we don't know those assignments, will will have to treat them as hidden variables. Let $\gamma_k^i = \Pr(s^i = k|x^i, y^i)$ be the probability that training example i belongs to mixture component k. With this model we can use the expectation-maximization (EM) algorithm to estimate the maximum likelihood parameters.

We start by doing an agglomerative clustering of a subset of our data. The point-to-point distance metric used for the clustering is Euclidian distance on the concatentation of x^i and y^i. Cluster distance is computed as the distance between the means of the clusters. We cluster until we have as many clusters as we want

mixture components, and then for each component we initialize γ_k^i to be high if the point $\langle x^i, y^i \rangle$ is in the correponding cluster, and low otherwise. We cluster enough of our data to ensure that each component has enough points to make the linear regession required in the M step described below be well-conditioned.

In the expectation (E) step, we temporarily assume our current model parameters are correct, and use them and the data to compute the responsibilities:

$$\gamma_k^i := \frac{\pi_k \Pr(y^i | x^i, s^i = j)}{\sum_{a=1}^{h} \pi_a \Pr(y^i | x^i, s^i = a)} \quad .$$

In the maximization (M) step, we temporarily assume the responsibilities are correct, and use them and the data to compute a new estimate of the model parameters. We do this by solving, for each component k and output dimension j, a weighted linear regression, with the responsibilities γ_k^i as weights. Weighted regression finds the weight vector, w_{jk}, minimizing

$$\sum_i \gamma_k^i (w_{jk} \cdot x^i - y_j^i)^2 \quad .$$

When the regression is numerically ill-conditioned, we use a ridge parameter to regularize it.

In addition, for each mixture component k, we re-estimate the standard deviation:

$$\sigma_{jk} := \frac{1}{\sum_i \gamma_j^i} \sum_i \gamma_j^i (w_{jk} \cdot x^i - y_j^i)^2 \quad .$$

Finally, we reestimate the mixture probabilities:

$$\pi_j := \frac{1}{n} \sum_{i=1}^{n} \gamma_j^i \quad .$$

This algorithm is guaranteed to find models for which the log likelihood of the data is monotonically increasing, but has the potential to be trapped in local optima. To ameliorate this effect, our implementation does random re-starts of EM and selects the solutions with the best log likelihood.

The problem of selecting an appropriate number of components can be difficult. One standard strategy is to try different values and select one based on held-out data or cross-validation. In this work, since we are ultimately interested in regression outputs and not the underlying cluster structure, we simply use more clusters than are likely to be necessary and ignore any that ultimately have little data assigned to them.

4.3 Using the model

Given the model parameters estimated from the training data, we can generate suggested partial paths from each strategy, and use those to bias the search of a sample-based planner. In our experiments, we have modified the SBL planner to accept these suggestions, but we expect that most sample-based planners could be similarly modified. We describe the way in which we modified the SBL planner below.

For a new planning problem with task-template description x, each strategy k generates a vector $y^k = w_k \cdot x$ that represents a path segment. However, in the new planning environment described by x, the path segment may have collisions. We want to avoid distracting the planner with bad suggestions, so we collision-check each configuration and each path between consecutive configurations, and split the path into valid subpaths. For example, if y is a suggested path, consisting of configurations $\langle c_1, \ldots, c_{15} \rangle$, imagine that configuration c_5 collides, as do the paths between c_9 and c_{10} and between c_{10} and c_{11}. We would remove the offending configurations, and split the path into three non-colliding segments: $\langle c_1 \ldots c_4 \rangle$, $\langle c_6 \ldots c_9 \rangle$, and $\langle c_{11} \ldots c_{15} \rangle$. All of the non-colliding path suggestions from each strategy are then given to the modified SBL planner, along with the initial start and goal query configurations.

The original SBL planner searches for a valid plan between two configurations by building two trees: T_s, rooted at the start configuration, and T_g, rooted at the goal configuration. Each node in the trees corresponds to a robot configuration, and the edges to a linear path between them. At each iteration a node, n, from one

of the trees is chosen to be expanded.[1] A node is expanded by sampling a new configuration n_{new} that is near n (in configuration space) and collision-free. In the function CONNECTTREES, the planner tries to connect T_s and T_g via n_{new}. To do this, it finds the node n_{close} in the other tree that is closest to n_{new}. If n_{close} and n_{new} are sufficiently close to one another, a candidate path from the start to the goal through the edge between n_{new} and n_{close} is proposed. At this point the edges along the path are checked for collision. If they are all collision-free, the path is returned. If an invalid edge is found, the path connecting the two trees is broken at the colliding edge, possibly moving some nodes from one tree to the other.

We extended this algorithm slightly, so that the query may include not only start and goal configurations, c_s and c_g, but a set of h path segments, $\langle p_1, ..., p_h \rangle$, where each path segment p_i is a list of configurations $\langle c_{i1}, ..., c_{ir} \rangle$. Note that r may be less than l, since nodes in collision with obstacles were removed. We now root trees at the first configuration c_{i1} in each of the path segments p_i, adding h trees to the planner's collection of trees. The rest of the configurations in each path segment are added as linear descendants, so each suggestion tree starts as a trunk with no branches.

Importantly, the order of the suggestions (barring those that were thrown out due to collisions) is preserved. This means that the suggested path segments are potentially more useful than a simple collection of suggested collision-free configurations, since we have reason to believe that the edges between them are also collision-free.

The original SBL algorithm chose with equal probability to expand either the start or goal tree. If we knew that the suggestions were perfect, we would simply require them to be connected to the start and goal locations by the planner. However, it is important to be robust to the case in which some or all of the suggestions are unhelpful. So, our planner chooses between the three *types* of trees uniformly: we choose the start tree and goal tree each with probability $1/3$, and one of the k previous path trees with probability $1/(3k)$. When we have generated a new node, we must

[1]The SBL planning algorithm has many sophisticated details that make it a high-performance planner, but which we will not describe here, such as how to decide which configuration to expand. While these details are crucial for the effectiveness of the planner, we did not alter them, and so omit them for the sake of brevity.

now consider which, if any, of our trees to connect together. Algorithms 4.2 and 4.3 show pseudo-code for this process. The overall goal is to consider, for each newly sampled node n_{new}, any tree-to-tree connection that n_{new} might allow us to make, but still to leave all collision checking until the very end, as in the SBL planner.

T_{expand} is the tree that was just expanded with the addition of the newly sampled node n_{new}. If T_{expand} is either the start or goal tree (T_s and T_g respectively), we first try to make a connection to the other endpoint tree, through the CONNECTTREES function in the SBL planner. If this function succeeds, we have found a candidate path and determined that it is actually collision free, so we have solved the query and need only return the path.

If we have not successfully connected the start and goal trees, we consider making connections between the newly expanded tree and each of the previous path trees, T_i. The MERGETREES function tries to make these connections. If a connection can be made between the new node n_{new} and a node $n_{connect}$ in some tree T_i, T_i is restructured so that $n_{connect}$ is the root of the tree, and then attached to n_{new}, combining the two trees into one. The paths between these trees are not checked for collisions at this point. As mentioned above, the original algorithm has a distance requirement on two nodes in different trees before the nodes can be considered a link in a candidate path. We use the same criterion here. If the new node is found to be close enough to a node in another tree, we reroot T_i at node n_{close}, and then graft the newly structured tree onto T_{expand} at node n_{new}. If one connection is successfully made, we consider no more connections until we have sampled a new node.

4.4 Experiments

We have experimented with task-templates in several different types of environment, to show that learning generalized paths in this way can work, and that the proposed partial paths can be used to speed up single-query planning. We found that the technique was quite successful in several environments, and improves planning time significantly in all of our domains, but provides less improvement in the more difficult domains. We present the results in each domain here, and then investigate the more

Algorithm 4.2 TRYCONNECTIONS(n_{new}, T_{expand}) : Boolean

```
1: success ← false
2: if T_expand == T_g then
3:    success ← connectTrees(n_new,T_expand, T_s)
4: else if T_expand == T_s then
5:    success ← connectTrees(n_new,T_expand, T_g)
6: else
7:    for previous path trees T_i ≠ T_expand do
8:       merged ← mergeTrees(n_new, T_expand,T_i)
9:       if merged then
10:         break
11:      end if
12:   end for
13: end if
14: return success
```

Algorithm 4.3 MERGETREES(n_{new}, T_{expand}, T_{elim}) : Boolean

```
1: n_close ← T_elim.CLOSESTNODE(n_new)
2: success ← CLOSEENOUGH(n_close, n_new)
3: if success then
4:    REROOTTREE(n_close, T_elim)
5:    GRAFTTREE(n_new, n_close, T_expand, T_elim)
6: end if
```

challenging domains in greater detail in chapter 4. The simplest environment is depicted in figure 4-2, and the rest are shown in figure 4-4. Each illustration shows the obstacle placement and the start and goal locations for a particular task instance.

The simplest environment is the doorway environment. It is just a wall with a hole in it, where the location and orientation of the wall changes. The robot is a mobile arm with a variable number of links in the arm, so that we can carefully explore how well the algorithms scale. Experiments done in this very simple domain were intended to be a proof of concept for both the generalized path learning from probabilistically generated paths, and the integration of suggested paths with the SBL planner.

The doorway in this domain, and the following one, is parameterized by the x, y position of the door, and the angle of the wall relative to horizontal. We chose a tight spot segment length of 7 for this domain, and we allowed EM to use 8 clusters.

The cluttered doorway domain extends the doorway domain slightly by introducing random obstacles during testing. The task template for this domain is the same as for the previous one, and in both cases training data was gathered in the absence of obstacles. Since randomly placing obstacles may completely obscure the door, we report planning times only for problems for which either the planner alone, or the modified planner, was able to find a solution.

The "bug trap" domain is a fairly standard motion planning challenge. This domain increases the required accuracy of the suggestions, as the hallway for getting out of the trap is long and narrow. The bug trap domain was parameterized by the angle of rotation of the trap around the center. Rather than using the raw angle, we used the sine and cosine of the angle, as the function function from the input parameters to the x and y degrees of freedom is simpler in that space. We also used degree three polynomial features, as we found experimentally that this feature space worked the best.

The angle and zig zag corridors are designed to show that the learning algorithm can learn more complicated path segments. These domains are tricky because the constrained portion is more complicated than in the previous examples. This means the learning is more difficult, because the training data must be well-aligned in order

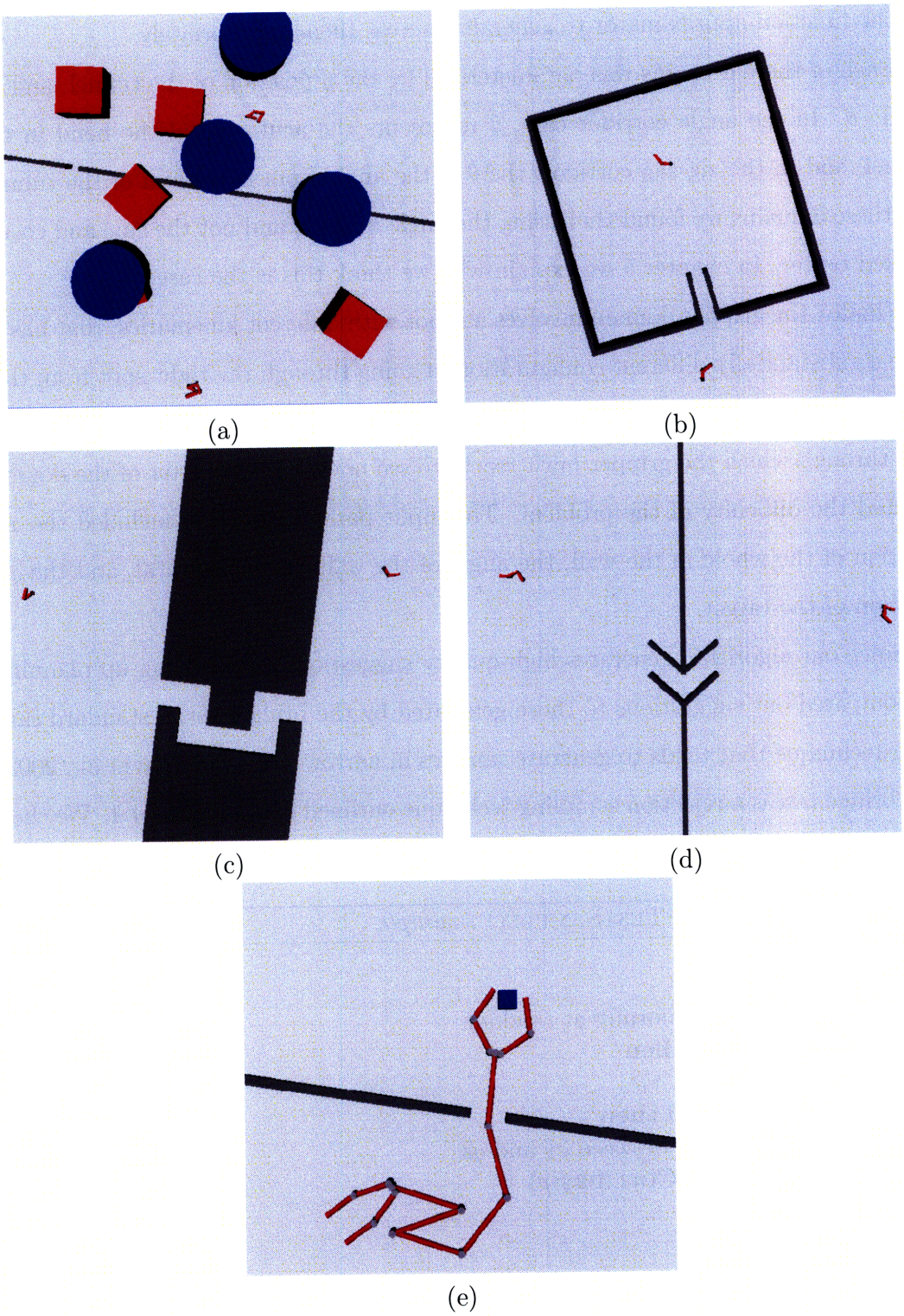


Figure 4-4: Four types of environment for experimental testing. The robot in each environment is a planar mobile arm with varying num of degrees of freedom, from 4 to 7. We will refer to the domains as (a) cluttered doorway, (b) bug trap, (c) zig zag corridor, (d) angle corridor, and (e) gripper.

for the function approximator to generalize across them appropriately.

Each of these domains was parameterized by the y position of the tunnel, and an angle, θ. In the angle corridor case, θ represents the acuteness of the bend in the tunnel, and in the zig-zag corridor, the θ is the angle from horizontal of the tunnel. For these domains we found that using the angle θ alone, and not the sine and cosine worked better. In chapter 5 we explain why we think this is the case.

Finally, the gripper domain involves a robot with different kinematics, that has to have a substantially different configuration in going through the tight spot from that at the goal. We did experiments with narrow, wide, and medium-width holes in the wall through which the gripper must reach, to see how this parameter of the domain effected the difficulty of the problem. The input parameterization included the x-y position of the whole in the wall, the angle of the wall from horizontal, and the x-y position of the target.

Since our algorithm generates high-quality suggestions for speeding up planning, we compared our suggestions to those generated by the bridge test, a standard sampling technique that tends to generate samples in narrow passages (Hsu et al., 2003). The bridge test is a rejection sampling technique outlined in algorithm 4.4. We chose the variance used in the Gaussian sampling step in line 5 by hand.

Algorithm 4.4 BRIDGETESTSAMPLE() : *sample*

```
1: found = false
2: while not found do
3:    Pick c1 from C uniformly at random
4:    if COLLIDES(c1) then
5:       Pick c2 ~ N(c1, σ)
6:       if COLLIDES(c2) then
7:          p = midpoint between c1 and c2
8:          found = not COLLIDES(p)
9:       end if
10:   end if
11: end while
12: return p
```

When our algorithm generated s suggested subpaths, each consisting of a subpath of l configurations, we generated both s and $s * l$ bridge samples, and used both sets

as suggestions for the planner. We wanted to allow the bridge test to generate enough samples to be useful, but not so many that it distracted the planner. We report the best of the two bridge sample times. We also neglect the time spent to generate the bridge samples, as we are primarily interested in evaluating the utility of the samples to the planner, and so did not optimize the bridge test sampling in any way. We note, however, that there is a not insubstantial cost associated with generating a large number of bridge samples in our domains, as there are few narrow passages relative to the size of the space. With our simple implementation, we required anywhere from a few seconds to a few minutes to generate the number of samples required to get the reported speedup.

A summary of the results in these domains is shown in figure 4-5. We compare the time spent by the unmodified SBL planner to the time required by the modified SBL planner supplied with bridge test suggestions and suggestions generated by our partial-path-suggestion algorithm. For each test, all algorithms were applied to the same planning problem. Each time measurement was an average of 10 training trials, in which the model was trained on 300-500 data points (except on the simple gripper experiment, which used 2000), and then tested on 100 planning problems. The running times for the partial-path-suggestion algorithm include the time required to generated the suggestions. We discuss the results for each domain in detail below.

4.4.1 The door task

The experiments in the simple door domain show that the suggestions can dramatically decrease the running time of the planner, with speed-ups of approximately 2, 8, 31 and 78 for the different number of degrees of freedom. For the robots with 4, 5, and 6 degrees of freedom, both methods returned an answer for every query. In the 7-DOF case, the unmodified planner found a solution to 99% of the queries, while the suggestions allowed it to answer 100% successfully within the allotted number (100,000) of iterations.

When obstacles were added to the doorway domain, the partial path suggestions are still helpful, but provide less leverage than the in uncluttered doorway domain.

domain	# training pts	average planner time	average time with bridge samples	average time with partial path suggestions	speedup with suggestions
4 DOF door	300	0.44 s	0.21 s	0.20 s	2X
5 DOF door	300	1.6 s	0.29 s	0.20 s	8X
6 DOF door	300	9.5 s	0.86 s	0.30 s	31X
7 DOF door	300	67.5 s	7.3 s	0.86 s	78X
6 DOF cluttered door	300	39.8 s	48.7	4.8 s	8X
4 DOF bug trap	1000	11.0 s	3.65 s	0.32 s	34X
4 DOF angle corr.	500	4.7 s	1.7 s	0.75 s	6X
4 DOF zig-zag corr.	500	28.55 s	10.12 s	4.01 s	7X
simple gripper narrow	2000	155.13 s	147.70 s	78.05 s	2X
simple gripper med.	2000	18.1 s	17.4 s	9.7 s	<2X
simple gripper wide	2000	7.2	5.2	4.3 s	<2X

Figure 4-5: Planning performance results.

Figure 4-6 shows two partial path suggestions, one of which is only slightly useful to the planner. Nonetheless, the suggestions do, on average, speed up the planner by a factor of over 8.

The comparison to the bridge test is interesting. In the low degree-of-freedome cases, the bridge test provides assistance to the planner that is comparable to our partial path suggestions (leaving aside the time required to generate the bridge samples), but the samples provided by the bridge test are less useful as the number of degrees of freedom go up. While the bridge test has proven very useful in generating narrow passage sample for rigid bodies, it is much more difficult to find these useful samples when robot becomes more complicated. Similarly, when the obstacles are added to the domain, the bridge test becomes distracted by the narrow passages between the obstacles. In most cases, these suggestions distract the planner from the more crucial narrow passage of the doorway. Figure 4-7 shows examples of samples which fulfill the bridge criteria, but are not as useful to the planner.

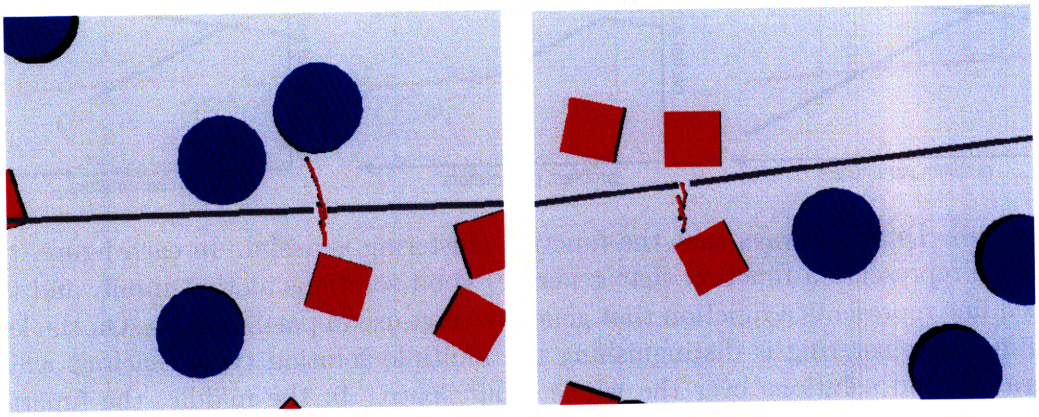


Figure 4-6: Partial path suggestions generated by our algorithm for the cluttered doorway domain, with colliding configurations removed. The example on the left will clearly be very useful to the planner; it contains 5 collision-free configurations, and the endpoints of the partial path have the robot completely on either side of the door. However the partial path suggestion on the right is less helpful. The planner solves this problem on average in 2.8 seconds, whereas the suggestions allow it to be solved in 1.4 seconds, a 2X speedup.

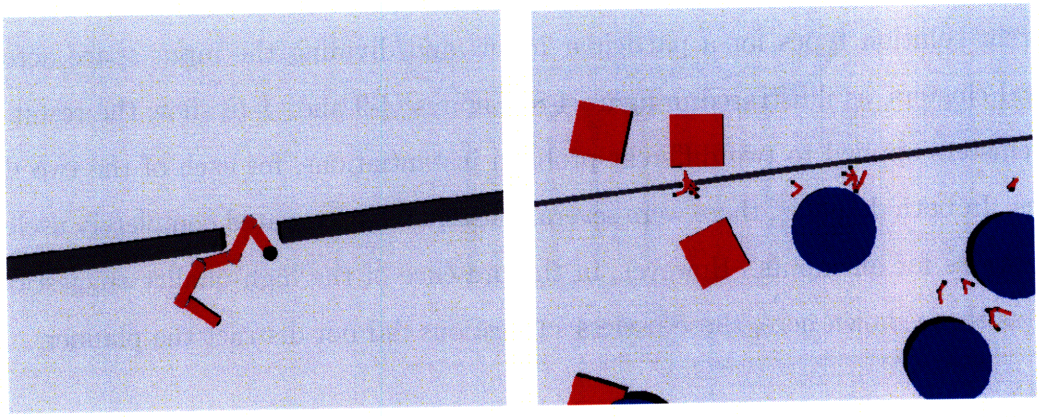


Figure 4-7: Suggestions generated by the bridge test. On the left is an example of how the higher degree-of-freedom platform caused difficulty for the bridge test. This sample is indeed in the narrow passage, but it is unlikely to be useful to the planner nonetheless. On the right we see that many of the samples generated by the bridge test are near the distractor obstacles rather than in the doorway.

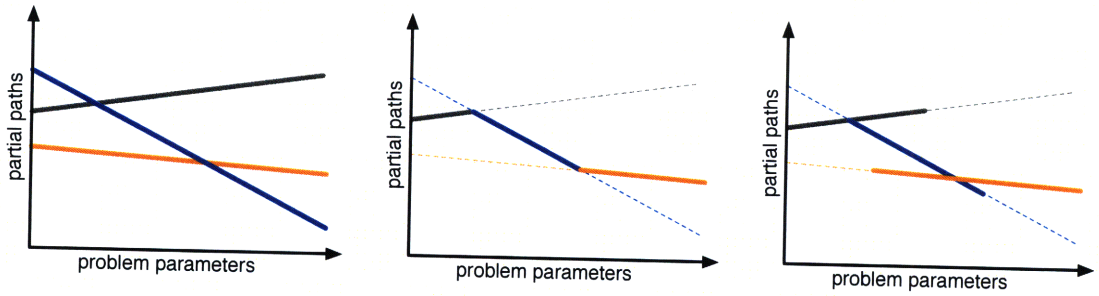


Figure 4-8: Different ways that the function clustering is useful. In each figure, the bold line represents a function that generates good solutions at that input, and the dashed line represents a function that generates non-useful partial paths. On the left, the function clustering is distinguishing the multiple solution types, each of which generates good solutions over the whole input space. In the middle, the function clustering divides up the input space, and each region of the input space is covered by just one function. On the right is a more realistic scenario, in which the function clustering finds functions that cover overlapping parts of the input space, but do not cover the whole space individually.

4.4.2 The bug trap and 4-DOF angle corridor

The bug trap and low degree-of-freedom angle corridor domains were both solved almost completely by the partial path suggestions, in the sense that at least one suggestion was collision-free and went from one side of the tight spot to the other. In both cases, the function clustering performs the duel duty of distinguishing the different solution types for a particular input, and dividing the input space across several clusters, as illustrated in figure 4-8. Figures 4-9 and 4-10 show the result of two clusters applied to two different problem instantiations, for each of the two domains. In both domains, there were several clusters that generated completely useless suggestions for all inputs. However, in the presence of the high quality suggestions such as those shown here, those useless suggestions did not distract the planner.

4.4.3 6-DOF angle and zig zag corridors

In the higher degree-of-freedom angle and zig zag corridors domains, the suggestions still produced substantial speedup, but it was not always the case that at least one suggestion was entirely collision-free as in the previous domains. In these more challenging domains, there were often holes left when the colliding configurations were

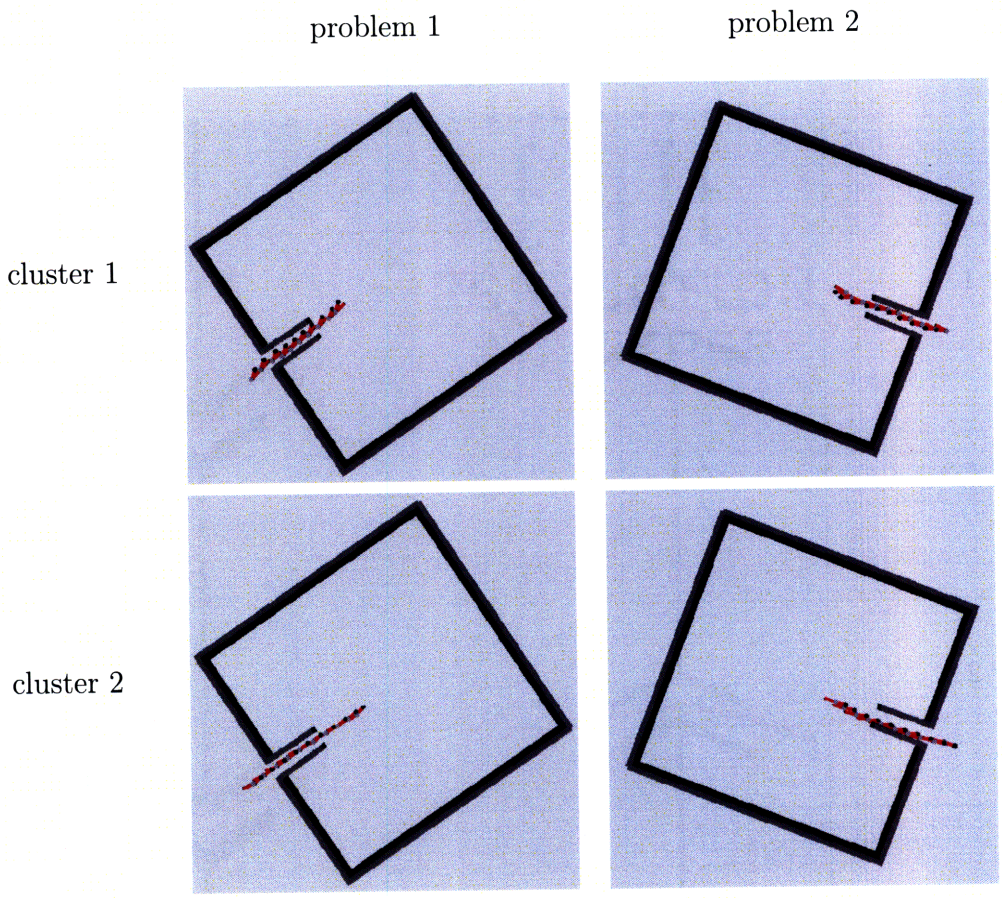


Figure 4-9: Two different strategies for solving the bug trap domain, applied to two different problem instances. The solution generated by cluster 2 for problem 2 includes a few colliding configurations in the middle of the path, but is useful to the planner nonetheless.

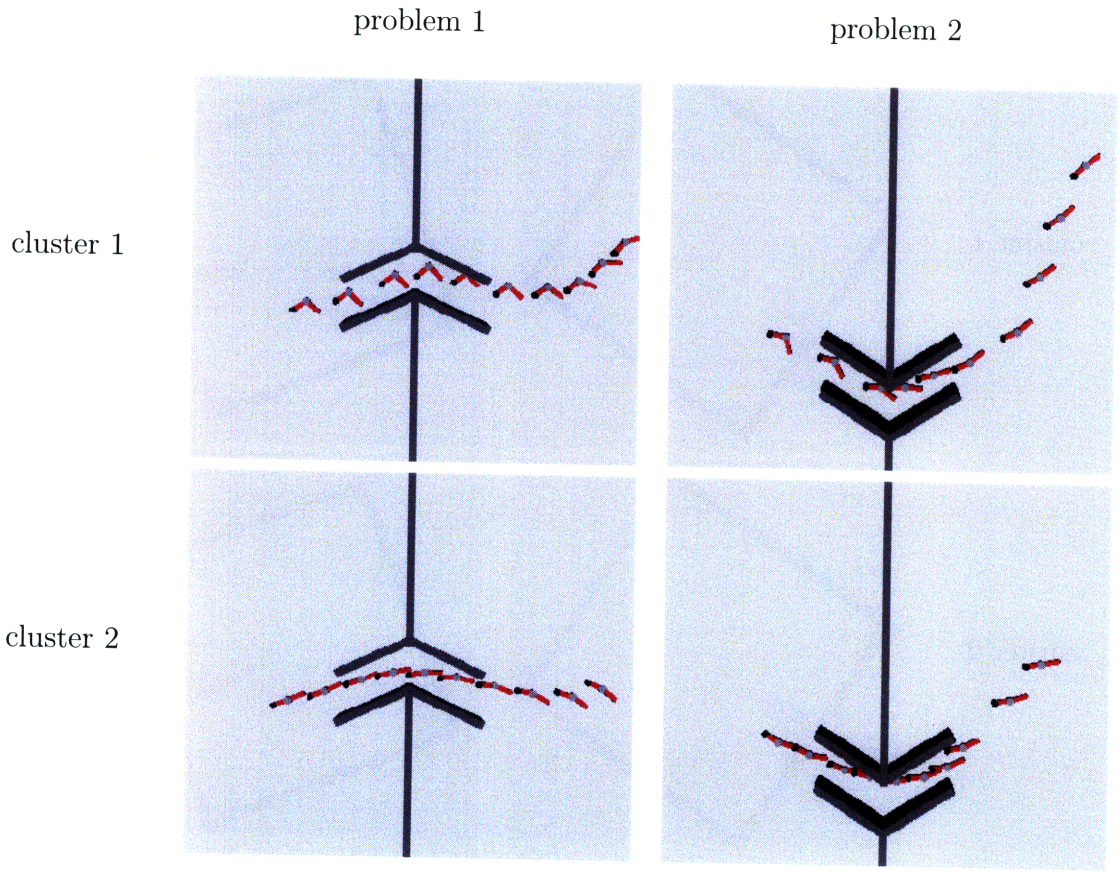


Figure 4-10: Two different strategies for solving the low degree-of-freedom angle corridor domain applied to two different problem instances. In this case, both solutions are non-colliding in both problem instantiations.

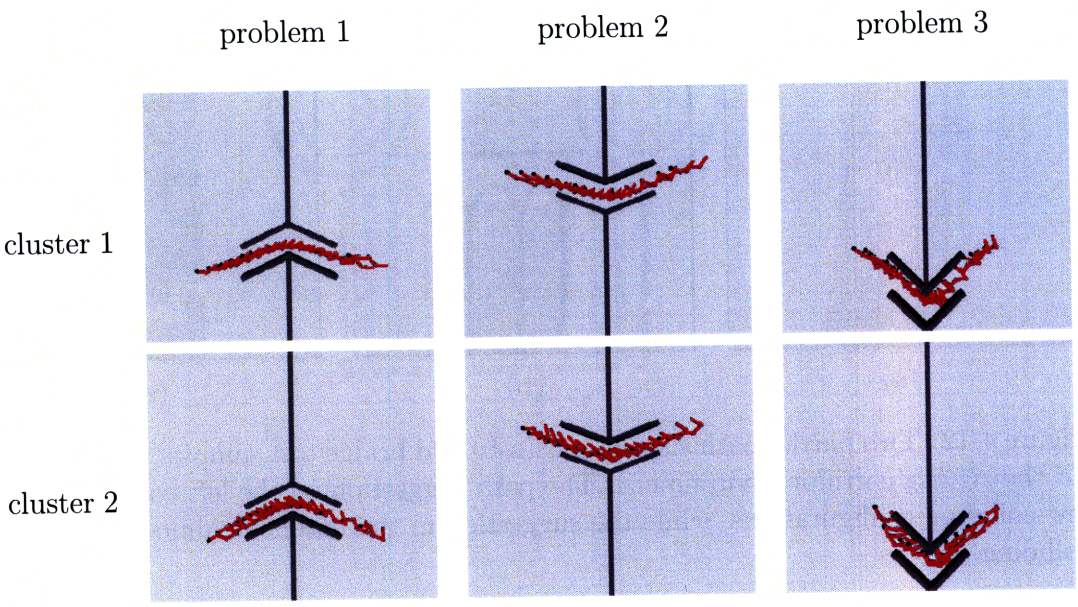
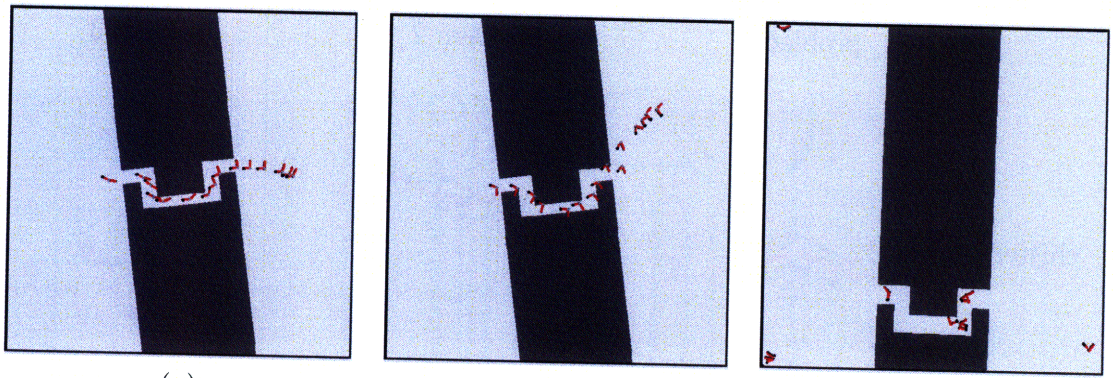


Figure 4-11: Suggestions generated by two strategies for three problem instances.

removed that had to be filled in by the planner, lessening the overall speedup. Figure 4-11 shows two strategies applied to three different problem instances. The first two problem instances are solved by the suggestions, but the third is not solved completely by any of the suggestions. When the colliding configurations are removed, these paths will have a gap in the middle that must be filled in by the planner.

In the zig zag environment, the suggestions were almost never completely collision-free, thus the substantial reduction in speedup relative to the other domains. Figure 4-12(a-b) shows a useful and a less useful suggestion. In the first case, there are two places where the removal of colliding configurations will require the planner to patch the path. In the second case, while the path is roughly the right shape, almost the entire path is in collision, meaning that it will not help the planner at all. Predictions such as these suggest that a fast way to adapt nearly-correct solutions should produce better results.

It is easy to see why the bridge test again produced less useful samples than the partial paths suggestion. An example of the bridge test samples is shown in figure 4-12(c). The bridge test again produces samples in the narrow passage that are difficult to one another, and is also distracted by irrelevant corners.



(a) (b) (c)

Figure 4-12: Two partial path suggestions (a-b) and bridge test samples (c) generated for the zig zag corridor environment. The path suggestion on the left contains only a few colliding configurations, while the suggestion in the middle is almost completely colliding.

4.4.4 Simple gripper

In the gripper case, we find that the suggestions always contain a few colliding configurations, but they are helpful enough to the planner that the time is cut by a factor of two. The fact that the suggestions provide greater speedup than the bridge samples suggests that it is useful to generate even a few sequences of samples that have high likelihood of having collision-free connections, relative to individual samples. Figure 4-13 shows a subpath suggested by our algorithm. Figure 4-14 shows examples of a useful sample and a distracting sample, both generated by the bridge test.

4.5 Conclusions

Our experiments show that by predicting a subpath of configurations that have high likelihood of collision-free connections between them, we can substantially speed up the planner. The suggested path samples have higher utility to the planner, both because they are more likely to be connected to one another, and because they are focused in the particular narrow passage that the robot is required to go through to solve the planning problem. Obviously the comparison is somewhat unfair, as the bridge test has no access to our task parameters, but it is a valuable demonstration that such information can be used effectively if it is available.

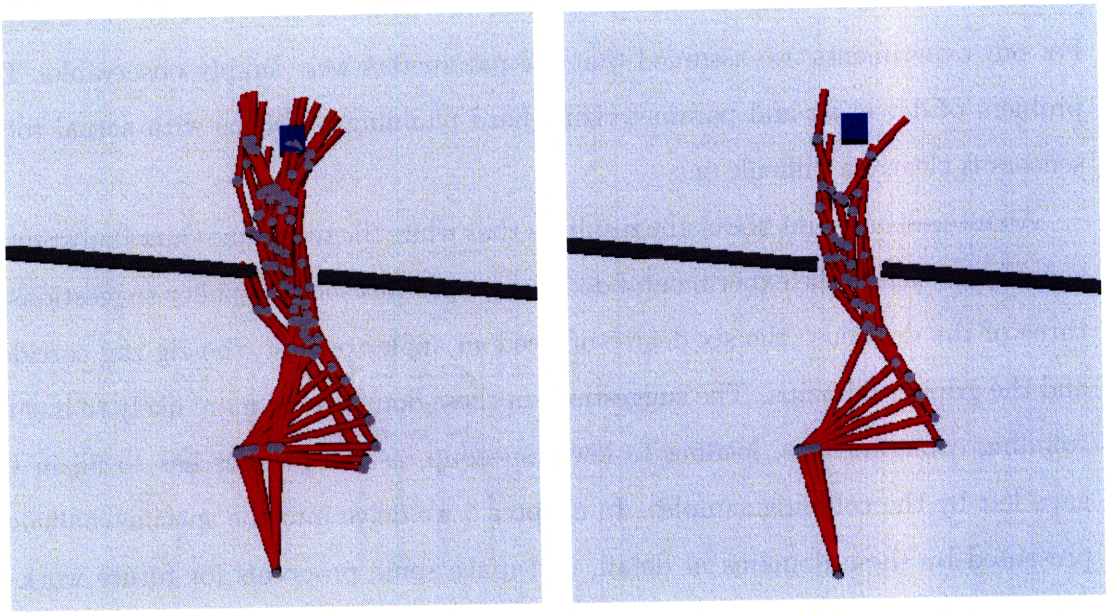


Figure 4-13: A relatively helpful gripper subpath suggestion. Left: full suggestion; right: suggestion with colliding configurations removed. Much of the path is in collision, but there are several key non-colliding configurations that are helpful to the planner.

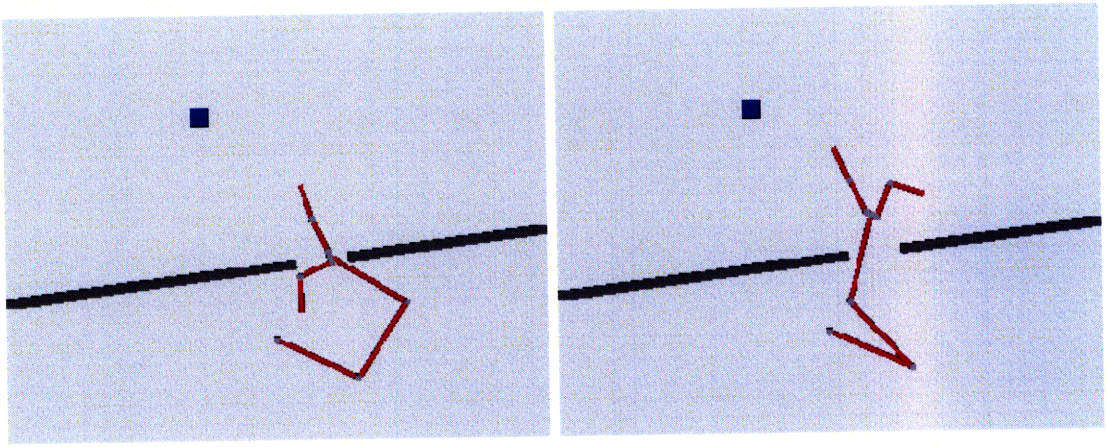


Figure 4-14: Two samples generated by the bridge test. The sample on the left is clearly not useful, while the one of the right is better.

One issue that would need to be addressed in order for this technique to be useful in more realistic settings is the question of determining the problem parameters. For our experiments, we assumed that the parameters were simply observable. The problem of detecting and parameterizing hard planning problems with actual robot sensors is clearly a difficult one.

An interesting point about the results is that while the technique shows substantial speedup in each of the experimental domains, it produces lower quality suggestions in three of the domains: the six-degree-of-freedom angle corridor, the zig zag corridor, and the gripper domains. The suggestions in these domains are more likely to include colliding configurations, leading to lower speedup, as the planner has to fill in the gaps left by the colliding samples. In chapter 5 we delve into the specific challenges presented by these domains in detail, and make some proposals for future work to address them.

Chapter 5

Problem Characterization

While the technique described in the last chapter met with success on standard motion planning problems, it was less successful on the more interesting problems, and so seems unlikely to be applicable in more realistic scenarios. In particular, in the angle corridor (with higher degrees of freedome), the zig zag corridor, and the gripper domains, our function clustering algorithm generated solutions that, while somewhat useful to the planner, had relatively high probability of containing colliding configurations. In this chapter, we explore the problem more carefully to understand what makes it challenging for traditional machine learning techniques.

5.1 Function existence

Our goal is to learn at least one function from the space of planning problem parameters (which we will call the *input space*), to the space of useful partial paths (which we will call the *solution space*). Given a planning problem, a partial path is considered useful if it is collision free and allows the planner to solve the planning problem in at most a tenth of the planning time required by the planner alone. Let I be the input space, P be the space of partial paths, and $S \subseteq P$ be the solution space. Then for a function $f : I \rightarrow P$, we will say that f covers $I_f \subseteq I$ if for all $i \in I_f$, $f(i) \in S$. We will say that a set of functions $F = f_j$ covers the space if for every input $i \in I$, $f(i) \in S$ for some function $finF$.

The entire enterprise of learning a function from the problem parameters to a useful partial path relies on the assumption that at least one such function exists, and that our hypothesis class is sufficient for representing the function. The function clustering algorithm allows the input space to be split up, with each function potentially covering a different subset of the input space, but all parts of the input space must be covered by some function. The first possible explanation for our algorithm's difficulty in some of our experimental domains is that there is, in fact, no such function. Further, if such a function does exist, it may be too complex to be represented by a function within our hypothesis class.

5.1.1 Zigzag corridor

In all of the domains that involve the mobile arm robot, as well as the zigzag corridor domain, there is a rigid transform that will map any solution for a particular input into a good solution for any other input, which implies a linear function over the input space, given the right parameterization of the problem.

Our zig zag corridor has two degrees of freedom, the vertical location of the corridor, y_c, and the angle of the corridor, θ_c. Consider a successful path $p = c_1, c_2, ...c_m$ that was generated with $y_c = 0$ and $\theta_c = 0$. Let c_j be one configuration in the path. For our mobile robot, this configuration would be $\langle x_r, y_r, \phi_0, \phi_1, ...\phi_{d-2} \rangle$, where d is the number of degrees of freedom. Now we have a new input in which y_c and θ_c are nonzero. Let $r = \sqrt{x_r^2 + y_r^2}$. The transformed configuration $c'_j = \langle x'_r, y'_r, \phi'_0, \phi'_1, ..., \phi'_m \rangle$ for the new input is:

$$\begin{aligned} x'_r &= r\cos\theta_c \\ y'_r &= r\sin\theta_c + y_c \\ \phi'_0 &= \phi_0 + \theta_c \\ \phi'_1, ..., \phi'_m &= \phi_1, ..., \phi_m \end{aligned}$$

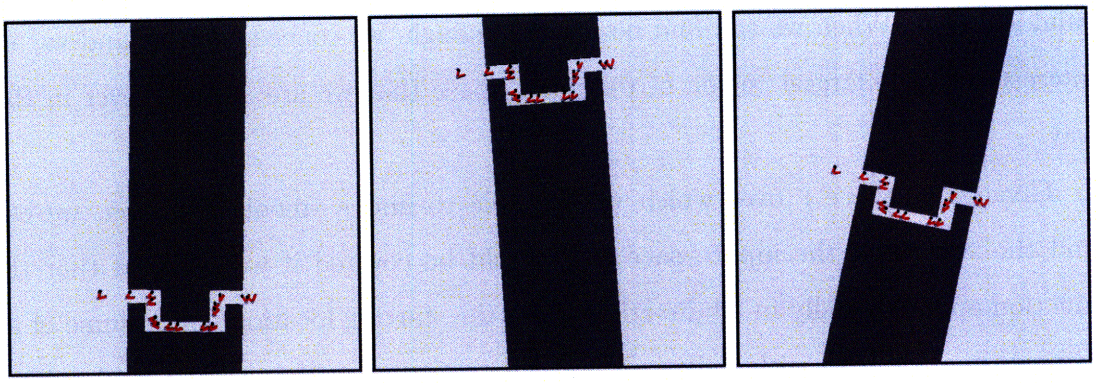


Figure 5-1: Partial paths generated from a linear function of the input space, for several different inputs.

.

Thus, with a parameterization that includes $\langle y_c, \theta_c, \sin\theta_c, \cos\theta_c \rangle$, there is a linear function of the parameters that determines a useful path. Figure 5-1 shows paths generated from a linear function of the input, applied to several different inputs.

5.1.2 Simple gripper

The simple gripper domain is much harder to analyse. Given our parameterization of the planning problems, it is certainly the case that no linear function of the parameters will produce useful paths over the whole input space. However, one might hope that several linear functions might represent solutions to parts of the input space, covering the whole space together. More likely, a polynomial function of the input space, or several of them, might be able to represent a good function. The very reason for trying to learn such a function is that this is a difficult function to write by hand.

However, there is some evidence that a relatively small number of continuous functions could cover the space effectively. If we change the planning problem only slightly, we can use potential fields to push a nearby solution into the space of valid solutions for the new planning problem. We can require that the change in input ϵ from the original input i_1 to the new input i_2 be small. Small enough, in fact, that using potential fields to adjust the original solution s_1 results in a new solution s_2, such that linearly interpolating s_1 and s_2 over the region $[i_1, i_2]$ always yields a

valid solution. When we can find no ϵ small enough, we concede defeat, and we are interested in the largest region of the input space that we are able to cover in this way.

Given a distance r over which we are able to find a smooth function, we can find the volume of the input space that would be covered if we assumed that the function applies equally in all directions from the starting location (the volume of an n-dimensional sphere with radius r), and compare that volume to the total volume of the input space. Using this back-of-the-envelope calculation, we find evidence that with about 8 functions, we should be able to cover the full input space in the 6-DOF angle corridor domain. This calculation suggests that the gripper domain may require more like 100 functions to cover the whole space.

The questions of whether a small number of smooth functions could be expected to cover the entire input space is still an open one. Given that we have not been able to find a small set of functions that cover the space, we cannot say with certainty that one exists. However, intuitively, the gripper is designed to be able to manipulate (in 2-dimensional space) effectively over its workspace, so we expect that we can move smoothly from one solution to another.

5.2 Function clustering complexity

Since we believe that a continuous function of sufficient simplicity exists, the difficulty must lie in clustering the data into discrete functions. Here we investigate the complexity of this problem in the simple zigzag corridor case. We begin by considering the solution space for a fixed planning problem.

For a fixed input i, let S_i be the solution space for that input; that is, the set of useful partial paths for all planning problems described by i. Two solutions $s_i, s_i' \in S_i$ are connected if there exists a continuous curve between the solutions that is contained entirely in S_i. Our solution paths are represented as points in a high-dimensional space with dl dimensions where d is the number of the robot's degrees of freedom, and l is the number of configurations in the path. Figure 5-2 shows a cartoon solution space

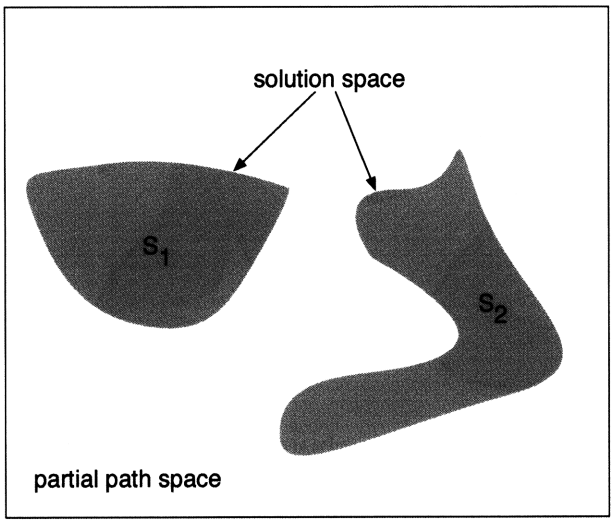


Figure 5-2: Example solution space. The solution space is made up of two disconnected regions, S_1 and S_2, each of which is path-connected. The region S_1 is convex, whereas S_2 is not.

for a two-dimensional path, which could be either a path with two configurations for a one-degree-of-freedom robot, or a path with one configuration for a two-degree-of-freedom robot. Any two solutions $s_1, s_1' \in S_1$ or $s_2, s_2' \in S_2$ are connected. However, any two solutions $s_1 \in S_1$ and $s_2 \in S_2$ are not connected, since any path between the two solutions would have to leave S.

Let's consider the ramifications of a non-convex solutions space for a clustering algorithm. Figure 5-3 (left) shows a non-convex solution space that has been clustered into three regions. One of the regions, R_2 happens to be convex but regions R_1 and R_3 are non-convex.

Now consider what happens when we sample from each cluster, and find the means of our samples. Figure 5-3 (middle, right) shows some possible samples, along with the average of the samples. Clearly it is possible, giving a lucky set of samples, to produce an average which is within the region, but it is not guaranteed. In fact, unless the centroid of the region falls within the region, in the limit of infinite samples, the average is increasingly likely to fall outside of the region.

Our clustering technique is trying to cluster our solution samples such that the mean of each cluster is a also a valid solution. While a clustering such as the one shown in the middle of figure 5-3 would be acceptable, given that particular data set, we can only guarantee that all the data in a cluster can be averaged if each cluster

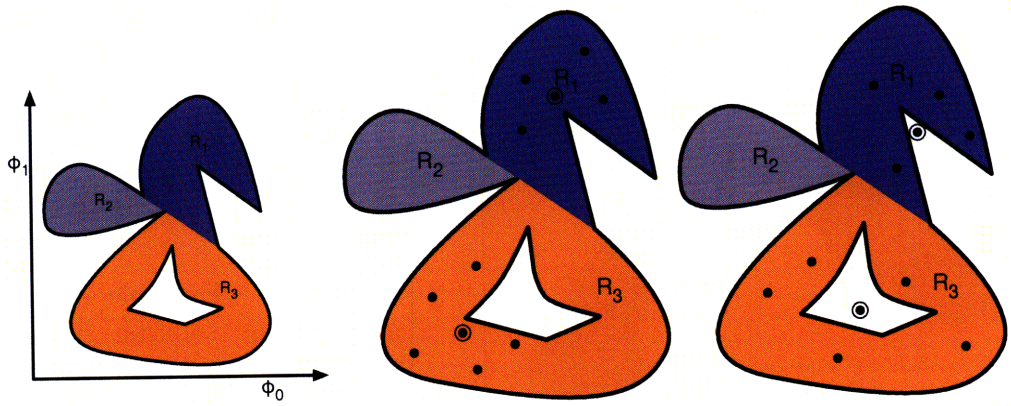


Figure 5-3: A non-convex solution space that has been clustered into three regions (left). Region R_2 is convex, while regions R_1 and R_3 are not. The middle and right show two sets of samples from the solution space, along with the mean of each cluster (circled). In the middle is a lucky set of samples, for which the mean in both non-convex regions falls in the solution space. On the right is a less fortunate set of samples. In this case the average is not a valid solution for this input.

is convex. The quality of a particular clustering is higher the more likely it is that a random sample of points from that cluster will have a mean that is also in the solution space. Thus, the best clustering is one that divides the space into convex regions. If the space is divided up in this way, we know that we can sample our data however we like, average those data points that fall into the same cluster, and each cluster will produce a valid solution. We define our clustering problem to be the problem of finding a (possibly soft) assignment of our data to groups, such that all points in each group are in one convex region of the solution space.

In general there are many ways to divide a non-convex space S into a convex covering, or a set of convex regions C_i such that $\bigcup\{C_i\} = S$. Figure 5-4 shows two ways to break a non-convex space into convex regions. Note that the data points could belong to more than one cluster (that is, the clusters may be overlapping); the important thing is that each cluster is convex. Among the various possible convex coverings, we prefer one with fewer convex sets, for reasons that we will discuss. Finding the minimal convex covering for a polygon is known to be NP-hard (Culberson & Reckhow, 1994). In our case the problem is still harder; the solution space is not known to be polygonal, and we have no description for the solution space, just the

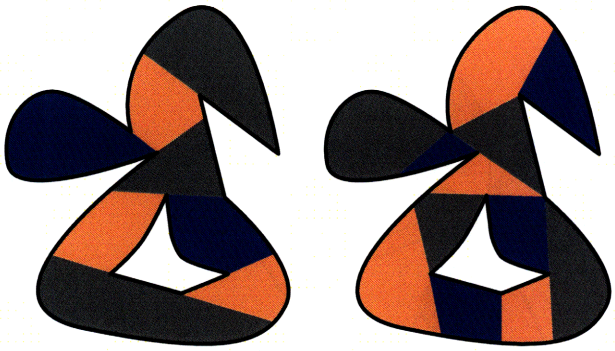


Figure 5-4: Two possible convex coverings for a non-convex space.

ability to sample from it.

5.2.1 Zigzag corridor

Now let's consider what happens to the solution space as it varies over the problem parameter space. We start by investigating the zigzag corridor domain. As we discussed earlier, the zigzag domain has the property that any solution s_1 for an input i_1 can be mapped to a solution s_2 for input i_2. Thus, for a set of solutions S_1 for input i_1, there is a linear function that maps S_1 into a set of solutions for i_2. It can be easily shown that if $C_1 \subseteq S_1$ is a convex subset of S_1, then the function mapping points from S_1 to S_2 induces a convex subset $C_2 \subseteq S_2$.

Let n be the dimensionality of the input space, and dl be the dimensionality of the solution space, where d is the number of the robot's degrees of freedom and l is the length of the subpaths we are learning. Given a convex solution set for a particular input i, C_i, any solution $s \in C_i$ can be mapped linearly to a solution for any other input. This means that the space of $C_i \times I$ is also a convex space. For a two-dimensional configuration space, one can visualize this as a linear extrusion of a convex space, as illustrated by figure 5-5

Since a convex solution set for a fixed input is a cross-section of a convex space over the input space, we still have the same criterion for a good cluster: all the points in one cluster should be in a convex set of solutions. In the zigzag domain, we are looking for a set of linear functions that cover the input space. Here we assume that we assume that we don't already know the linear function, as ask how much data

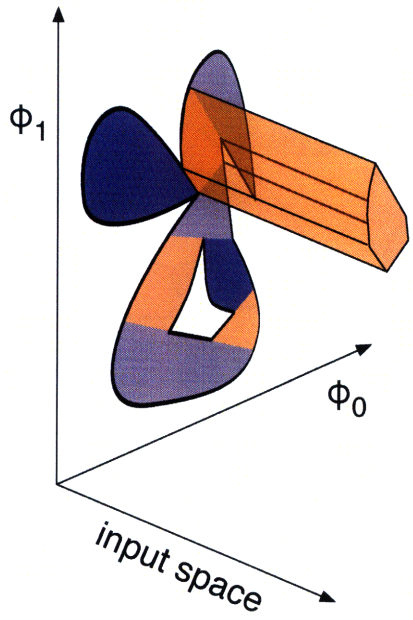


Figure 5-5: A convex solution set as it varies over the input space, given the linear mapping from a solution for one input to a solution at another.

we need in order to guarantee that we have enough to find such a function, and how hard the clustering problem is once we have the data.

Since the input space has dimensionality n, we need $n + 1$ points to define one function. We can choose any $n + 1$ points in a convex region, and this will define a function that is valid over the convex hull of the inputs. If we have a convex cover for a fixed input, each convex region in the cover, along with our linear mapping from one solution to another, defines a convex space. There is some minimal convex covering for the entire $I \times S$ space that requires k convex regions. Thus there is a minimal convex covering for the whole space with k convex regions. If we sample n points from one of these convex regions, that defines a function that covers the space between the sample inputs.

Thus, with carefully chosen inputs, and sampled solutions, we require just $n + 1$ points that are all sampled from one of our k convex regions. To ensure that we have at least that many points in at least one convex region, we need $nk + 1$ points. If we try to find a good function with fewer data points, we may be forced to average incompatible points together in every cluster, leading to no good functions. Given that we have $nk + 1$ points, we now have to determine which points should determine

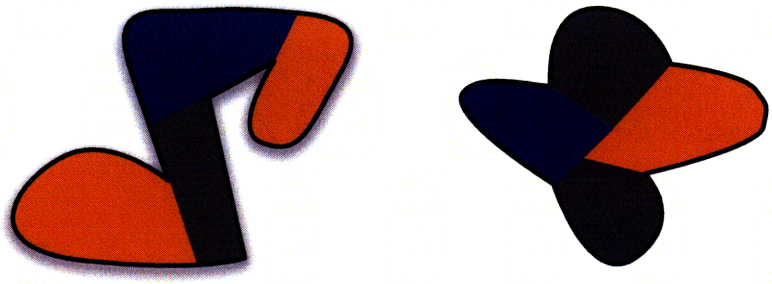


Figure 5-6: Two solutions spaces (for fixed input) with the same number of regions in the minimal convex covering, but different convexity. If any subset of regions on the right were grouped together as one cluster, it remains likely that points sampled from that cluster would have a valid mean. On the left, however, it is far more likely that incorrect clustering would lead to clusters with invalid means.

our function. There are $\frac{(kn+1)!}{(n+1)!(kn-n)!}$ possible subsets of $n+1$ points to consider.

For the domains in which our function clustering algorithm was successful, the input space had dimensionality 4 (doorway domain), 3 (angle corridor domain), and 2 (bug trap domain). The zigzag corridor similarly has input dimension 4. Thus, the increased difficulty of the zigzag domain must be attributable to the highly non-convex shape of the solution space. We do not know the minimum number of convex regions reguired to cover the solution space in each domain, but we can estimate a commonly accepted metric for convexity: the probability that two randomly chosen solutions are visible to one another.

The notion of convexity as a metric quantity is illustrated in figure 5-6. Here we see two spaces, each of which has the same number of regions in the minimal convex cover (4), but the space on the right is far more likely to produce clusters with valid means, even if the clustering is not strictly convex. Thus, we would expect the space on the right to be easier to cluster well enough, even if not perfectly.

If we measure the convexity of each domain by estimating the probability that two randomly chosen solutions are interpolable, we see in figure 5-7 that the zig zag domain has much lower convexity than the other domains. In figure **??**, we can see experimentally that the non-convexity manifests itself in the ease with which solutions for a fixed input can be clustered by EM. We conclude that the non-convexity of the

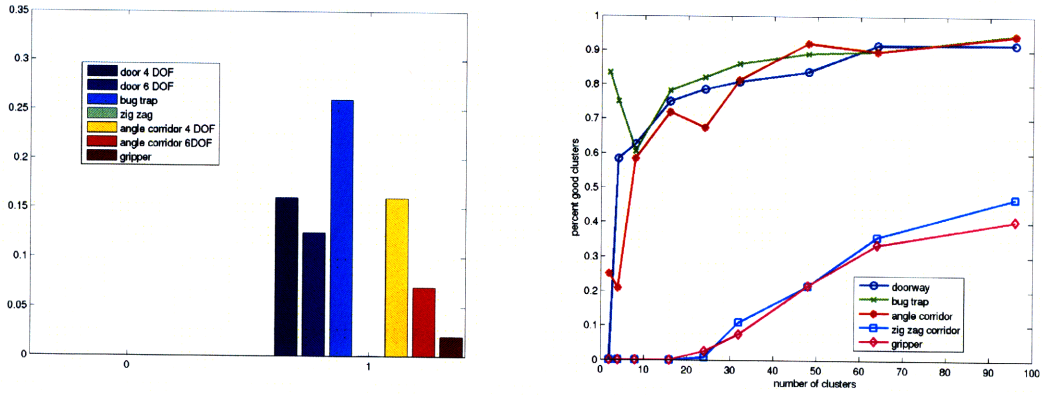


Figure 5-7: Convexity results. Left: the measured convexity for each domain. The convexity for the zig zag domain is markedly lower than the other domains, explaining the difficulty of the domain. Right: The probability that a cluster generates a valid prediction, as a function of the number of clusters used by EM to cluster solutions for one input (averaged over 6 inputs).

zig zag domain is main source of difficulty.

5.2.2 Simple gripper and angle corridor

The analysis of the previous section suggests that the zigzag corridor domain should be roughly the same difficulty as the simple gripper, but we find that, even with four times as much data, the function clustering algorithm is substantially less successful in the gripper domain. Similarly the higher degree-of-freedom angle corridor appear likely to be easier than the zig zag corridor, but performs about the same. Understanding the difficulty of the gripper and angle corridor cases is made more challenging by the fact that we do not know that a linear function exists that maps one solution to another. In fact, we know that such a function does not exist.

Let us consider how we might try to find a linear function of the input space that covers as much of the input space as possible. If we choose a function over the input space, the solution space around the function's prediction at each input is not necessary convex, even if we know that it is convex at one particular input. Figure 5-8 shows some example solutions spaces surrounding a linear function for a two dimensional solution space and a one-dimensional parameter space.

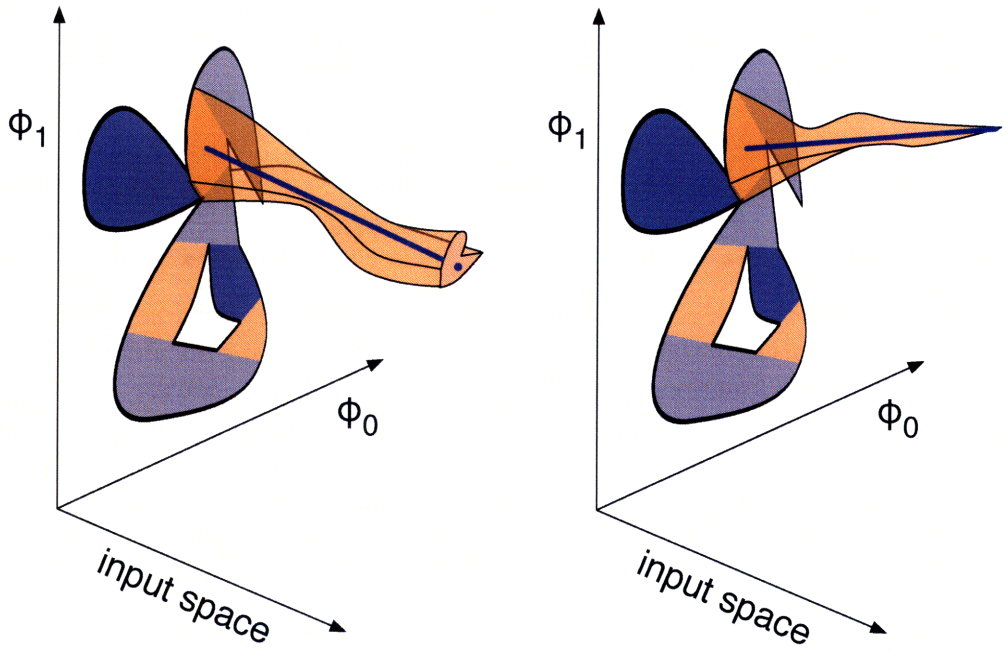


Figure 5-8: Two solution spaces produced by a linear function of the input space. On the left the function produces valid solutions over the whole input range, but the distance between the function's prediction and the edge of the solution space varies over the input range. The solution space for a fixed input surrounding the function is not necessarily convex. On the right is a linear function where the solution space dwindles to nothing, so the function does not cover the whole input range.

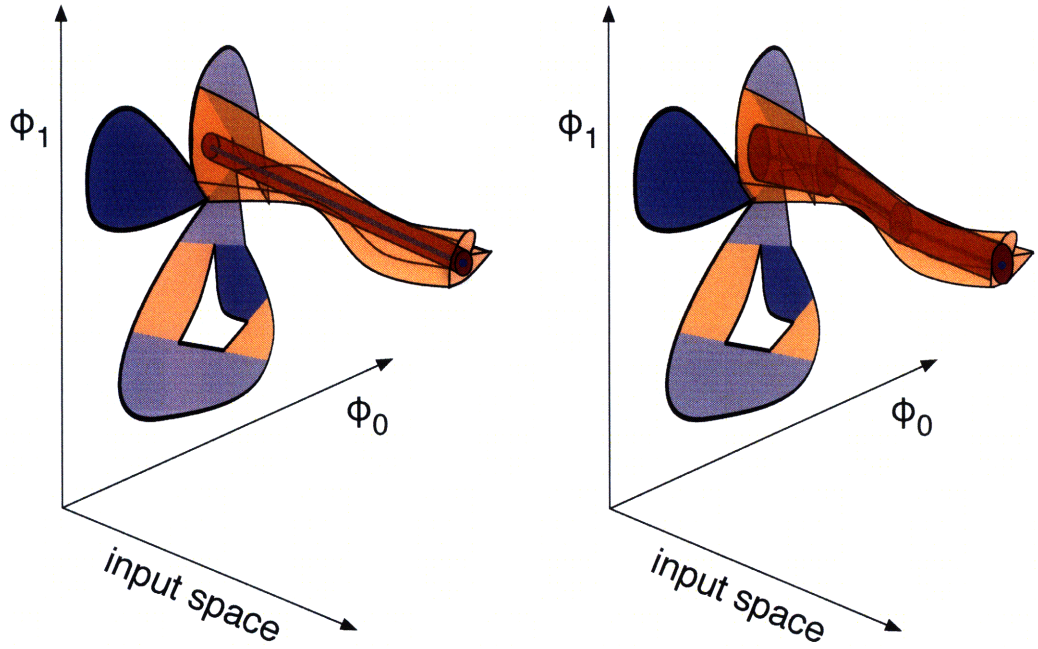


Figure 5-9: An illustration of the largest convex region surrounding one linear function (left) or multiple linear functions (right) used to cover the input space.

We can still use the criterion we developed previously for a good clustering: all points assigned to a cluster are from a convex region of the $I \times S$ space. However, since we now have no guarantee that the edges of the space are linear, we no longer know that we can use the size of the minimal convex cover to guarantee that we have enough points to define at least one good function. The size of the convex region that surrounds the function is now limited by the size of the smallest convex region that the function passes through. Figure 5-9(left) illustrates this idea. Thus, we will need to sample more points to ensure that we have enough from a particular convex region to define a function. We could intead choose to approximate the function with several linear segments in the hopes that each one will be contained in a larger convex region, and therefore be easier to sample from (as illustrated in figure 5-9(right), but this again increases the number of samples that we need, since it increases the number of functions that we need to define.

5.3 Possible approaches

While we have argued that the function clustering technique is likely to fail on complex problems due to the intrinsic difficulty of the problem, what about another technique? We have reason to believe that a small set of contiunous functions should be sufficient for predicting solutions over the whole input space, so it seems reasonable to think that there might be some other way to find such a set of functions. Here we discuss some of the standard machine learning techniques that appear most relevant to our problem.

5.3.1 Finding a nicer space

One idea is to find a space in which our problem is easier. One common way to transform one space into another is via dimensionality reduction. Not only might we then have a lower dimensional space, which we've shown to be important to the complexity of the problem, but we might have a space in which our solution space is more convex, making an approximate solution more acceptable.

Unfortunately, we can't do dimensionality reduction on the full $I \times S$ space, since we need to represent a function from I to S. Imagine that we have been able to find a new space A by reducing the dimensionality of $I \times S$. Now we need to learn a function that maps some subspace A_I of A to another subspace A_S of A, and we need to be able to map an input in our original space $i \in I$ to a point in A_I and then map the resulting point in A_S back to a point $s \in S$. Thus the complete space is not an appropriate target for dimensionality reduction.

We might instead consider reducing the dimensionality of our solution space, giving us a new output space O for our function. We can then learn a function $f : I \rightarrow O$ and then map each output point back to the original space. This means that we need to be able go backwards from the new space to our original space, something that most dimensionality reduction algorithms are not designed to do. Locally linear embedding ((Roweis & Saul, 2000)) and ISOMAP ((Tenenbaum, de Silva, & Langford, 2000)), for example, produce embeddings from which the original data can be recon-

structed, but the reconstruction requires finding the nearest neighbors of a point to be mapped back to the original space. Since this step takes place at planning time for us, the process of un-embedding the data is too slow.

Another possibility is principle components analysis. Limited experiments with our data indicated that reconstructing the original data with a subset of the components found by PCA was insufficient: paths that originally did not collide were mapped to colliding paths when components were not used in reconstruction. Further, since PCA simply scales and rotates the original space, it is unlikely to be able to make a highly non-convex space more convex, which is what we need in order to make our problem more tractable.

5.3.2 Classification

Another attribute of our problem is that it is possible to generate bad data (non-useful or colliding partial paths) as well as data from the solution space. We might imagine that using the bad data in some way would allow us to better determine which points belong together. If we have a data set with labelled good and bad points, we could use the substantial body of work which has been done on the problem of classification.

However we still need, in the end, to generate solutions as a function of the input parameters, so our problem is not strictly one of classification alone. While we have labels for whether each point is good or bad, we don't have any information about which good points can be safely used together to determine a function. Nonethless, we might expect that we could find separators in the full $I \times S$ space, and then use the data in each separated regions to do regression. We could use a decision tree to find multiple separators, or we could use a class of separators that is capable of separating our data with just one separator, such as an SVM with a radial basis kernel. Figure 5-10 shows how this process might go.

However, to arrive at the useful separators in figure 5-10, we would need to have just the right class of separators. Figure 5-11 shows what several classfication techniques would likely produce on the same data shown in figure 5-10. Each of the classification techniques is able to perfectly separate the data, but leaves us with no

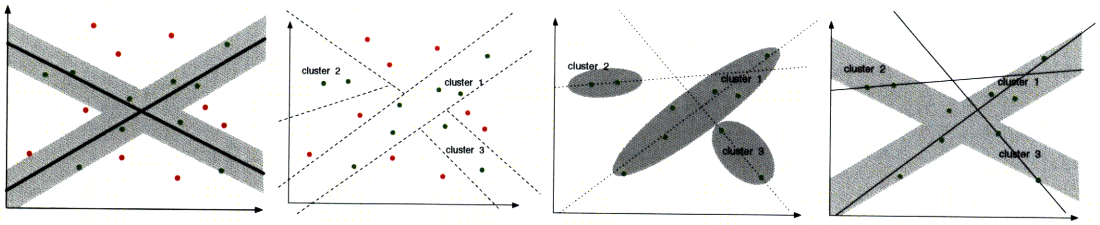


Figure 5-10: An example of how a classification algorithm might find separators that allow us to cluster our data and then use regression to find a function for each cluster. On the left is the original data (good and bad data points in green and red. respectively), along with the solution space (in grey) and two functions which would be valid over the whole input space. Middle left: set of linear separators that might be found by a classification algorithm. Middle right: the clustering of good data induced by the separators. Far right: The predictions of each cluster, along with the original solution space. Clearly the function associated with cluster 1 is good enough, which those associated with clusters 2 and 3 are not good over a substantial part of the input space.

useful distinctions between the different types of good data.

5.3.3 Generating better data

The source of difficulty for our learning algorithm seems to be the large number of incompatible solutions that must be teased apart when the set of training solutions is generated more or less randomly from the space of solutions. However, we would be content to learn just one or a few strategies for solving each parameterized problem. A natural question, then, is whether it would be possible to generate training data that is drawn from just one or a few solution types. This should allow us to generate much less data, and make our function clustering job much easier.

The challenge in trying to generate data that is consistent with a few strategies is that we don't know the underlying function for any of the strategies: this is precisely the problem we are trying to solve. Nor do we know a way, given an instance of particular strategy at input i, to generate another instance of that same strategy at a new input i'. Nonetheless, if we could generate data that represents fewer underlying strategies, we might hope to improve the performance of our function clustering algorithm. Here we explore several ideas for achieving this.

One possibility is to bias the planner that we use to generate data so that it

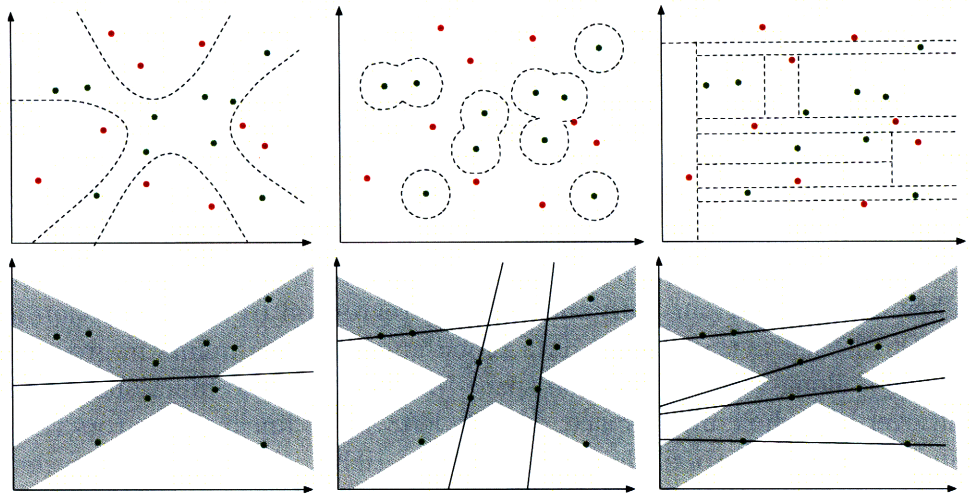


Figure 5-11: Several perfect classifications of the data (top: separators; bottom: predictions and solution space), each of which leads to a bad clustering from the perspective of our regression task. Left: quadratic separators. Middle: radial basis function separator. Right: decision tree separator.

will generate data that is less varied. Or, given plans generated by the unbiased planner, we could post-process the plans to canonicalize them in some way. We could perhaps optimize some quality of the paths, such as the distance of each configuration from a preferred configuration, or the total distance (possibly in configuration space) travelled by the path. We did not explore this option in depth. It is challenging, in the general case, to arrive at a mechanism for biasing the planner that does not interfere with its ability to find solutions to hard problems. Post-processing paths seems more feasible, but there is an open question about what we could optimize that would make our learning problem easier. It is easy to imagine an optimization that would create data that exemplify fewer underlying strategies, but where each strategy is difficult to represent.

Another possibility is to generate just a few solutions and then use heuristics to adjust the solutions to new inputs. We might hope that the set of solutions generated from adapting one solution might all fall into the same category of solution, and therefore be suitable for using as function approximation training data.

It is possible to use potential fields **??** to adapt a solution s which is valid for input

i to a nearby input $i + \epsilon$ when ϵ is fairly small. A possible approach is to start with a small number of solutions, and, by taking small steps in the input and adapting the solutions appropriately, generate a data set consisting of compatible paths. We found that using potential fields was not a promising direction. If ϵ was very small, then the potential field technique was able to successfully adapt the solutions, but progress was made over the input space so slowly that we were not able to cover the 5-dimensional input space of the gripper. If ϵ is increased to make faster progress, the potential field method either fails to adapt the solutions, or has to alter the solutions so much that they are no longer compatible.

5.3.4 Incremental learning

Another idea is to use incremental learning to decrease the total number of types of solutions generated and learned. Pseudocode for this idea is shown in algorithm 5.3.4. The idea is to generate solutions randomly by calling the planner only when we are unable to either predict a solution or adapt a prediction, again using potential fields.

Algorithm 5.1 INCREMENTALLEARN(*batchSize*, *numBatches*) : *predictor*

```
1: predictor = 0
2: data = []
3: for i from 1 upto numBatches do
4:   for j from 1 upto batchSize do
5:     problem = NewPlanningProblem()
6:     for k from 1 upto NumPredictions(predictor) do
7:       prediction = Predict(predictor, k)
8:       if goodSolution(prediction) then
9:         data.append(problem, prediction)
10:      end if
11:    end for
12:  end for
13:  predictor = makeTrainPredictor(data)
14: end for
15: return predictor
```

We found that this algorithm was also unable to gain traction on the problem. Figure 5-12 illustrates why the techniqe is unsuccessful. When we have only a few randomly generated samples for training, we are most likely to learn a function based

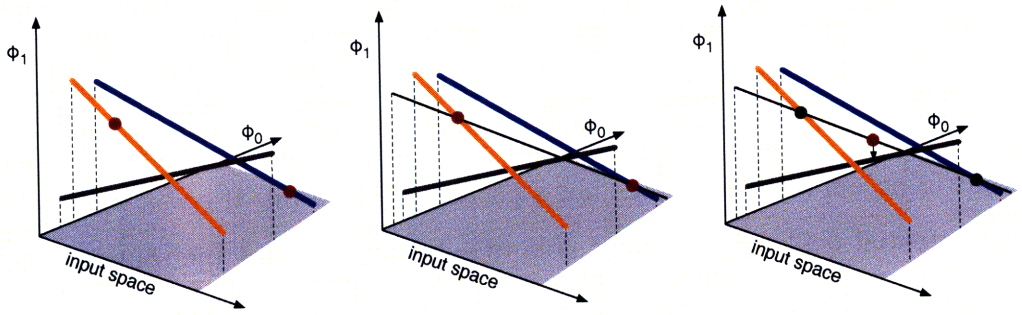


Figure 5-12: An illustration of an incremental learner. In each, the thick lines represent the solution space. Left: some number of samples are randomly generated. Middle: the learner is trained on the existing data. Right: proposed solutions are generated according to the learned function, and then adapted. The illustration shows the problem with such a technique.

on points in a non-convex region of the solution space, meaning that the function predicts invalid solutions over much of the input space. When we adapt these predictions, we have guarantee, or even reason to believe, that the adapted predictions will lie within convex solution regions that contain points we already have.

5.3.5 Function parameter search

Another idea is to simply search in the space of function parameters for a function that adequately covers the input space. If we use the number of correct predictions as a scoring metric, we could hope to do stochastic gradient descent in the space of regression parameters.

The space of parameters is quite large, as the partial paths typically have length 15 and the platforms have at least four degrees of freedom. Thus, with a 4-dimensional input space, we have 240 parameters. Clearly starting with a random set of parameters and taking steps in parameter space is unlikely to be successful. Instead, we can take steps in parameter space by adding, removing, and altering points in a regression training set. That is, let the data set determine the parameters and take steps through the parameter space by changing the data set.

In our implementation, we selected initial parameters by using potential fields applied to one solution to generate similar solutions to nearby problems. Once we

had initial parameters, we considered several search step, and used a scoring metric to evaluate the steps. Each search step is a change to the data set which will generate new regression parameters. The score for a particular setting of the parameters is simply the success rate on a pre-determined set of test inputs.

We consider several possible search steps. For a set of training points $T = \{\langle i_j, s_j \rangle\}$, where i_j is a particular input and s_j is a valid solution at input i_j, regression determines parameters that define a function f across the whole input space. One possible search step is to simply remove a point from T. Another simple step would be to replace the point $\langle i_j, s_j \rangle$ with the current prediction at that input: $\langle i_j, f(i_j) \rangle$.

More complicated steps involve adding new points to the training set. Let $p(i, s)$ be a the result of applying potential fields to a partial path s, giving a new partial path s' that is valid for input i. This function will only produce valid solutions of the original solution s is a valid solution to some input i_j that is close to i. One possible search step is to choose a new input i_{new} that is not currently represented in T, find a nearby point $\langle i_j, s_j \rangle \in T$, and add the point $\langle i_{new}, p(i_{new}, s_j) \rangle$ to T. We can also adapt the current prediction using the regression parameters, instead of a nearby point in the training set: $\langle i_{new}, p(f(i_{new})) \rangle$. Each of these potential search steps is scored, and then the step with the highest score is taken.

We applied this approach primarily to the zig zag domain, since we were confident of the existance of a linear function that covers the entire input space. Our experiments with this technique confirmed that the space of parameters contains many local optima. Initially we used greedy search, taking only steps that improved the score. This approach got stuck fairly quickly, with parameters that we good very locally, but failed to generalize across the input space at all. Using simulated annealing we were able to find parameters that covered a larger region of the input space, but we were still not able to find parameters for a single linear function over the whole space, despite the fact that they are known to exist.

5.4 Summary

From the investigations in this chapter, we conclude that the problem of learning functions from problem descriptions to useful motion plans is made difficult by the very large number of possible solutions for each input. Even in cases where simple functions are known to exist, it is likely to be intractible to completely represent the space of solutions, particularly in more realistic domains. It will be necessary to find a way to limit the set of solutions we use as training data for our function approximation algorithm.

We further conclude that the problem of learning motion plans involves challenges that are different from those addressed by traditional machine learning techniques. The next chapter suggests some techniques that we believe are more likely to be helpful in continuing work on this topic.

Chapter 6

Conclusions and future work

In this thesis, we have investigated the feasibility of learning to predict motion plans, using plans generated by a probabilistic planner as training data. We first investigated simply remembering previous solutions in a roadmap designed to adapt to the distribution of planning problems the agent encountered. While this technique provided an improvement over planning from scratch in domains with very regular distributions of obstacles and tasks, we found that it was not able to generalize to new tasks sufficiently well.

We next proposed an approach for generalizing previous planning experience by using that experience as data for learning a regression from some task parameters to a partial path that provides significant leverage to a single-query planner. Here we found that while the number of useful partial paths is small in comparison to the entire space of partial paths (and therefore hard to find), it is large tnough that it is infeasible to represent them all.

Since we do not require more than a few parameterized partial paths if each is applicable to a wide range of parameters, this discovery does not immediately exclude the possibility of learning a sufficient number of parameterized partial paths. However, without actually knowing the appropriate parameters for functions from problem parameters to useful partial paths, it is not possible to generate data that is guaranteed to be consistent with a small set of such functions. The primary result of this work is to conclude that learning parameterized motion plans requires finding a

way to represent only a compatible subset (or a few compatible subsets) of solutions.

6.1 Future work

We have demonstrated that learning to solve interesting planning problems requires limiting the portion of the solution space that is represented. Here we suggest some avenues for future work in this area that seem more likely to be successful, in light of our understanding of the problem. We first suggest two techniques that would make the approximation problem less exacting, and then two techniques designed to address the complexity issue discussed in the last chapter that is due to the many incompatible solution types in more complex domains.

6.1.1 Suggestion adaptation

The suggestions produced by our function clustering algorithm were often very close to being extremely useful to the planner, in that a small number of configuration were in collsion, and often the collisions were not very severe. This suggests that there might be a way to adapt the solutions at planning time so that they contain fewer colliding configurations. One possibility is to again investigate the use of potential fields to quickly adapt suggestions before passing them on to the planner. Preliminary investigations into this possible technique suggest that the potential field implementation would have to be carefully optimized in order to fix suggestions fast enough.

6.1.2 Obstacle or robot dilation (free space contraction)

One technique that has been used to speed up planning in problems with configuration-space narrow passages, is one of freespace dilation (Hsu et al., 2006). In this approach, the obstacles or the robot are shrunk, making the freespace more expansive and the planning problems easier. Once a plan has been found in this easier space, it is adapted to the real, more difficult space if possible.

We could apply the inverse operation for our problem: make the narrow passages even more narrow, and use data generated by solving the resulting more challenging problems as training data for our function approximator. This process would generate data with greater clearance from obstacles when the robot and/or obstacles were normal size, meaning that the function approximator would have more room for error in approximating the data set.

Preliminary experimentation with this technique showed some promise, but there is more work to be done before the technique can be generally applied. First, while the proposed approach would certainly increase the quality of the suggestions, in that they would be less likely to collide, it does not address the complexity problem of having too many possible solutions to represent. Also, it is not easy to know how much the free space can be contracted before the problem becomes too hard or impossible. Nonetheless, such a technique could be used in conjunction with some of the other proposed future work directions to make the approximation problem easier.

6.1.3 Canonicalization

One promising direction would be canonicalization of paths. Humans are highly redundant, and nonetheless manage to learn motion strategies for complex motions. While there is no consensus about what criterion humans use to select from the myriad possible solutions, it is clear that we do something reasonably consistent. While we have not yet investigated this possibility, it seems a fruitful direction for research. The difficulty is in choosing a method for canonicalizing that restricts the data enough to make the learning problem tractible, while still retaining all of the global properties that make each training point useful.

It might also be possible to bias the planning process toward some canonical set of solutions. Here the challenge is to restrict the data enough while striving not to make the planning problems themselves intractible or impossible. In the doorway task, for example, we could require the robot to go through the door with the arm in front of the robot, and that would eliminate the primary source of redundant solutions. However, if we make this restriction more generally, we may not be able to solve

planning problems in domains that perhaps require that the arm sometimes trail the robot. This is an area that warrants more research.

6.1.4 Problem decomposition

We have done some preliminary work on an algorithm designed to decompose the problem into individual waypoint configurations. Restricting the clustering problem to deal with one configuration at a time avoids the exponential blowup in the number of possible solutions, making the clustering tractible. The difficulty in solving subproblems locally is that the solution to each subproblem must contribute to the global goal of generating a subpath that is helpful to the planner. To address this, we still generate whole useful subpaths as training data, but then train each slot in the subpath independently.

Algorithm 6.1 shows pseudocode for the waypoint learning algorithm. For each step in the subpath, starting with the final configuration, training data is generated by solving planning problems and finding the narrow passage, as described in 4.1. The training data is then clustered and predictors are trained for each cluster. We then choose a set of learners that adequately cover the input space, and remove the rest, to avoid exponential blowup. Then this process is repeated for the previous step in the path. When the training data is collected, each training point is tested to see whether it connects to one of the retained predictors at the next step, and if not, it is discarded.

Once predictors for each step of the subpath are learned, we have a must more constrained planning problem. Pseudocode for solving this problem is shown in algorithm 6.2. For step i in the subpath, we need consider connections between all possible predictions between the previous $(i-1)$ step and this one. If we associate a cost with colliding connections between configurations, and a higher cost with colliding configurations, we can, at each step, find the lowest-cost connection. We then use the lowest-cost path as a suggestion for the planner.

Preliminary results with this technique in the higher degree-of-freedom angle and zig zag corridor domains indicate that the approach has promise. We find that while

Algorithm 6.1 LEARNWAYPOINTS(*length*) : *predictors*

```
1: predictors = []
2: for i from length downto 1 do
3:   data = []
4:   for j from 1 upto ptsPerCluster * numClusters do
5:     problem = NewPlanningProblem()
6:     solution = GetSolution(problem, length)
7:     if i == length or ConnectsTo(solution[i],predictors[0]) then
8:       data.push(problem, solution[i])
9:     end if
10:  end for
11:  clusters = ClusterData(data)
12:  predictor = TrainLearners(clusters)
13:  predictor = CullBadLearners(predictor)
14:  predictors.push(predictor)
15: end for
16: return predictors
```

Algorithm 6.2 PREDICTPATH(*predictors*) : *path*

```
1: paths = [[]]
2: for all predictor in predictors do
3:   newPaths = [[]]
4:   for i from 1 upto NumPredictions(predictor) do
5:     minCost = ∞
6:     waypoint = Predict(predictor, i)
7:     for all path in paths do
8:       p = path
9:       cost = PathCost(p, waypoint)
10:      if cost < minCost then
11:        minCost = cost
12:        if ConnectsTo(p, waypoint) then
13:          p.push(waypoint)
14:        end if
15:        bestp = p
16:      end if
17:    end for
18:    newPaths.push(bestp)
19:  end for
20:  paths = newPaths
21: end for
22: return LowestCostPath(paths)
```

the average planning time using the generated suggestions is slower than those using the full-path function cluster algorithm described in chapter 4, the median time is much lower. Using the new technique, perfect subpaths are predicted more than half of the time. However, in a number of cases, no useful prediction is generated. We hope to improve the technique so that it generates perfect solutions a higher percentage of the time, and so that it generates at least partially useful suggestions all of the time, as did the full subpath learning technique. On possible simple approach would be to combine the two techniques: the waypoint generation technique could propose very high quality suggestions when possible, and when it fails, the less-than-perfect but still useful suggestions from the full subpath learners could step in.

It is less clear that the technique will be successful in the simple gripper case, and this is the most interesting area for further exploration with this approach. When the set of predictors for a particular step in the subpath is chosen, commitments are made to solving the problem with a small set of strategies. Since this commitment is made using a local criterion, it is possible that the rest of the subpath will be impossible to learn. In the angle and zig zag corridor domains, this does not seem to be the case, but it remains to be seen whether or not this is a problem in the simple gripper domain.

6.2 Lessons Learned

Here we summarize the lessons learned during the course of this work. Since the work remains somewhat unfinished, we hope that continuing work in the area of learning motion plans will benefit from our investigations.

- Memorization of motion plans for later use in similar environments can be effective in low-entropy environments, but it is important to keep the set of stored motion plans as small as possible. However, memorization and reuse of motion plans does not generalize well enough to be applicable in more realistic domains.

- A more flexible approach involves learning a function from problem parameters

to useful subpaths that can be used by a probabilistic planner to find a solution more quickly. For significantly redundant robotic platforms, representing the whole space of useful partial paths is intractible. A learning technique for motion plans in this setting will need to find a way to restrict the solutions space so that learning is possible. We suggest some possible future avenues addressing this issue in section 6.1.

- Learning motion plans is a regression problem with the additional constraint that the full output space is too large to represent completely. This makes the problem fundamentally different from those addressed by apparently relevant existing machine learning techniques.

- To make progress in learning parameterized motion plans, the most important problem to solve seems to be the exponential blowup in the number of solutions that is caused by the need for motion plans to be compatible along the whole length of the path. Preliminary work suggests that a technique designed to learn each step of the path independently, but using a global usefulness criterion for data generation, may address this problem effectively.

Bibliography

Alami, R., Siméon, T., & Krishna, K. M. (2002). On the influence of sensor capacities and environment dynamics onto collision-free motion plans. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pp. 2395–2400.

Asfour, T., Azad, P., Vahrenkamp, N., Regenstein, K., Bierbaum, A., Welke, K., Scröeder, J., & Dillman, R. (2008). Toward humanoid manipulation in human-centred environments. *Robotics and Autonomous Systems, 56(1).*

Atkeson, C., Moore, A., & Schaal, S. (1997). Locally weighted learning. *AI Review, 11*, 11–73.

Bailey, T., & Durrant-Whyte, H. (2006). Simultaneous localization and mapping (slam): part ii. *Robotics and Automation Magazine, 13(3)*, 108–117.

Barraquand, J., Kavraki, L., Latombe, J.-C., Li, T.-Y., Motwani, R., & Raghavan, P. (1997). A random sampling scheme for path planning. *International Journal of Robotics Research, 16(6)*, 759–774.

Barraquand, J., Langois, B., & Latombe, J.-C. (1992). Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man, and Cybernetics, 22(2).*

Bekris, K., Chen, B., Ladd, A., Plaku, E., & Kavraki, L. (2003). Multiple query probabilistic roadmap planning using single query planning primitives. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pp. 656–661.

van den Berg, J. P., & Overmars, M. (2006). Planning the shortest safe path amidst unpredictably moving obstacles. In *Proc. Workshop on Algorithmic Foundations*

of Robotics (WAFR).

van den Berg, J. P., & Overmars, M. H. (2004). Roadmap-based motion planning in dynamic environments. Tech. rep. UU-CS-2004-020, Institute of Information and Computing Sciences, Utrecht University.

van den Berg, J. P., & Overmars, M. H. (2005). Using workspace information as a guide to non-uniform sampling in probabilistic roadmap planners. *Intl. J. of Robotics Research, 24(12)*, 1055–1071.

Bohlin, R., & Kavraki, L. (2000). Path planning using lazy prm. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 512–528.

Boor, V., Overmars, M. H., & van der Stappen, A. F. (1999). The Gaussian sampling strategy for probabilistic roadmap planners. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1018–1023.

Brandt, D. (2006). Comparison of a and RRT-connect motion planning techniques for self-reconfiguration planning. In *Proc. IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems*, pp. 892–897.

Burns, B., & Brock, O. (2005a). Sampling-based motion planning using predictive models. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 3120–3125.

Burns, B., & Brock, O. (2005b). Single-query entropy-guided path planning. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA).*

Burns, B., & Brock, O. (2005c). Toward optimal configuation space sampling. In *Prob. Robotics: Science and Systems (RSS)*, pp. 105–112.

Burns, B., & Brock, O. (2007). Sampling-based motion planning with sensing uncertainty. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA).*

Butz, M., Herbort, O., & Hoffmann, J. (2007). Exploiting redundancy for flexible behavior: Unsupervised learning in a modular sensorimotor control architecture. *Psychological Review, 114(4)*, 1015–1046.

Canny, J. (1988). *The Complexity Of Robot Motion Planning.* MIT Press.

Canny, J. (1989). On computability of fine motion plans. In *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA)*, pp. 177–182.

Caselli, S., & Reggiani, M. (2000). Erpp: An experience-based randomized path planner. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1002–1008.

Chakravorty, S., & Saha, R. (2007). Hierarchical motion planning under uncertainty. In *Proc. IEEE Conf. on Decision and Control.*

Chen, P. (1997). On intelligent motion planning via learning. *J. of Intelligent Robotic Systems, 19(3)*, 299–320.

Cheng, P., & Lavalle, S. (2002). Resolution compelte rapidly-exploring random trees. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 267–272.

Cooper, S., Hertzmann, A., & Popović, Z. (2007). Active learning for real-time motion controllers. *ACM Transactions on Graphics, 26(3).*

Culberson, J. C., & Reckhow, R. A. (1994). Covering polygons is hard. *Journal of Algorithms, 17(1)*, 2–44.

Donald, B. (1984). Motion planning with six degrees of freedom. Tech. rep. AI-TR-791, Massachusetts Institute of Technology Artificial Intelligence Laboratory.

D'Souza, A., Vijayakumar, S., & Schaal, S. (2001). Learning inverse kinematics. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS).*

Durrant-Whyte, H., & Bailey, T. (2006). Simultaneous localization and mapping (slam): part i. *Robotics and Automation Magazine, 13(2)*, 99–110.

Faverjon, B., & Tournassoud, P. (1987a). A local based approach for path planning of manipulators with a high number of degrees of freedom. In *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA)*, pp. 1152–1159.

Faverjon, B., & Tournassoud, P. (1987b). The mixed approach for motion planning: Learning global strategies from a local planner. In *Intl. Joint Conf. on Artificial Intelligence (IJCAI).*

Ferguson, D., Kalra, N., & Stentz, A. (2006). Replanning with RRTs. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1243–1248.

Fiorini, P., & Shiller, Z. (1998). Motion planning in dynamic environments using velocity obstacles. *Intl J. of Robotics Research, 17(7)*, 760–772.

Flanagan, J., Vetter, P., Johansson, R., & Wolpert, D. (2003). Prediction precedes control in motion learning. *Current Biology, 13*, 146–150.

Gaffney, S., & Smyth, P. (1999). Trajectory clustering with mixtures of regression models. In *Conference in Knowledge Discovery and Data Mining*, pp. 63–72.

Gayle, R., Sud, A., Lin, M., & Minocha, D. (2007). Reactive deforming roadmaps: Motion planning of multiple robots in dynamic environments. In *Proc. IEEE/RSJ Int'l Conf. on Intelligent Robots and Systems (IROS)*.

Ge, Q., & McCarthy, J. (1989). Equations for boundaries of joint obstacles for planar robots. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 164–169.

Grollman, D. H., & Jenkins, O. C. (2008). Sparse incremental learning for interactive robot control policy estimation. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

Haigh, K. Z., & Veloso, M. (1995). Route planning by analogy. In Veloso, M., & Aamodt, A. (Eds.), *Case-Based Reasoning Research and Development. First International Conference, ICCBR-95*, pp. 169–180, Sesimbra, Portugal. (Berlin, Germany: Springer-Verlag).

Haigh, K. Z., Shewchuk, J. R., & Veloso, M. M. (1997). Exploiting domain geometry in analogical route planning. *JETAI, 9*(4), 509–541.

Hauser, K., Bretl, T., & Latombe, J.-C. (2005). Learning-assisted multi-step planning. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*.

Holleman, C., & Kavraki, L. (2003). A framework for using the workspace medial axis in prm planners. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1408–1413.

Hsiao, K., Kaelbling, L., & Lozano-Pérez, T. (2007). Grasping pomdps. In *Proc. IEEE Conf. on Robotics and Automation (ICRA).*

Hsu, D., Jiang, T., Reif, J., & Sun, Z. (2003). The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pp. 4420–4426.

Hsu, D., Kindel, R., Latombe, J.-C., & Rock, S. (2002). Randomized kinodynamic motion planning with dynamic obstacles. *Intl. J. of Robotics Research, 21(3)*, 233–255.

Hsu, D., Latombe, J.-C., & Motwani, R. (1999). Path planning in expansive configuration spaces. *Intl. J. of Computational Geometry and Applications, 9(4-5)*, 495–512.

Hsu, D., Sánchez-Ante, G., l. Cheng, H., & Latombe, J.-C. (2006). Multi-level free-space dilation for sampling narrow passages in prm planning. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1255–1260.

Hsu, D., Sánchez-Ante, G., & Sun, Z. (2005). Hybrid prm sampling with a cost-sensitive adaptive strategy. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 3885–3891.

Hwang, Y., & Ahuja, N. (1992). Gross motion planning – a survey. *ACM Computing Surveys, 24(3).*

Jaillet, L., & Siméon, T. (2004). A prm-based motion planner for dynamically changing environments. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems.*

Jordan, M., & Rumelhart, D. (1992). Forward models: Supervised learning with a distal teacher. *Cognitive Science, 16*, 307–354.

Kalisiak, M., & van de Panne, M. (2007). Faster motion planning using learned local viability models. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA).*

Kallmann, M., & Matarić, M. (2004). Motion planning using dynamic roadmaps. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '04).*

Kavraki, L., Svestka, P., Latombe, J.-C., & Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. Robot. and Autom., 12(4)*, 566–580.

Kawato, M. (1999). Internal models for motor control and trajectory planning. *Current Opinion in Neurobiology, 9*, 718–727.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *Intl. J. of Robotics Research, 5*(1).

Krakauer, J., Ghilardi, M.-F., & Ghez, C. (1999). Independent learning of internal models for kinematics and dynamic control of reaching. *Nature Neuroscience, 2(11)*, 1026–1031.

Kuffner, J., & Lavalle, S. (2000). RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pp. 995–1001.

Kurniawati, H., & Hsu, D. (2004). Workspace importance sampling for probabilistic roadmap planning. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS).*

Latombe, J.-C. (1988). Motion planning with uncertainty: The preimage backchaining approach. Tech. rep. CS-TR-88-1196, Stanford University.

Latombe, J.-C. (1991). *Robot Motion Planning.* Kluwer Academic Publishers.

LaValle, S. (1998). Rapidly-exploring random trees: A new tool for path planning. Tech. rep. TR 98-11, Computer Science Dept., Iowa State University.

Lavalle, S. (2004). On the relationship between classical grid search and probabilistic roadmaps. *Intl. Journal of Robotics Research, 23(7-8)*, 673–692.

LaValle, S., & Sharma, R. (1997). On motion planning in changing, partially-predictable environments. *International Journal of Robotics Research, 16*, 775–805.

LaValle, S. M. (2006). *Planning Algorithms.* [Online]. Available at http://planning.cs.uiuc.edu/.

Lazanas, A., & Latombe, J. (1995). Motion planning with uncertainty: A landmark approach. *Artificial Intelligence, 76*(1-2).

Leven, P., & Hutchinson, S. (2002). A framework for real-time path planning in changing environments. *International Journal of Robotics Research, 21(12)*, 999–1030.

Leven, P., & Hutchinson, S. (2003). Using manipulability to bias sampling during the construction of probabilisitic roadmaps. *IEEE Transactions on Robotics and Automation, 19(6)*, 1020–1026.

Li, T.-Y., & Latombe, J.-C. (1997). On-line manipulation planning for two robot arms in a dynamic environment. *International Journal of Robotics Research, 16(2)*, 144–167.

Li, T.-Y., & Shie, Y.-C. (2002). An incremental learning approach to motion planning with roadmap management.. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 3411–3416.

Lien, J.-M., Thomas, S. L., & Amato, N. M. (2003). A general framework for sampling on the medial axis of the free space. In *Proc. IEEE Intl. Conf.on Robotics and Automation (ICRA)*, pp. 4439–4444.

Lindemann, S., & Lavalle, S. (2004). Incrementally reducing dispersion by increasing voronoi dispersion in RRTs. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 3251–3257.

Lopes, M., & Santos-Victor, J. (2006). Learning sensory-motor maps for redundant robots. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS).*

Lozano-Pérez, T. (1983). Spatial planning: A configuration space approach. *IEEE Trans. on Computers, C-32(2)*, 108–120.

Lozano-Pérez, T. (1987). A simple motion-planning algorithm for general robot manipulators. *IEEE J. of Robotics and Automation, RA-3(3)*, 224–238.

Lozano-Pérez, T., Mason, M., & Taylor, R. (1984). Automatic synthesis of fine-motion strategies for robots. *Intl. J. of Robotics Research, 3(1)*, 3–24.

Lozano-Pérez, T., & Wesley, M. (1979). An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM, 22(10)*, 560–570.

Mazer, E., Ahuactzin, J., Talbi, G., & Bessiere, P. (1998). The ariadne's clew algorithm. *Journal of Artificial Intelligence Research, 9*, 295–316.

McLean, A., & Laugier, C. (1996). Update and repair of a roadmap after model error discovery. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pp. 917–924.

Molina-Vilaplana, J., no Molina, J. P., & López-Coranado, J. (2004). Hyper rbf model for accurate reaching in redundant robotic systems. *Neurocomputing, 61*, 495–501.

Morales, M., Tapia, L., Pearce, R., Rodriguez, S., & Amato, N. (2004). A machine learning approach for feature-sensitive motion planning. In *Proc. Intl. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, pp. 361–376.

Nieuwenhuisen, D., van den Berg, J., & Overmars, M. (2007). Efficient path planning in changing environments. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems.*

Nikovski, D., & Nourbakhsh, I. (2002). Learning probabilistic models for optimal visual servo control of dynamic manipulation. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS).*

Ó'Dúnlaing, C., & Yap, C. (1985). A "retraction" method for planning the motion of a disc. *Journal of Algorithms, 6*, 104–111.

Paden, B., Mess, A., & Fisher, M. (1989). Path planning using a jacobian-based freespace generation algorithm. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1732–1737.

Peters, J., & Schaal, S. (2007). Reinforcement learning by reward-weighted regression for operational space control. In *Proc. of the Intl. Conf. on Machine Learning.*

Randlov, J., & Alstrom, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. In *Proc. of the Intl. Conf. on Machine Learning.*

Reif, J. (1979). Complexity of the mover's problem and generalizations. In *Proceedings of the 20th IEEE Symposium of the Foundations of Computer Science*, pp. 421–427.

Reif, J., & Sharir, M. (1994). Motion planning in the presence of moving obstacles. *Journal of the ACM, 41*, 764–790.

Rodriguez, S., Lien, J.-M., & Amato, N. (2005). Planning motion in completely deformable environments. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 2466–2471.

Rodriguez, S., Thomas, S., Pearce, R., & Amato, N. (2006). Resampl: A region-sensitive adaptive motion planner. In *Proc. Intl. Workshop on the Algorithmic Foundations of Robotics (WAFR).*

Roweis, S., & Saul, L. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science, 290(5500)*, 2323–2326.

Saha, M., & Latombe, J.-C. (2005). Finding narrow passages with probabilistic roadmaps: The small step retraction method. In *Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS).*

Sanchez, G., & Latombe, J.-C. (2001). A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *International Symposium on Robotics Research.*

Schwartz, J., & Sharir, M. (1983). On the piano movers' problem: Ii. general techniques for computing topological properties of algebraic manifolds. *Advances in Applied Mathematics, 4*, 298–351.

Siméon, T., Laumond, J.-P., Cortés, J., & Sahbani, A. (2004). Manipulation planning with probabilistic roadmaps. *Intl. J. of Robotics Research, 23*, 729–746.

Siméon, T., Laumond, J.-P., & Nissoux, C. (2000). Visibility-based probabilistic roadmaps for motion planning. *Advanced Robotics Journal, 14*(6).

Smart, W. D., & Kaelbling, L. P. (2000). Practical reinforcement learning in continuous spaces. In Langley, P. (Ed.), *Proc. Intl. Conf. on Machine Learning (ICML)*, pp. 903–910, San Francisco, CA. Morgan Kaufmann.

Sud, A., Gayle, R., Guy, S., Anderson, E., Lin, M., & Manocha, D. (2007). Real-time navigation of independent agents using adaptive roadmaps. In *Proc. ACM Symposium VCRST.*

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* MIT Press.

Tenenbaum, J., de Silva, V., & Langford, J. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science, 290(5500)*, 2319–2323.

Thomas, S., Morales, M., Tang, X., & Amato, N. (2007). Biasing samplers to improve motion planning performance. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1625–1630.

Whitney, D. (1969). Resolved motion rate control of manipulators and human prostheses. *IEEE Trans. on Man-Machine Systems, 10(2)*, 47–53.

Wilmarth, S. A., Amato, N. M., & Stiller, P. F. (1999). Maprm: A probabilistic roadmap planner with sampling on the medial axis of free space. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pp. 1024–1031.

Wolpert, D., & Flanagan, J. (2001). Motor prediction. *Current Biology, 11(18).*

Yang, Y., & Brock, O. (2006). Elastic roadmaps: Globally task-consistent motion for autonomous mobile manipulation in dynamic environments. In *Proc. Robotics: Science and Systems.*

Yershova, A., Jaillet, L., Siméon, T., & LaValle, S. (2005). Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain. In *Proc. IEEE Intl. Conf. on Robotics and Automation (ICRA).*