Graph-based Trajectory Planning through Programming by Demonstration

Nik A. Melchior
melchior@cmu.edu

CMU-RI-TR-11-40

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics.

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

July 2011

Thesis Committee:
Reid Simmons (chair)
Manuela Veloso
Jeff Schneider
O.C. Jenkins (Brown University)

Abstract

Autonomous robots are becoming increasingly commonplace in industry, space exploration, and even domestic applications. These diverse fields share the need for robots to perform increasingly complex motion behaviors for interacting with the world. As the robots' tasks become more varied and sophisticated, though, the challenge of programming them becomes more difficult and domain-specific. Robotics experts without domain knowledge may not be well-suited for communicating task-specific goals and constraints to the robot, but domain experts may not possess the skills for programming robots through conventional means. Ideally, any person capable of demonstrating the necessary skill should be able to instruct the robot to do so. In this thesis, we examine the use of demonstration to program or, more aptly, to teach a robot to perform precise motion tasks.

Programming by Demonstration (PbD) offers an expressive means for teaching while being accessible to domain experts who may be novices in robotics. This learning paradigm relies on human demonstrations to build a model of a motion task. This thesis develops an algorithm for learning from examples that is capable of producing trajectories that are collision-free and that preserve non-geometric constraints such as end-effector orientation, without requiring special training for the teacher or a model of the environment. This approach is capable of learning precise motions, even when the precision required is on the same order of magnitude as the noise in the demonstrations. Finally, this approach is robust to the occasional errors in strategy and jitter in movement inherent in imperfect human demonstrations.

The approach contributed in this thesis begins with the construction of a *neighbor graph*, which determines the correspondences between multiple imperfect demonstrations. This graph permits the robot to plan novel trajectories that safely and smoothly generalize the teacher's behavior. Finally, like any good learner, a robot should assess its knowledge and ask questions about any detected deficiencies. The learner presented here detects regions of the task in which the demonstrations appear to be ambiguous or insufficient, and requests additional information from the teacher. This algorithm is demonstrated in example domains with a 7 degree-of-freedom manipulator, and user trials are presented.

Acknowledgements

I would like to thank my advisor, Reid Simmons, for his support and assistance in this work and throughout my graduate career. I am grateful for for his guidance and shared knowledge in the many areas of robotics that I have been privileged to explore with him. I am also grateful to my master's advisor, Bill Smart, whose enthusiasm and encouragement ignited my interest in robotics. I hope to find opportunities to work with each of them in the future.

I would not have been able to complete this work without the helpful discussions and information provided by my committee members, Manuela Veloso, Jeff Schneider, and Chad Jenkins. Chad's thesis work inspired my own, and I am grateful for the code he shared with me. I also received indispensable assistance from the people at Intel Labs Pittsburgh, who graciously allowed me to use their robot for my experiments. I am particularly grateful to Mike Vande Weghe, Dmitry Berenson, and Mehmet Doğar for the time they spent introducing me to HERB.

The Syndicate/Trestle/IDSR/Ace/Boogaloo team has been an invaluable source of joy, wisdom, and growing experiences for me. I have enjoyed working with and getting to know Brad Hamner, Brennan Sellner, Fred Heger, Laura Hiatt Magill, Seth Koterba, and Breelyn Kane through the years that these projects have evolved. I am also indebted to the staff at CMU who have enabled my work and kept both people and machines running smoothly: Greg Armstrong, Karen Widmaier, and Marliese Bonk.

My family has provided endless love and support during my many years as a student. My parents, Al and Linda Melchior, have made sacrifices to encourage my exploration in everything that held my interest and to help me achieve my goals – even when my goals have taken me far from them. My sister, Kim Welton, has graciously endured growing up with a nerd, and has helped me to grow in more ways than she can know.

I am thankful for the friends who have become family while I pursued my education. My pastor Steve Wilson has been a mentor to me, and I am grateful for all he has taught me. My friends at City Reformed Church, and especially the SQCG, have sustained me with their prayers, kind words, and good food. My friends in Clan Zwerdog have impressed and encouraged me through their insights and accomplishments without forsaking the immaturity of our college years.

Finally, I thank my Lord and Savior Jesus Christ for calling me and providing me with a purpose in life. *Soli Deo gloria*.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

> *Make experiements. Seek the recipe of life... Show us what we must do. The Robots can accomplish everything that the human beings showed them.*
>
> — Radius, *R.U.R. (Rossum's Universal Robots)*, 1920

Robots are becoming a more common sight in many domains. They are used as tools for manufacturing, instruments for surgery, and toys for consumers. As their areas of application expand, so does the diversity of their operators. Though these users may be novice robot programmers, they often posses specialized knowledge about their application domain; knowledge that must be transfered to the robot if it is to perform its task. Programming by demonstration (PbD) is an approach that facilitates knowledge transfer from a domain expert to an autonomous system. It provides an intuitive approach for someone skilled in performing a task to teach a robot to perform that task without having to learn to program the robot. Additionally, a robotics expert is not required to become skilled in the task.

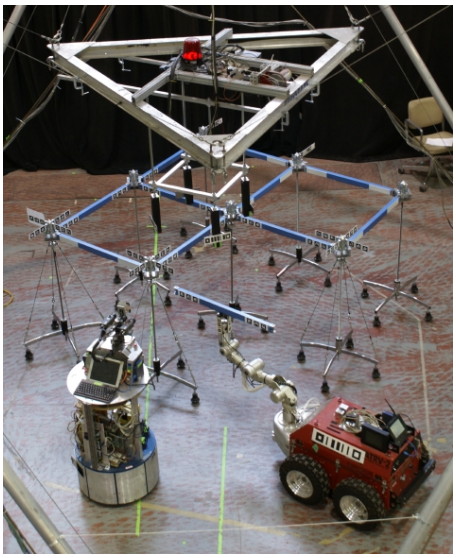
One of the primary difficulties in PbD approaches to robotic manipulation is representing and understanding the constraints imposed by the physical world. Physical obstacles, or *geometric* constraints, are the most obvious issues. A robot tasked with reaching out to grasp a target object or placing an object in a specified location must not bump into other objects while it is working. But precisely detecting the locations of objects, even in a static environment, can be difficult for current sensor technologies. Computer vision and LIDAR sensors can have difficulty detecting the precise shapes and positions of irregularly-shaped objects,

particularly those with spindly projections or hollows. Additionally, sensors must be located to view all relevant features of the objects without suffering from occlusion due to the objects or the robot itself. Alternately, the robot may be provided with an a priori model of the objects of interest, but constructing this model may again require the skills of a specialist.

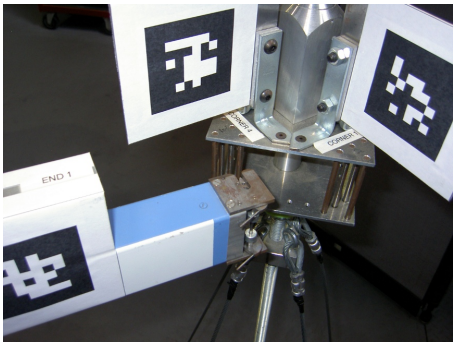
Even if a model is constructed to provide the robot with knowledge of obstacles, it must also consider *non-geometric* task constraints. These are prohibitions or prescriptions of movements that are not directly concerned with avoiding contact between the robot and physical obstacles. For example, a robot carrying a cup of liquid must maintain its end-effector orientation to avoid spilling. A robot routing cable around complex objects may need to follow a particular path to avoid snagging the cable. While a detailed physical simulation may be able to detect sequences of motions that allow the cable to become entangled with objects in the environment, it would be challenging for a novice robot user to communicate this constraint to a planner. However, if a user is able to demonstrate successful strategies for completing the task, he does not need to articulate the criteria that he is optimizing in a manner comprehensible to the robot.

Thus, PbD offers the means for a domain expert to teach a robot about geometric and non-geometric task constraints without explicitly formalizing these constraints. The robot may then create novel plans respecting these constraints, usually through generalization of the actions performed by the teacher. Since the amount of available demonstration is typically limited with respect to a robot learner's (typically high-dimensional and continuous) state space, the primary challenge of PbD systems is to determine behavior in undemonstrated situations. In fact, if imperfect demonstrations are admitted, as in this thesis, then example behaviors are not to be precisely reproduced when planning.

Another advantage of the PbD approach is that it offers the learner an opportunity to request additional information and clarification in parts of the task where the initial demonstrations are unclear. In this work, we develop a strategy for resolving *bifurcations* in demonstrated strategies: areas in which the teacher has provided conflicting advice. For example, demonstrations may take different routes around obsta-



(a)

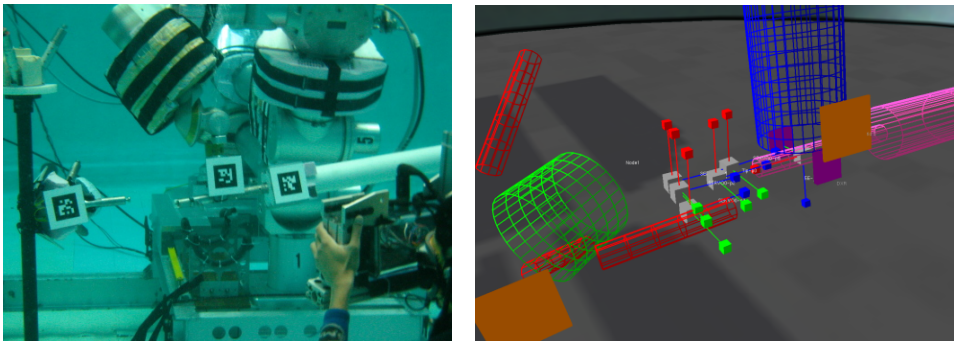


(b)

Figure 1.1: The Trestle construction task. (a) A grid of beams constructed by mobile robots. (b) The beam connections have tight mechanical tolerances.

cles, or a redundant manipulator may follow the same path with different configurations. Whether the teacher purposefully demonstrated diverging strategies, or did so accidentally, a learner that recognizes the presence of multiple strategies can ask the teacher which strategy it should follow in the plans that it creates. This leads to robot plans that are more predictable, repeatable, and possibly closer to the teacher's intentions, even if he was not able to perfectly demonstrate these intentions.

This work is motivated by robot manipulation tasks that require precise movements, such as construction and assembly tasks. For example, the Trestle construction scenario [60] shown in Figure 1.1(a) involves a team of heterogeneous robots constructing a grid of blue *beams*, connected by silver *nodes* on stanchions. In this task, the crane (top) braces the nodes and enables gross movements of the partially-completed structure, and the silver robot (bottom left) carries a stereo camera pair for viewing and localizing construction components. Assembly operations



(a) (b)

Figure 1.2: The IDSR construction task executed underwater. (a) The robot manipulates a beam to connect it with the stationary node on the left. (b) A simulation of the same scenario. The gray boxes with multi-colored coordinate axes are motion waypoints.

are primarily accomplished by the 7 degree-of-freedom (DOF) arm on a mobile base (bottom right). This arm grasps a beam and moves it so that the spring-loaded hook on the end connects to a receptacle on a node. The closeup view of the connection mechanism in Figure 1.1(b) reveals that the assembly operation requires a precise approach strategy. The beam must be positioned between the top and bottom plates of the node and precisely oriented so that the notches on the beam end align with a pin on the node. The visual fiducials, which appear to be 2-dimensional barcodes, are attached to beams and nodes to provide identification and localization of the construction components relative to one another.

Figure 1.2(a) illustrates a similarly constrained construction operation. In this scenario, a robot assembles a portion of the EASE structure [61], a large underwater truss used in neutral-buoyancy astronaut training. Again, the structure of the node and beam components necessitates a precise assembly strategy. The simulation in Figure 1.2(b) shows the node as a green and red wireframe object on the left side of the image, and a beam (pink) grasped by the robot (blue) on the right. The gray boxes represent hand-tuned motion waypoints defined relative to the connection point on the node, which provide a motion plan for performing the assembly operation. The end of the beam is guided through

the waypoints in order to form a connection.

In each of these tasks, precise motion strategies are required, and are executed in domains in which dynamics do not play a significant role. These strategies are intuitive to a person familiar with the hardware, but can still be challenging to articulate in a format amenable to execution by a robot manipulator. The motion waypoints illustrated in Figure 1.2(b) are positioned relative to a goal point on the node, and each has its own coordinate frame. Three translational and three rotational tolerances are specified in the local frame of each waypoint. Such plans are challenging for a roboticist to produce, despite familiarity with kinematics and transforming coordinate frames. In fact, for the tasks pictured here, the waypoints and tolerances were refined through trial and error over the course of dozens of connection attempts by users who were familiar with the robots, their control software, and the construction hardware in use.

If robots are to enjoy widespread use as tools for these sort of construction and assembly operations, though, they should be more easily programmed by someone familiar with only the task hardware. We would like robots to be more easily employed as tools for similar assembly tasks in fields such as automobile manufacturing and circuit board assembly. In the cases described above, achieving a final end-effector configuration was the goal, but motion was constrained to a particular strategy by the shapes of the components. In other tasks, such as welding seams and routing cables, the goal is defined by the entire motion.

This thesis considers similar motion tasks in which the motions are constrained relative to some target object. We do not require a model of the target or the larger environment in which the robot operates. The robot only needs to be able to localize itself relative to the target so that it can create a local coordinate frame for relating multiple examples of the task. When operating in a static environment, a model of the robot is used to ensure the safety of generated plans. Our goal is to learn to repeat the execution of a task like the ones described above, in which the desired motion relative to the target is repeated with variations in the starting configuration, such as one might see on an assembly line.

1.1 Thesis Statement

Thesis: *A Programming by Demonstration approach to planning is able to construct safe, efficient, natural motion plans for robotic manipulators by using imperfect demonstrations and requesting advice when demonstrations conflict.*

This document details the algorithms created for learning the structure of tasks from demonstration trajectories, detecting inconsistencies and errors in demonstrations, and planning new trajectories to accomplish the same task. We show that the new plans safely avoid static obstacles without requiring a model of the environment. Plans are further evaluated in terms of efficiency and naturalness by showing that they exhibit shorter path length and lower curvature than example trajectories without deviating from the task structure demonstrated. Finally, user trials show that this method provides a safe, intuitive method for novices to transfer knowledge to a robot.

The following subsections introduce the three main contributions of this thesis. First, we describe our neighbor graph, the structure for relating imprecise demonstrations of a robotic motion tasks, and the procedure for constructing it. Next, our approach to planning exploits this neighbor graph to produce novel plans for the robot to execute the demonstrated task. Finally, we introduce our active learning strategies, in which the robot learner solicits additional guidance from the teacher to refine its knowledge of the task and its ability to plan.

1.2 Neighbor Graph

Demonstration presents a powerful modality for programming a robot without sacrificing accessibility to novice users. Kinesthetic demonstrations in particular require no specific knowledge of the robot's kinematics, nor a computer model of the robot's environment. The trade-off, though, is that the robot possesses no objective function for comparing the relative value of multiple examples of the same task. Indeed, before solving this unsupervised learning problem, correspondences must be found between portions of the examples.

The *neighbor graph* is our means for representing the structure of the demonstrated task and the relationships between multiple example trajectories. The example trajectories are more than simply collections of points in the robot's workspace or configuration space. As time-series data, they implicitly encode a *task space*, representing a motion demonstrated through execution of the task. However, multiple examples, even of the same overall strategy, will not be identical to one another. Some variations arise due to lack of constraints in the task itself. When precision is not required, demonstrations tend to diverge from one another. Other variations, though, arise due to imperfections and errors in demonstrations: motions that should not be reproduced in new plans. The challenge in building a neighbor graph is to detect and distinguish these variations and build a structure representing the task being demonstrated.

Our algorithm for building this structure was inspired by ST-Isomap [34], an algorithm for relating time-series (Spatio-Temporal) information, and embedding it in a lower-dimensional space. To improve the ability of ST-Isomap to correlate examples, we augment this approach with heuristics designed to detect several issues typical to motion trajectories. The resulting structure organizes multiple demonstrations of the same task and insulates imperfections and other undesired bits of demonstrations to reduce their influence on new plans.

Although the best evaluation of a neighbor graph of this kind is the plans that can be built from it, we examine specific problematic cases to ensure that common errors do not produce degenerate neighbor graphs. For example, graphs created in non-Markovian spaces should not include links between portions of the examples that overlap in space but occur at different points in the task. Backtracking motions and other minor deviations unique to a single example should not be connected to other demonstrations. Instead, a proper neighbor graph should include a strong web of connections only between similar portions of the demonstration trajectories.

1.3 Planning

The goal of Programming by Demonstration is not merely to understand the task being demonstrated, but to be able to perform the task autonomously. Additionally, since we expect the teacher's demonstrations to be imperfect, the robot learner must avoid the jitter, unintended deviations, and other errors apparent in provided examples. Much of the work of detecting these errors is accomplished while constructing the neighbor graph, which the planner follows from start to goal of the task. The planner creates a new trajectory that remains near clusters of demonstration trajectories in order to produce a plan that accomplishes the task using the same general strategy as the demonstrations. This requirement distinguishes PbD from most traditional motion planning techniques since the quality of the plan does not depend on a metric or objective function over the space or actions over which the learner operates. Instead, our planner interpolates between portions of demonstration trajectories linked in the neighbor graph. Since this graph connects similar portions of the demonstration trajectories and elides links in areas of deviations, planned paths should represent the strategy intended by the teacher.

Our experimental evaluation of plans produced by this algorithm show that they safely, smoothly, and efficiently follow the demonstrations provided. Since our approach does not require a model of the environment, safety, in our case, means avoiding static obstacles. We show that new plans do not cause the robot to occupy any space that was not occupied during the production of example demonstrations. We also show that planned paths exhibit noticeably less curvature while retaining the overall shape of the examples. This leads to a small reduction in path length (on the order of 7%) for planned trajectories.

1.4 Active Learning

Localized errors and imperfections in demonstration trajectories are largely avoided in planned trajectories by heuristics used in the construction of the neighbor graph and the techniques used by the planner. However, collections of imperfect demonstrations sometimes exhibit inconsisten-

cies that cannot be interpreted as accidental errors. Whether by accident or design, sets of demonstration trajectories may contain systematic differences: distinctions that split the examples into two or more internally consistent subsets. For instance, the teacher may demonstrate multiple distinct strategies. Since planning relies on the ability to interpolate between similar demonstration trajectories, it is important that distinct strategies be identified and considered separately.

In this thesis, our learning algorithm identifies *bifurcations*, locations where demonstrations split into multiple strategies. Trajectories are planned in each branch of the bifurcation and presented to the teacher as a means of soliciting additional information. The teacher is asked to specify which branch the learner should use when planning new paths.

This approach is evaluated through user trials conducted with both robot experts and novices. Participants were asked to teach a robot to perform a simple motion task, possibly demonstrating multiple strategies. The robot identified diverging strategies, when present, and demonstrated the distinction to the teacher. Finally, the teacher indicated which strategy the robot should choose, and a final path was generated and executed by the robot. The results showed that users with a wide range of robotics experience were able to teach a robot to successfully perform the demonstrated task, mimicking their own strategy.

1.5 Document Outline

- Chapter 2 describes related work in the areas of Programming by Demonstration, motion planning, and machine learning.

- Chapter 3 discusses an approach to PdD that relies on dimensionality reduction. Similar approaches have shown some promise and our early work pursuing this strategy provided valuable insights into the PbD problem. However, this chapter also evaluates why dimensionality reduction is unable to create smooth, natural plans, especially when provided with imperfect demonstrations.

- Chapter 4 provides an overview of our PbD approach and introduces the components of our work.

- Chapter 5 describes the neighbor graph that is constructed from demonstration trajectories provided to the robot learner. This graph provides a framework for relating the demonstrations to one another and the means for planning and active learning. The application of dimensionality reduction provides a means for intuitive evaluation of the graphs created.
- Chapter 6 details our approach for creating novel task plans using the neighbor graph described in Chapter 5. Experimental trials demonstrate that the plans created are safe, smooth, and efficient, while following the strategy demonstrated by the teacher.
- Chapter 7 describes the learner's active approach to detecting and resolving conflicting information in demonstration trajectories. Specifically, the learner detects situations in which demonstrations diverge from one another and requests advice as to how it should plan when these areas are encountered. User trials show that this method is effective for users with various levels of experience in operating robots.
- Chapter 8 presents conclusions and future work, as well as a summary of the thesis.

1.6 Summary

This thesis presents a complete set of algorithms for understanding the structure of a motion task through demonstration trajectories and planning novel trajectories for accomplishing that task. This strategy permits knowledge transfer from a domain expert to a robot without requiring traditional robot programming skills. Additionally, our approach does not require perfect demonstrations of the task to be learned. Instead, minor errors and deviations in demonstrations are detected and ignored by the learner, while significant conflicts between demonstrations are resolved by asking for clarification advice from the teacher. The robot is the capable of creating new plans that respect the strategies demonstrated by the teacher. Experiments and user trials show that the planned trajectories are safe and efficient, and are able to elide errors

in demonstrations without deviating significantly from the structure of the demonstrations. Moreover, the training procedure is intuitive and accessible for teachers with little, or no, experience operating robots.

Chapter 2
Related Work

> *For the most part I write myself. That is, I have the innate ability to learn from experience. But this ability was originally coded into me by my creator.*
>
> — The Librarian, *Snow Crash*, 1992

Programming by Demonstration has strong roots in several fields of robotics, so we examine related techniques in each of them. Since we are interested in learning trajectories, or plans for motion, based on examples, this work lies between the traditional approaches in Motion Planning and Machine Learning. Without an explicit model of the environment, most motion planning techniques cannot be directly applied to this problem. On the other hand, some machine learning techniques risk over-generalizing behaviors without the spatial knowledge required to precisely determine areas of applicability, or the ability to obtain additional information in uncertain or undemonstrated areas. PbD can be viewed as a form of *Supervised Learning*, in which the learner is presented with a set of labelled training data from which it must produce a policy. Many recent techniques also rely on teacher feedback during plan execution or asking questions of the teacher. These Active Learners are able improve their policy in uncertain and undemonstrated states. PbD is not limited to learning motion trajectories. These techniques are also useful for learning high-level, symbolic behaviors and their relationship. These behaviors can then be assembled to perform tasks, often involving complex motions. We will pay special attention to a family of PbD techniques that makes use of dimensionality reduction, since these techniques formed the basis of our original approach to PbD.

Throughout the literature, a variety of terms have been used to refer to Programming by Demonstration and related techniques. These include Learning from Demonstration, Learning by Showing, Imitation Learning, Apprenticeship Learning, and Behavioral Cloning. Extensive overviews of the field [12] and literature reviews [4] explore these topics more fully.

2.1 Motion Planning

Motion planning is a broad and deep field at the very heart of mobile robotics. Although the task of learning trajectories from demonstration could be considered a branch of motion planning, the traditional definition of this term differs significantly from what we hope to achieve. Specifically, motion planning does not deal with learning from examples, but rather generating a plan based on knowledge of the environment. This knowledge may be known in detail at the outset, or it may be sensed as the robot moves about. Typical examples include grid-based planners such as A* and D* [68], as well as randomized planners such as Rapidly-exploring Random Trees (RRT) [41] and Probabilistic Roadmaps (PRM) [37]. Since this type of approach might be possible for our task, we examine its applicability.

We have already argued against manual measurement and modelling of the task workspace, but it might also be possible to demonstrate the bounds of the workspace by teleoperating the robot itself near the limits of safe motion. Alternatively, stereo vision or laser range-finders might be used to model simple environments, but these sensors may fail in cluttered or unstructured environments, especially in the presence of small obstacles or obstacles that occlude the view of the sensor. Even when a model can be obtained, though, this approach offers no means to convey non-geometric constraints, and it is difficult to perform safely and completely. However, some of the seminal work in learning from examples, done by Haruhiko Asada [5, 6], uses constrained movement along obstacle surfaces to find the important components of demonstrated control. More recent work [14] builds on this approach to learn a set of control primitives that move between obstacle surfaces in configuration

space. Trajectory generation then consists of combining these primitives to form an efficient plan. This approach requires a compliant manipulator, or very precise sensing and actuation. Other early work is described by Kuniyoshi, Inaba, and Inoue [40], and tackles the entire problem from visual recognition of human activity to symbolic dependency analysis for ordering the subtasks.

However, we argue that a non-programmer would have difficulty determining the extent of the robot's knowledge during training and recognizing when the demonstration is complete. A non-programmer would have further difficulty expressing the task itself in this framework. Motion planners generally allow specification of only the goal position. This should be simple to convey, but no additional constraints may be easily specified, though a skilled programmer may be able to construct them. The waypoint-based techniques described in Chapter 1 fall into this category. The waypoints provide pose constraints that the robot must achieve in the specified order, but do not specify actions between them. In practice, programmers often rely on implicit constraints provided by their knowledge of the robot's motion planner, and specify waypoints close enough to one another to ensure predictable behavior between them. Again, this technique is difficult for inexperienced users to master.

2.2 Supervised Learning

Inspiration for our approach is also drawn from the field of machine learning. Given a set of example trajectories, there are many strategies for generalizing the examples for a robot to perform the task, typically relying on some form of interpolating or weighted blending of example trajectories. The simplest algorithm is nearest neighbor: the robot simply reenacts one of the human-provided example trajectories. We argue that this technique is undesirable because it does not consider safety when moving from the initial pose onto the demonstration trajectory, and it forces the robot to reproduce all of the jitter and inaccuracy found in the demonstrated trajectories. An extension to this technique, k-nearest neighbor (k-NN) [7], blends the actions of k examples into a new trajectory for the robot to follow. While this can smooth out some of the jitter

or mistaken movements in individual examples, it carries the danger of blending incompatible examples if the algorithm is not guided in some way. Nevertheless, these simple techniques are useful as components of more complex algorithms.

Atkeson provides an excellent survey [7] of techniques for learning from examples. An application of these concepts is presented by Bentivegna [10], using domain-specific primitives to describe training examples. Ratliff [54] attempts to learn cost functions based on features of the provided examples. Pook and Ballard [52] use k-NN to learn behavioral primitives that are combined using Hidden Markov Models (HMMs). Stolle [69] uses a variety of traditional motion planning techniques to quickly generate example trajectories, then uses nearest neighbor to find an example applicable to a given state. The planner is invoked again when failure is detected during execution. Similarly, Reinforcement Learning techniques [65, 78] can rely on automatically computed actions that, unlike our approach, require a model for generation.

Work by Ijspeert, Nakanishi, and Schall [32] has focused on reconstructing a single example by blending a set of primitive basis functions. There are some mathematical advantages to this approach, especially if the manipulator may be perturbed during execution. Drumwright, Jenkins, and Matarić [23] describe a method for learning similar types of primitives. These primitives can be used for activity recognition and classification, as well. Kaiser and Dillmann [35] exploit task knowledge to smooth irrelevant motions from even a single example trajectory.

A significant amount of work has also been devoted to fitting multiple training trajectories using various types of splines. Ude, Atkeson, and Riley [76] used B-splines to learn joint trajectories. Lee [42] used splines in phase space in order to fit both position and velocity. Aleotti, Caselli, and Maccherozzi [1] used NURBS curves to learn trajectories, relying on previous findings [70] that NURBS are optimal under certain conditions. A multi-layered approach in this category is described by Campbell and colleagues [13]. In this work, simple blending is used to smooth example trajectories in identical instances of a simple reaching task. However, a more complex procedure is used to interpolate between different instances of the task in which the goal position is moved about

the workspace. As in k-NN-based approaches, though, blending between splines introduces the danger of planning through unmodelled obstacles and reproducing undesired features of the examples. This approach also requires careful consideration of the scale of features to be retained in learned trajectories. Sparse spline control points may be allowable in most parts of a task, but precise manipulation may require denser control points to produce small movements or sharp turns at other times. These spline-based approaches, like the primitives-based approaches described above, impose a structure that is then used to build new plans. Although this structure may be flexible, and is sometimes generated as part of the learning approach, it imposes representational constraints on the trajectories that are not required by our neighbor-graph strategy.

Other approaches attempt to learn the dynamical models of the system using Gaussian Mixture Models (GMMs) [29], Hidden Markov Models (HMMs) [16], Neural Networks [8, 51], or other statistical feedback techniques [44]. Such models may have trouble planning in non-Markovian spaces unless the state space is made more complex (e.g. by increasing its dimensionality). Finding a feasible state representation can be challenging, but our neighbor graph avoids this by providing a means to ensure a plan is "connected" to the appropriate portions of demonstrations, even in non-Markovian spaces.

2.3 Programming by Demonstration with Feedback

Robot learners do not always produce the precise behavior intended by their teachers. Even when they are provided with high-quality examples from which to learn, they may combine them in unexpected ways such as interpolating two motions that travel on opposite sides of an obstacle. Moreover, human teachers are unlikely to provide perfect examples to the learner. Jitter and other unintended movements may appear in the training examples, and thus be reproduced by the learner.

One solution to this problem is to ask the user to inspect the generated plan to ensure that it is safe and lacks unintended features. Aleotti [1] introduces such a solution: the planned path is displayed in a

virtual environment and the user is able to edit it by clicking and dragging with the mouse. Of course, this solution requires a complete model of the environment, which is undesirable or impossible in many situations.

Dillman and colleagues have investigated similar solutions [26] that initiate a dialog with the user to inspect and correct generated trajectories in a simulated environment. Their system also queries the user as to symbolic constraints such as ordering between subtasks. More recently, they have worked on subtask activity recognition [83]. Similar work by Kang [36] used grasp recognition to segment portions of the trajectory.

Other work has used the jitter and human variability to determine the range of obstacle-free workspace. A series of papers by Delson and West [19–21] provides a useful insight when operating within the plane. This approach relies on the teacher, when providing examples, to ensure that all examples are *homotopic* (i.e. the path followed goes on the same side of each obstacle in each case). Thus, only the path of the end-effector is guaranteed collision-free, and only when the user provides the correct set of examples. This collision-free space is used to find the shortest possible path from start to goal. A potential extension to three dimensions is also presented [20]. Our work extends some of these concepts into arbitrary-dimensional spaces.

If prevention of collisions is not an issue, or is handled by some other mechanism, there may still be other reasons to allow a human teacher to review and reject or critique robot-generated trajectories. Argall [3] presents a method to improve upon a simple nearest-neighbor algorithm using feedback from a human. Portions of a trajectory already executed by the learner can be marked as bad or inefficient. The distance metric used to determine neighbor trajectories is then altered to suppress the reuse of the poor example trajectory used in that instance. When a learner can detect its own lack of knowledge in certain areas, feedback can be requested in the form of additional motion examples. Some of these feedback techniques could be applied to our approach, as well, as discussed in future work. Where Stolle [69] relied on a motion planner to generate the new examples, Chernova [15] presents a similar technique relying on a human, instead. In this work, Gaussian Mixture Models

(GMMs) are used to estimate the confidence of the learned policy over the state space. Work by Grollman and Jenkins [27], using the framework of Mixed Initiative planning, allows the robot to indicate to the teacher areas of uncertainty where additional examples could improve its policy. In these cases, the uncertain (potentially non-Markovian) areas are resolved by asking the teacher for additional information, or allowing the robot learner to choose among the demonstrated actions. Similarly, our neighbor graph and bifurcation detection allow us to find areas in which conflicting information has been provided by the teacher and to request clarification.

Mayer [45] investigated modelling trajectory learning using fluid dynamics. The idea is that examples 'stir' the fluid, inducing a vector field of trajectories that guide the robot's motions in a smooth way. This formulation also permits subtle corrections when additional examples are provided. The researchers also present a useful approach to pruning unnecessary portions of the example trajectories (due to jitter or other human inefficiency) and for decomposing examples into meaningful segments.

The learner's performance may be further improved by incorporating feedback from the user during execution of learner-generated trajectories. In this case, the problem more closely resembles reinforcement learning, since the teacher is providing (positive and negative) rewards at particular states during execution. The problem may be formulated as Bayesian inference [17] or a Markov Decision Process [78], or more frequently as a Q-learner [79], such as by Argall [3] and Bentivegna [9].

In applying reinforcement learning techniques to an implemented system, it should be advantageous to keep in mind the social aspects of the interaction between the teacher and the learner [43]. In addition, Thomaz [72] offers guidance on methods that human teachers naturally apply to guiding a learner.

2.4 Symbolic Programming by Demonstration

In addition to the low-level task of generating trajectories from examples, other researchers have investigated higher-level forms of planning that

reuse trajectories, or portions of trajectories, and combine them in new ways. The primitives, or individual learned elements, can be trained individually, or extracted (manually or automatically) from more complex examples.

For example, Atkeson and Bentivegna present [10] an air-hockey-playing robot that learns to combine primitives such as "left hit", "right hit", and "block". These primitives are specified by the programmer and recognized by the robot during learning. Because the robot can evaluate the performance of each primitive toward reaching the overall goal, the robot is capable of improving its performance of each primitive beyond any training example provided to it. Thus, the robot is able to improve performance beyond that of the teacher.

Lent [77] observes that manually dividing the demonstrations into stages may be necessary for learning complex behaviors. Depending on the stage of the task, humans may react differently to the same sensor input. Since the stage of the task may not be directly observable from sensor data, the teacher must specify semantic divisions in the task. This formulation hints at the applicability of a Hidden Markov Model for learning appropriate actions. The current stage, or primitive, is the hidden state, while any available sensor readings may be observed. This approach is explored by Hovland [30] in the framework of a hybrid dynamic system.

While these works have assumed that the primitives are specified by the human and marked for each example, there has also been work in which the robot learns the primitives itself. This leads to a wider field of learning from observation in which perspective taking [73] becomes important. This is a principle by which the robot interprets examples from the perspective of the teacher. This strategy can help resolve ambiguities in human demonstrations. Cynthia Breazeal and her students have investigated methods for estimating the teacher's belief state during learning to facilitate perspective-taking [46]. Work in the development of a model of working memory for robots [64] also helps establish a framework for communication between the robot and the human teacher.

Other works separate the idea of *what* to do from *how* to do it. Iba and colleagues present [49] a method for learning both the task and the

low-level behavior necessary to perform it. Iba's dissertation [31] describes the human-robot interaction necessary for an unskilled user to both teach behaviors and combine them into new task skills.

However, additional example trajectories do not simply increase the knowledge at their initial location. The new trajectories will traverse additional portions of the configuration space, providing the learner with examples of desired behavior in these areas, some of which may be previously unvisited. In many domains, it is advantageous to choose examples which will maximize the unknown areas visited by each example so as to gain as much new information as possible from each teacher demonstration. The active optimization problem of maximizing the value of each new example while limiting the total number of examples has been studied in some depth [47,53]. In our domain, decision-tree-based methods [15,58] have had some success in guiding learners.

2.5 Dimensionality Reduction

A particular strategy for programming by demonstration that has received some recent interest is dimensionality reduction [25]. This technique involves computing a mapping from some high-dimensional input space to a lower-dimensional space in which learning is simplified. This is important even when the input space is simply the joint positions of the robot arm itself: a seemingly minimal representation. However, redundancy and unused degrees of freedom can complicate the learning procedure, and there is typically structure in the motions of a task that can be captured by a low-dimensional embedding.

The advantages of this approach are most obvious in situations in which the dimensionality of the input space is large. In fact, this is an important step in most works that use human motion capture to train robotic manipulators. Motion capture may use dozens of visual markers on the human body, inertial sensors, or even low resolution images directly from the vision system. These systems may be used to train a robot with six or fewer degrees of freedom. If only the end effector position is to be learned, there must be a mapping from the motion capture examples to a lower number of dimensions used for control while pre-

serving necessary information. The search for this mapping is known as the correspondence problem [18], and is important even in understanding how humans and animals recognize and learn from biological motion [55]. Schaal, Ijspeert, and Billard present [59] an excellent survey of techniques and formal frameworks for establishing a correspondence between a learner's available actions and the teacher's demonstration.

Dimensionality reduction may be equally useful, though, even when the input space is equivalent to the command space for the robot. That is, even if the training data consists of teleoperation commands (or positions) for the robot itself, learning may be improved by discovering an intrinsic mapping of robot position to a space with lower dimensionality. When successful, this technique eliminates much of the noise or jitter that distinguishes the examples because the lower-dimensional space lacks the degrees of freedom to represent all of the uncorrelated jitter across trajectories. However, small movements of the same order of magnitude as the noise, but which are necessary for the task, are preserved through the mapping because they will appear in all examples. In contrast, alternate smoothing procedures that examine a single trajectory at a time cannot determine which small-scale features are shared across examples.

One promising technique for establishing this low-dimensional mapping is ST-Isomap [34,50], an extension of the original Isomap algorithm [71] for handling spatio-temporal (time-series) data. This approach has also been extended to include both position and other sensor data as input on the Robonaut platform. The result is used to distinguish between successful and unsuccessful teleoperations of a given task [50]. When used with pre-segmented examples, the technique can help build a library of motion primitives [23,33] for use in constructing higher-level skills. The insights of Isomap and ST-Isomap helped guide the work of this thesis, and are explored more fully in Chapter 3.

Another technique used on Robonaut was briefly mentioned earlier in [13]. A technique originally developed in the graphics community, known as Verbs and Adverbs [56], is used to represent actions, or verbs, in a low dimensional space. Adverbs are used to parameterize specific features of the action, such as the final position of the end effector. The

adverbs affect the mapping of learned behaviors back to control space. Like the symbolic approaches described in Section 2.4 or the primitive-based approaches in Section 2.2, this technique involves an intermediate representation that constrains portions of the demonstrations and plans, and organizes them into categories. This use of categories is a trade-off. If the categories are not well-chosen, certain trajectories cannot be faithfully represented in this system. However, by grouping portions of trajectories into qualitatively similar categories, planning by interpolation within a single category may be simplified due to constraints on the demonstrations that belong to a single category. This could have benefits when trying to detect an objective function being optimized by the demonstrations, as described in our future work.

2.6 Summary

Programming by Demonstration is an effective approach for conveying skills to a robot learner, and much work has been devoted to the use of these techniques for teaching motion skills. Various techniques based on Supervised Learning are used to relate multiple task demonstrations to one another and to imitate the behavior of the teacher. Dimensionality reduction provides an intriguing method for representing the correspondences between examples such that unwanted noise is reduced. This thesis extends previous work in this area to produce smooth plans consistent with provided demonstrations. Other critical issues addressed by PbD techniques include the generalization of learned behaviors to undemonstrated and uncertain areas and incorporation of additional feedback from the teacher beyond an initial set of demonstrations.

Behaviors are typically generalized through blending similar demonstrations. This entails the (usually implicit) assumption that all the blended demonstrations are compatible, that is, that they represent the same strategy. This assumption is made explicit in one work [20], which requires the user to assert that all example trajectories are homotopic. This helps ensure that the plans generated by the learner safely avoid obstacles in the environment. In this thesis, we introduce algorithms to automatically distinguish examples that may be safely interpolated.

Additional feedback from the teacher can provide correction to the learner's behaviors or additional confidence in uncertain areas. This feedback may take the form of scores or corrective suggestions to plans or portions of plans generated by the learner. Feedback may be initiated by the teacher or solicited by the learner if it is able to determine parts of the state space in which it has insufficient or inconsistent examples. Similarly, the learner presented in this work detects areas of diverging strategies in the demonstration trajectories, and requests clarifying advice from the teacher.

Chapter 3
Background: Dimensionality Reduction

You have been scanned, assessed, understood, Doctor.
Your limits and capacities have been extrapolated.

— Supreme Dalek and Cyber Leader,
"The Pandorica Opens", *Doctor Who*, 2010

Dimensionality reduction is a useful technique in many learning settings, including programming by demonstration. Although data may be naturally presented to a learning algorithm in a high-dimensional space, features relevant to the problem at hand may occupy only a few (often linearly separable) dimensions. Removing irrelevant information and creating a simpler representation of data can help us understand how best to make use of the information presented.

In this chapter, we provide some background on the dimensionality reduction problem and its potential application to time-series data. Next, we discuss our PbD system for learning and planning in reduced-dimensionality spaces. Finally, we identify some problems inherent to this approach and consider alternate strategies.

3.1 General Technique

Dimensionality reduction is a general class of techniques for finding a new representation of data, generally in a lower-dimensional space, while retaining its essential characteristics. This mapping to the new space is known as an *embedding*, though the term can also refer to the new representation of the data itself. The new space is known as a *latent* space, because it is inherent to the data.

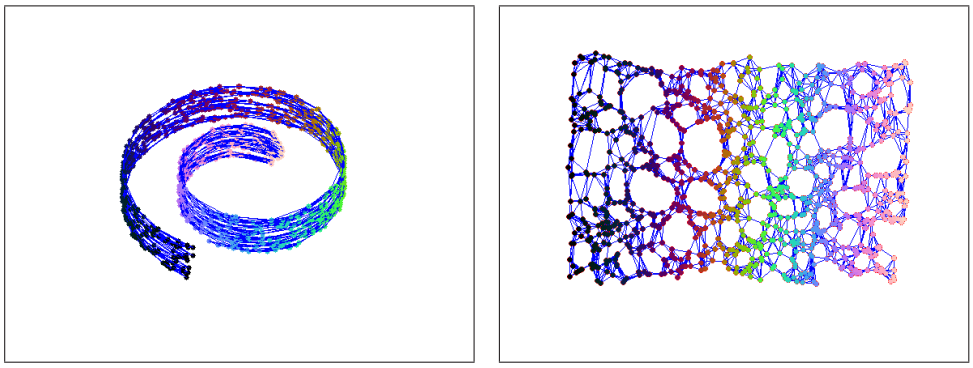


Figure 3.1: The Swiss roll dataset in three dimensions, and the two-dimensional embedding discovered by Isomap. Blue lines represent neighbor links used to determine geodesic distances.

Despite its name, dimensionality reduction does not always require that the embedding reduces the number of dimensions; in some cases, the data is simply transformed in order to produce a representation that emphasizes certain qualities. For example, linear embeddings such as Singular Value Decomposition (SVD) [39] create a new set of orthonormal basis vectors. SVD orders these vectors so that decreasing proportions of the variance in the data is contained in each subsequent dimension. Thus, simply ignoring the highest dimensions of the transformed data does not lose much information, but allows a simpler representation.

Global linear transformations create a new representation of the original space, but other techniques can discover latent spaces tuned more specifically to particular dataset within these spaces. For example, Isomap [71] seeks to discover lower-dimensional manifolds within higher-dimensional data. This algorithm assumes that the relationships between nearby (neighboring) points are significant, and attempts to preserve distances between them. Points that are distant in the original space are related not by the linear distance separating them, but by the distance through sets of neighboring points. Isomap thus builds a *neighbor graph* relating the data and calculates *geodesic* distances between all pairs of points. The geodesic distance between two points is the shortest dis-

Algorithm 3.1 Isomap

```
1: function ISOMAP(pts, ε)
2:     ▷ Construct neighbor graph
3:     Graph G_N
4:     for all i ∈ pts do
5:         for all j ∈ pts do
6:             if DISTANCE(i, j) < ε then
7:                 G_N(i, j) ← DISTANCE(i, j)
8:             else
9:                 G_N(i, j) ← ∞

10:    ▷ Compute all-pairs shortest paths
11:    Graph G_D
12:    for all i ∈ pts do
13:        G_D[i] ← DIJKSTRA(G_N, i)

14:    ▷ Create embedding
15:    Points y
16:    λ, v ← EIGS(G_D)                ▷ Sorted in decreasing order
17:    for all i ∈ pts do
18:        for all p ∈ 1..pts.dimension do
19:            y[i][p] ← √λ[p] v[p][i]

20:    return y
```

tance through the graph, where distances along neighbor links are calculated using a conventional distance metric (e.g. Euclidean). The embedding preserves these pairwise distances as best as possible in a lower-dimensional space.

Isomap is frequently illustrated using the Swiss roll dataset in Figure 3.1. The data consists of a plane "rolled up" in three dimensions. We say that the data is intrinsically two-dimensional, but is embedded in three dimensions. The Isomap algorithm is able to recover the intrinsic representation through the procedure detailed in Algorithm 3.1. First, the neighbor graph G_N is constructed with links between nearby points. The Isomap variant listed here, called ε-Isomap, uses the parameter ε as a distance threshold for considering two points to be neighbors. Alter-

nately, K-Isomap uses an integer parameter k, such that the k points closest to each point are linked in the neighbor graph. In either variant, distances between non-neighboring points are set to ∞ in G_N. Next, a complete graph G_D of geodesic distances is computed from the neighbor graph. In the listing above, the algorithm computes one row of the distance graph at a time by using Dijkstra's single-source shortest path algorithm [22]. Finally, it calculates the eigenvectors v of the distance graph, and new coordinate vectors y, which correspond to the original points pts in a new Euclidean space. The first d dimensions of this space provide the d-dimensional embedding that best preserves the distances in G_D.

Returning to the Swiss roll example, we see that the "unrolling" of the intrinsically two-dimensional data depends on the creation of a proper neighbor graph. This graph should provide geodesic distances (in the original high-dimensional space) that closely approximate the Euclidean distances between points in the low-dimensional latent space. Thus, the value of the parameter ε or k is critical. If this value is too large, neighbor links will be formed between layers of the swiss roll. The effect of this problem is explored later in Section 3.4.

3.2 Trajectory Embedding

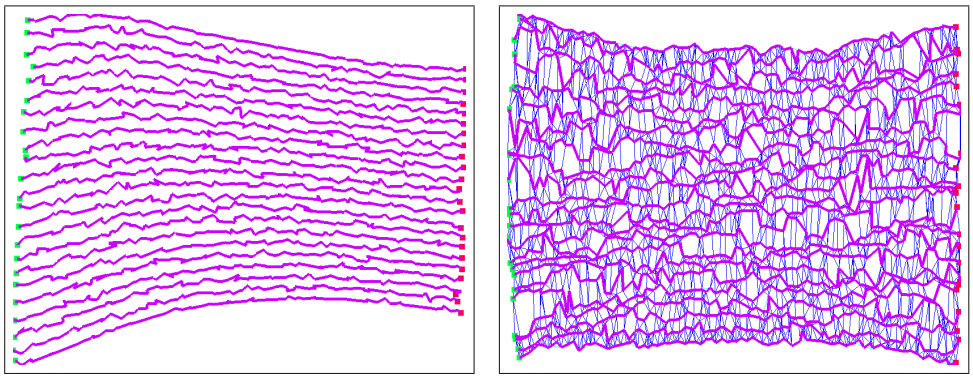
Dimensionality reduction is typically used in Programming by Demonstration to deal with the correspondence problem [18] between high-dimensional training data (such as human motion capture) and low-dimensional controls (such as dexterous arm joint angles). When kinesthetic demonstrations are used, the training data is collected in the robot's control space. However, finding an intrinsic low-dimensional representation of the data is still attractive, as it accentuates commonality among multiple training examples, and eliminates much of the noise (due to imperfect sensors) and jitter (due to imperfect human motion) that needlessly distinguishes them. Smoothing is achieved because the lower dimensionality space lacks the degrees of freedom to precisely represent every aspect of the example trajectories. Thus, the high frequency noise is eliminated while the features common to all examples are emphasized.

This strategy is to be favored over other techniques that smooth single trajectories at a time by removing high-frequency or low magnitude variations. Although neither approach requires domain knowledge for its application, dimensionality reduction is able to preserve features common to many trajectories, even at the same magnitude as the noise, because it operates on all the trajectories at once, recognizing and preserving common features.

In addition, we expect that most motion tasks are inherently low-dimensional. A single trajectory is a linear sequence of points. The positions of these points may be recorded in a configuration space of arbitrary dimensionality, but a single dimension, time, is sufficient to specify a particular element in the trajectory. Additional demonstrations of the same motion introduce some variation. If these demonstrations are constrained (or scaled) to the same period of time, then we can examine the variation between examples at any given point in the time dimension. Conceptually, this representation of the collection of demonstrations is an embedding in two dimensions: time and variation between demonstrations. Experiments conducted during our own early work, as well as by other dimensionality-reduction researchers [74], indicated that two dimensions were usually sufficient to represent manipulation tasks occurring in six to eight dimensions.

Spatio-Temporal Isomap (ST-Isomap) [34], is an extension of the original Isomap algorithm that exploits the structure of time-series data. Instead of simply treating the input data as a collection of points, ST-Isomap attempts to account for relationships between subsequent points in a single trajectory (temporal neighbors), and similar points in different trajectories that correspond to one another (spatio-temporal neighbors) by decreasing the perceived distance between these types of neighbor points. Figure 3.2 illustrates some embeddings of trajectories that traverse the Swiss roll dataset. The knowledge that the points are contained within trajectories provides some prior information to help build neighborhoods for each point. Points are connected to their predecessors and successors, but not to more distant (in terms of time) points within their own trajectory, even if they are nearby in three-dimensional space.

This use of trajectory information often permits dimensionality re-

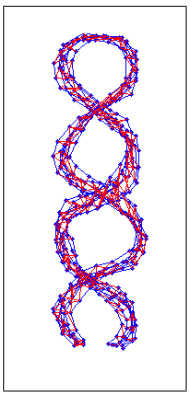


(a) (b)

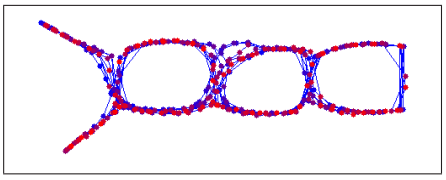
Figure 3.2: Embeddings of trajectories traversing the Swiss roll manifold. (a) Trajectories with a small amount of jitter constrained to the 2D manifold. (b) Trajectories with 3D noise added and the neighbor links (blue) used for dimensionality reduction.

duction to reorganize data in a structure more conducive to planning. The slalom dataset shown in Figure 3.3(a) could present a challenge for many PbD techniques in its original representation. The ST-Isomap embedding does not change the dimensionality of the data, but reorganizes it to make planning much easier. Finally, a good dimensionality reduction technique is capable of smoothing jitter and noise in trajectories without removing significant common features of the same order of magnitude. This is especially important for areas of application such as tight-tolerance assembly work.

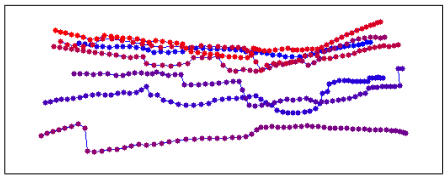
Planning in a low-dimensional latent space is attractive because the arrangement of trajectory points in this space should have semantic meaning. Since the dimensionality reduction algorithm attempts to preserve distances between neighboring points (precisely those points that we declare to be semantically similar) the latent space provides a representation of the trajectories inherent to the task itself. For example, the trajectories of Figure 3.3(a) cross over themselves in workspace, but are "unrolled" in the latent space (Figure 3.3(c)). The start and goal points, previously close to one another, are now at opposite ends of the horizontal axis. Thus, this dimension represents progress through the task, while



(a)



(b)

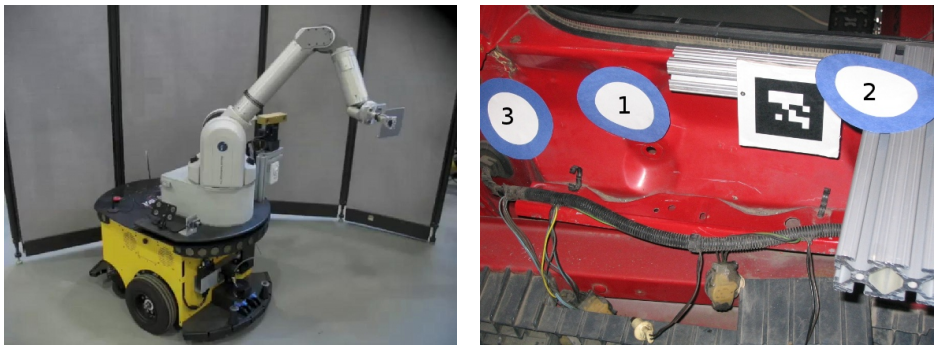


(c)

Figure 3.3: The *slalom* dataset, shown to the left, exemplifies the strength of the ST-Isomap algorithm. The trajectories begin and end at the bottom of the image, and cross left and right as they move to the top of the image and back to the bottom. The Isomap embedding in (b) suffers from an inability to distinguish different parts of the trajectories when they cross one another. The ST-Isomap embedding in (c), however, presents an intuitive representation in which the horizontal axis approximates time, and most of the variation between examples is in the vertical axis. The abrupt vertical jumps within trajectories are likely artifacts due to the discrete nature of the neighbor relationship between trajectories.

the vertical dimension separates distinctions between demonstrations. Trajectories cross one another in the latent space as they move closer and farther from one another in the original space. Planning through this space would seem to be intuitive since movement in each direction has a semantic explanation. However, noise in the neighbor links does produce some artifacts in the embedding. When neighbor links between pairs of trajectories appear and disappear over time, discontinuities in the vertical dimension of the latent space appear. As discussed in the next section, this causes similar discontinuities in plans produced in this space.

To illustrate the various dimensionality reduction techniques introduced so far, we introduce a simple example task for creating high-dimensional trajectories. The Barrett WAM 7-DOF dexterous arm shown in Fig-



(a) (b)

Figure 3.4: The robot (a) was used to reach out and contact each of the targets placed on the car bumper (b). The resulting trajectories will be used to illustrate dimensionality reduction techniques.

ure Figure 3.4(a) was used to demonstrate a reaching task depicted in Figure 3.4(b). When the arm is operated in gravity-compensation mode, it can be easily moved by hand. We recorded the position of all seven joints during a contrived task, in which the teacher was asked to stretch out the arm to touch 3 nearby points, with the end-effector in a different orientation each time. Figure 3.5 shows the two-dimensional singular value decomposition (SVD) projection of several example trajectories demonstrated by a human operator. The blue traces in the plot show the first two dimensions of the example trajectories after SVD transformation. We operate in joint space because the arm is redundant, and we should be able to distinguish between different configurations that result in the same end-effector pose.

Examining the SVD projection in Figure 3.5, we see that the trajectories are grouped together well, but different portions of the trajectories overlap. Since touch points 1 and 3 are close in the workspace, they also appear close in joint space, even though the points are always visited at different times during the execution of the task. This can complicate planning since the joint angles alone do not provide enough information to determine the desired motion to execute. Thus, the task in non-Markovian with respect to configuration space. The two-dimensional embedding created by conventional Isomap (Figure 3.6(a)) stretches most

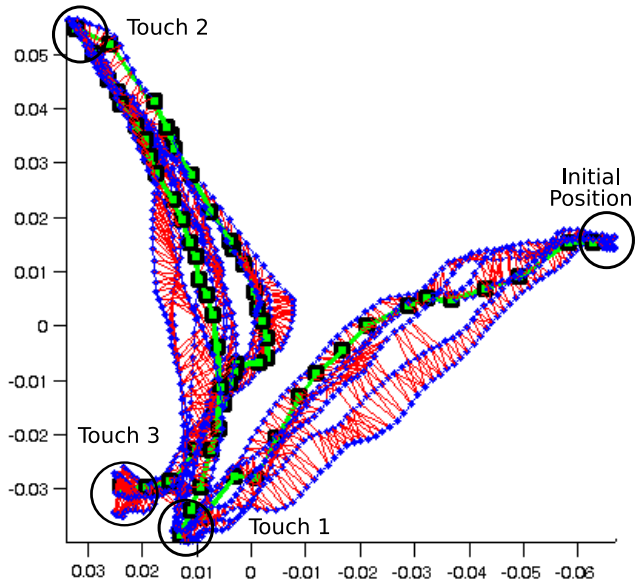
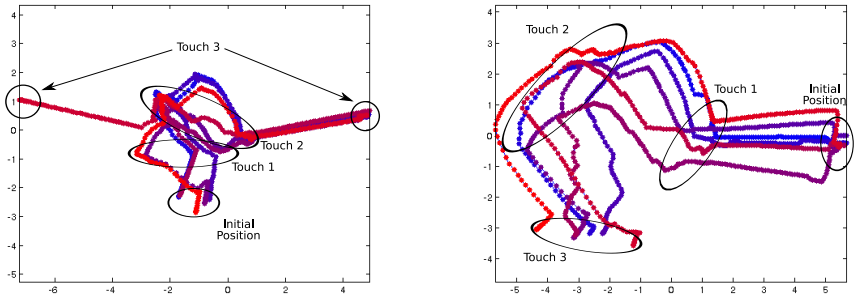


Figure 3.5: The 2D SVD projection of sample trajectories represented by thick blue lines. The thin red lines represent neighbor links determined by ST-Isomap. A planned path is shown by the larger, green squares.

of the examples so that they do not overlap, but still does not provide a suitable space for planning, because the final position of one of the trajectories is far removed from the others and the touch points are not well-distinguished.

The neighbor assignment mechanisms used by ST-Isomap (represented by the red lines in Figure 3.5) help to overcome the problems of the previous two techniques. The ST-Isomap embedding in Figure 3.6(b) is able to distinguish between points in different parts of the task, and the initial and final positions of all examples are reasonably well grouped. The non-linear embedding has transformed a non-Markovian joint space into a Markovian latent space. This provides a more useful representation for planning since all trajectories follow similar courses through the embedding space and the separation between them is a representation of the inherent variations between examples.



(a) (b)

Figure 3.6: The two-dimensional embeddings created by (a) Isomap and (b) ST-Isomap.

3.3 Planning and Executing

Having constructed a low-dimensional latent space, we can now consider using it to create plans. Of course, trajectories created in this two-dimensional space must be transformed to the seven-dimensional configuration space before they can be executed on the robot. However, there is no unique mapping between points in these spaces. Instead, individual points in the planned path must be *lifted* to the original high-dimensional space. This can be accomplished using the Delaunay triangulation [11] (see Figure 3.7 of the points in two-dimensions). For any query point in the plane, the enclosing triangle is found. Although the vertices of this triangle may not be joined by neighbor links, their proximity in the embedding corresponds to their geodesic distances. Thus, we expect the points to be quite close in the neighbor graph. The barycentric coordinates of the query point within the triangle are used as weights to interpolate between the points corresponding to the triangle vertices in the original space. It should be noted that the mapping from the planning space to the original space is naturally a one-to-many mapping. However, since the correspondence between points in the planning space and the original points in the higher dimensional space is known, interpolation ensures that the range of the query point is in the correct region of the configuration space. However, this method is reliable only

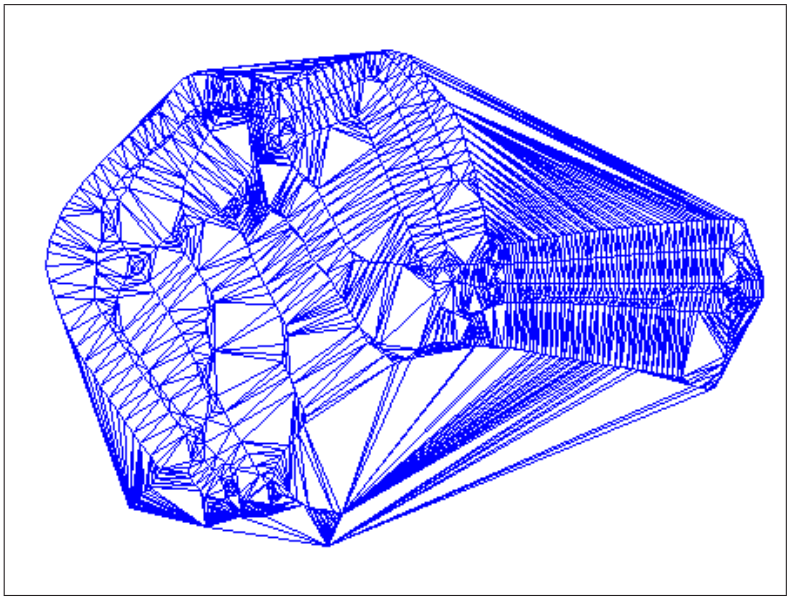
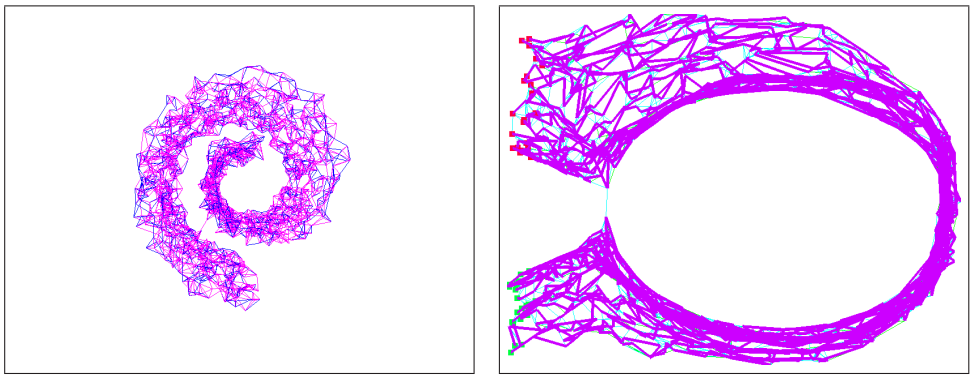


Figure 3.7: The Delaunay triangulation of approximately 1300 points in 6 example trajectories from a 7 dimensional configuration space. The ST-Isomap embedding is shown in Figure 3.6(b).

for query points that lie within the triangulation of the points of the example trajectories. Fortunately, this is precisely the area in which we can be confident executing novel plans. Points outside the triangulation must be outside the planned regions, and thus represent extrapolations outside the demonstrated area of the configuration space.

A similar method may be used to map points in the other direction, from the configuration space to the planning space. This mapping is required to query the plan for an action to perform at a given configuration. Again, the query point should lie within the collision-free area of the space. Using the correspondences already known between points in both spaces, we again interpolate between neighbors found in the configuration space. These neighbors may be found through conventional nearest-neighbor approaches, though knowledge of the task structure may help resolve ambiguities. For example, if the task always begins in the same region of space, the neighbors for an initial query may be fixed as the set of initial points from demonstration trajectories.



(a) (b)

Figure 3.8: A single spurious neighbor link can have a substantial effect on the quality of the embedding. (a) The 3D Swiss roll with significant noise added. A naïve neighbor-finding technique creates a spurious link across layers of the roll. (b) The resulting embedding.

Dimensionality reduction also offers a convenient metric for the quality of the embedding produced. For tasks that are inherently two-dimensional, we expect a low residual variance, the portion of the variance within the dataset that is lost by the embedding. This measure is typically used to compare embedding algorithms because, given identical data points and neighbor graph as input, the algorithm that produces a lower residual has the better embedding. However, if we apply the same algorithm to a dataset with multiple neighbor graphs, the residual represents the quality of the graph. The neighbor graph that best captures the inherent two-dimensional structure of the data will produce the lowest residual. Section 5.4 presents residuals for some task embeddings.

3.4 Inherent Difficulties

Unfortunately, these and similar approaches to dimensionality reduction are particularly susceptible to errors caused by spurious neighbor links. While these algorithms are robust to even a large number of false negatives (that is, neighbor links missing where they should be present) even a single false positive link can have disastrous effects on the em-

bedding [74]. Spurious neighbor links create "short circuit" connections between unrelated portions of trajectories, thus lowering the geodesic distances between points in these semantically distant regions. Consequently, the embedding algorithms try hard to retain the proximity of these regions in the low-dimensional latent space, a space in which distance should have semantic relevance.

Additionally, *lifting* a novel trajectory from the latent space is likely to produce second-order discontinuities. Unfortunately, a non-linear embedding algorithm such as Isomap does not guarantee smoothness of lifted trajectories, even when points are densely sampled from a smooth latent-space trajectory. This is because the neighbors used to interpolate a lifted point are a small subset of the nearest points from the example dataset. Two points may be arbitrarily close in the latent space while still having different nearest neighbors. Although we expect these neighbors to be linked in the neighbor graph, they are not necessarily close in the original space, so discontinuities result. Moreover, because the dimensionality reduction is not *injective*, there can be ambiguities in lifting a point from the planning space back to the original space. That is, multiple regions of the configuration space can map to the same regions of the lower-dimensional planning space. These problems are actually exacerbated as additional example trajectories are provided to the learner. More examples produce a more cluttered latent space with more interconnections between neighbor points. This creates additional opportunities for discontinuities in lifted trajectories. Clearly, decreasing performance with an increasing amount of information is not a desired attribute of a planner.

Chapter 5 describes our neighbor-finding technique, which improves upon ST-Isomap in representing the semantic structure of trajectory data. Although this approach mostly avoids the problem of spurious neighbor links through successful use of heuristics, it is still susceptible to the discontinuities and ambiguities of lifting from a low-dimensional space. The discontinuity problem is clearly illustrated in one of our experimental domains: the wire maze. The experimental setup is pictured in Figure 3.9. Participants were asked to guide a 7-degree of freedom Barrett WAM arm through a wire maze. Kinesthetic demonstrations

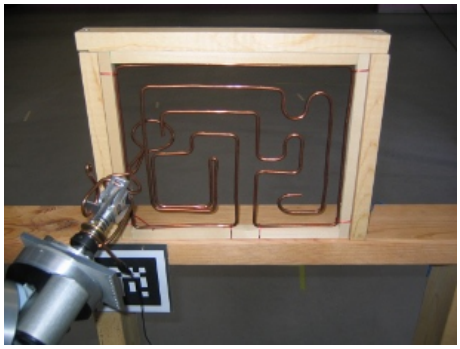


Figure 3.9: The WAM manipulator traversing a wire maze. The 2-D barcode in the lower-left is a visual fiducial used to detect the location of the rig relative to the robot.

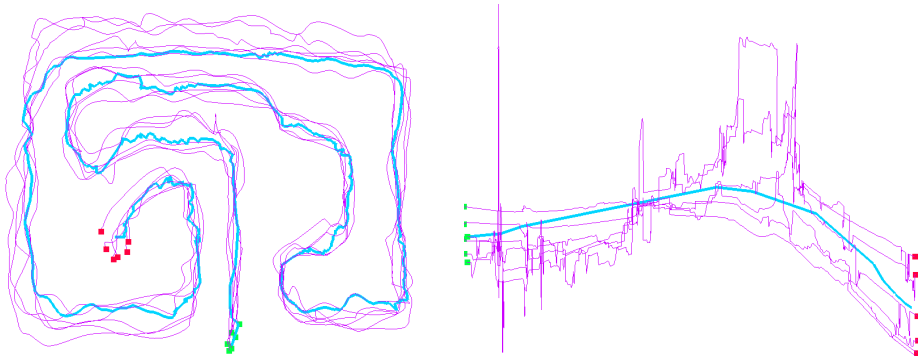


Figure 3.10: Traces of the end-effector position while traversing the wire maze, shown in the workspace (left) and in the reduced dimensionality space (right). The light blue trace is a naïve plan created in the reduced dimensionality space, then lifted to the original workspace.

were performed by placing the arm in a passive gravity-compensation mode, a state in which the arm responds to physical manipulation, but otherwise holds its position. Figure 3.10 displays workspace traces of six demonstration trajectories (in purple) from one of the participants, beginning at the green points near the bottom of the image and ending at the red points. The latent space embedding created by our algorithm is shown on the right. As expected, this embedding essentially "unrolls" the trajectories, stretching them out so that task time, or pro-

gression through the maze, occurs from left to right on the horizontal axis. The vertical axis provides a dimension for variation between examples. Again, vertical discontinuities appear in the embedded demonstrations due to noise in the neighbor graph. A few missing links between nearby demonstration trajectories will cause increased geodesic distances between points in these trajectories, and force them away from one another in the embedding. The blue line in this image represents a candidate plan, created as a series of line segments stretching from the start to the goal in the latent space. When this plan is lifted back to the original workspace in the left image, discontinuities result. These discontinuities are especially evident in some of the areas of horizontal motion near the center of the maze: vertical zig-zagging motions appear in the plan where none are present in the demonstrations. Although a post-processing step could be applied to smooth the planned trajectory, this approach should be avoided. Although the zig-zagging discontinuous plan is undesirable, it is not clear that a smoothed version will follow the nuances of the demonstration trajectories. Instead, we must consider a new approach to planning that directly produces useful plans in the target space. In Chapter 6, we present a planning algorithm that operates directly over the neighbor graph, thus avoiding the difficulties of the latent space.

3.5 Summary

Dimensionality reduction is a powerful technique for discovering the inherent structure of high-dimensional data and representing it in a more intuitive manner. In addition, the structure provided by the prior knowledge that a collection of points actually represent a series of trajectories can greatly assist general-purpose dimensionality reduction algorithms, allowing them to create embeddings apparently well-suited to planning. Unfortunately, small errors such as a single spurious neighbor link can upset this embedding, and trajectories created in the latent space require post-processing after transferring to the original space. For these reasons, dimensionality reduction appears to be ill-suited for Programming by Demonstration. However, the structure constructed by

the trajectory neighbor-finding still provides useful semantic information about the demonstration trajectories and the space they occupy. In future chapters, we will develop an approach to Programming by Demonstration that makes use of these neighbor relationships to develop an understanding of demonstrations provided by a human teacher and to plan novel trajectories.

Chapter 4
Approach

Just what do you think you're doing, Dave? Dave, I really think I'm entitled to an answer to that question.

— HAL 9000, *2001: A Space Odyssey*, 1968

4.1 Overview

This thesis proposes an algorithm allowing a robot, such as a manipulator arm, to learn to perform kinematic movement tasks through unmodelled space. The robot learner will observe example trajectories performed by the teacher and ask clarifying questions as necessary, then generalize those examples to plan novel trajectories that are safe and respect the constraints contained within the examples.

Our approach to programming by demonstration begins with a collection of example trajectories. In this work, we create example trajectories using *kinesthetic* demonstrations. That is, the teacher directly manipulates the robot to move it through the task. Other PbD approaches may operate on trajectories created by another agent, such as motion capture of humans performing the task. However, these approaches must contend with errors introduced by sensing. Additionally, there is the problem of finding correspondences between the different kinematic models of the teacher and the learner. Kinesthetic demonstrations avoid these possible sources of error.

Our neighbor-finding algorithm provides structure to the example trajectories by constructing a neighbor graph connecting discretely sampled points between examples. Because human demonstrations are im-

perfect, the learner will need to resolve inconsistencies, imperfections, and noise in the example trajectories. A collection of heuristics is able to resolve these problems by analyzing relationships between sample points.

When learning is complete, the robot is capable of planning new paths that safely execute the task demonstrated by the teacher. We do not learn a policy because the space in which the learner operates (configuration or workspace) tends to be non-Markovian. That is, the same areas of the planning space may be visited more than once during the execution of the task, and different actions must be executed each time. Instead, new plans are created by following the neighbor graph and interpolating between nearby points. Interpolation allows new plans to smoothly follow the strategy demonstrated by the teacher, but to avoid jitter and other inconsistent movements that would be duplicated if the agent simply followed an example trajectory.

Finally, some areas of conflicting information, namely bifurcations, in the neighbor graph, require additional advice from the teacher. In these cases, the learner requests clarification by asking the teacher which branch is preferred. In our approach, the learner generates and performs trajectories following each branch of the bifurcation and asks for the teacher's preference.

4.2 Neighbor Graph

The *neighbor graph* is created to provide a structure, revealing relationships between configurations visited by the robot during the training process. This structure provides a means for the learner to understand the task at hand by connecting similar portions of the task demonstrated in multiple trajectories. It also provides a framework in which to plan novel trajectories by interpolating between neighboring example points.

More formally, a *trajectory* is represented as a sequence of points traversed by the robot. A *demonstration* trajectory consists of points sampled from the path the robot traversed while under the control of the human teacher. We sample points at a regular distance interval along the path, as this allows us to control the size of features represented in tra-

jectories at the cost of computational speed and storage space. A teacher provides multiple demonstrations of the motion task to be learned, and these demonstrations are combined to form the neighbor graph.

In terms of notation, individual trajectories will be represented by lowercase variables, and ordered collections of trajectories will use uppercase. In both cases, zero-based array indexing is used to refer to points: $t[i]$ refers to the i^{th} point in trajectory t, and $T[i]$ refers to the i^{th} point in a concatenation of all the points in the trajectories in T. The typical double-bar norm denotes the number of points in a trajectory or collection, while the single-bar norm denotes the number of trajectories in a collection. Subscripting is used to refer to a trajectory within a collection. So, the equation:

$$\|T\| = \sum_{i=0}^{|T|-1} \|T_i\|$$

states that the number of points in the trajectory collection, T, is equal to the sum of numbers of points in each of the trajectories it contains.

The neighbor graph is a weighted, directed graph formed by connecting nearby points within and between trajectories. The vertices of this graph consist of all the points in all of the demonstration trajectories. The edges, or *neighbor links*, connect these points and fall into two categories: intra-trajectory links L_a, and inter-trajectory links L_e. In both cases, the edge weights are defined by an appropriate distance metric between the two points; in the 3-dimensional workspace illustrated in this work, we use Cartesian distance.

The set of intra-trajectory edges is formed by linking subsequent points within each individual trajectory. Since a weighted graph edge can be defined as an ordered triplet of source vertex, destination vertex, and weight, the intra-trajectory edges may be defined by:

$$L_a = \bigcup_{i=0}^{|T|-1} \bigcup_{j=1}^{\|T_i\|-1} \{(T_i[j-1], T_i[j], \mathsf{dist}\,(T_i[j-1], T_i[j]))\} \tag{4.1}$$

The second set of edges, those linking similar points between tra-

jectories, are more complex, and are the focus of Chapter 5. Although inter-trajectory links are symmetric, we use directed edges for consistency with intra-trajectory links, which are ordered by time. We simply add one edge in each direction for entries in L_e. Using standard graph notation, the neighbor graph, G, is then defined by the combination of the trajectory collection T and the *adjacency list*, or list of edges L, which is the union of L_a and L_e.

$$\begin{aligned} G &= \{T, L\} \\ &= \{T, L_a \cup L_e\} \end{aligned}$$

The inter-trajectory links, L_e, are based on each point's nearest neighbors in configuration space, but several heuristics are applied to ensure that these links connect points representing similar locations in *task space*. That is, the points should occur during semantically similar times during the execution of the task. The heuristics used are primarily based on the types of imperfections expected to be present in human demonstrations. For instance, even when a human attempts to produce a smooth arc, jitter in the person's movement (or noise in the robot's sensors) will produce undesired deviations from the nominal curve. Larger imperfections such as false starts, retracing previous actions, and unintentional divergences create imperfections in the examples that the robot should not follow. Rather than asking the teacher to mark or edit these undesired sections of demonstration data or, worse, throwing out the entire demonstration, our neighbor-finding heuristics allow the learner to detect and cope with these problems. The nearest-neighbor graph provides an estimate of which sections of other example trajectories correspond to these problem areas because such sections will have few (if any) neighbor links joining them to other demonstrations. When new plans are created, the undesired deviations are ignored, and the learner follows the nominal behavior.

Although many of the imperfections that the learner is attempting to avoid may appear as noise or unnecessarily long routes between two points, path smoothing and other forms of optimization are not viable approaches for cleaning up demonstration trajectories. There is no sin-

gle objective function applicable to all tasks. A small deviation from a straight-line path may appear to be caused by jitter, but if this deviation occurs in all demonstrations, the learner should consider it a necessary behavior in newly planned paths. Likewise, the learner should not attempt to plan shorter paths, even when it is certain that no obstacles are present. Non-geometric constraints, or other concerns, may have influenced the teacher's choice of paths. Following the example trajectories is the only way to optimize for the unknown objective that the teacher is demonstrating. However, we are able to improve upon individual examples by understanding the correspondences between multiple examples. Deviations that appear in only one example end up being uncorrelated to others in the neighbor graph, and so are likely to be mistakes. Essentially, the neighbor graph allows us to interpolate between the best portions of all the example trajectories.

4.3 Planning

Planning is performed by following the neighbor graph through time, interpolating the behavior of the teacher by interpolating between nearby trajectories. The neighbor graph described in the previous section tries to ensure that neighboring portions of trajectories actually represent the same portion of the task, and are not merely close to one another in a non-Markovian state space. Furthermore, imperfections in demonstration trajectories are avoided since they are not strongly connected in the neighbor graph.

A *plan*, like a demonstration, is a sequence of points that can be traversed by the robot. However, a plan is created by the robot learner, not demonstrated by the teacher. A plan begins in the *start region*, which is an area of space inside the convex hull defined by the initial points of demonstrations. Since the planning algorithm relies heavily on interpolation to safely execute the task, it should be unsurprising that plans must start at a location that can be found by interpolating the starting points of demonstrations. Similarly, the *goal*, or endpoint, of a plan is defined by proximity to the final points in demonstrations. However, all demonstrations may not end near one another, for reasons described in

the next section, so a single goal region cannot be easily defined. Instead, the termination criterion for planning will be described in Chapter 6.

Given a query point and its neighbors in the demonstration set, the (possibly weighted) average of the actions performed during demonstration is used to project the query point forward. Next, gradient ascent is used to adjust the location of planned point. This ensures that the plan follows dense clusters of demonstrations. Finally, new neighbors are chosen by advancing the previous neighbors along the neighbor graph, selecting those neighbors that are close to the newly planned point.

Since plans are always created by interpolating between example points, some assurance of obstacle avoidance is provided by this algorithm. However, undetected bifurcations in the example trajectories, or other issues, may compromise the safety of this method. However, the absence of geometric obstacles is determined given a model of the robot and the demonstration trajectories. For each configuration visited by the robot during demonstration, the swath of workspace that it occupies is determined by forward kinematics. By accumulating all these swaths in a discretized occupancy grid, the safe areas of the workspace can be determined. The planner consults this occupancy grid to ensure that planned paths do not move the robot into unknown regions. If a planned point is placed in an unsafe region, the point is shifted into the nearest safe region.

Although safety is a hard constraint, additional trajectory objectives, or soft constraints, may be considered to influence the generation of new trajectories. For example, if smooth trajectories are desired, changes in path curvature should be limited. To improve robot efficiency, path length may be minimized. As noted earlier, though, these objectives are not necessarily universal to all tasks, and should only be applied when the task warrants. Another possible objective that is not achievable with our approach is jitter or random motion that is uncorrelated between demonstration trajectories, but required for a particular task. For example, the path followed by a downhill skier weaves back and forth, transverse to the primary direction of motion. The weaving motions may be uncorrelated between demonstrations, but should not be averaged. Similar situations may arise in robotic manufacturing applications, for

example, when small parts must be jostled to ensure they do not stick together. Correctly handling intentional randomization is reserved for future work.

4.4 Active Learning

While the learner's goal is to produce trajectories that mimic the behavior of the human teacher, we must recognize that the example trajectories are not perfect. We have already discussed how the neighbor graph can be constructed despite noisy examples and inefficiencies in the example paths. In some cases, though, the teacher may provide the learner with conflicting information. That is, example trajectories may perform qualitatively different actions in the same, or similar, states. These different actions divide the demonstration trajectories into two or more *strategies*: collections of demonstrations that perform the motion task by following qualitatively similar paths. Although this situation often indicates hidden variables distinguishing apparently identical states, the robot may not be capable of sensing (or algorithmically processing) that information. Alternately, the teacher, being imperfect, may have inadvertently provided the learner with conflicting examples. Whatever the cause, the learner should resolve its own uncertainty by asking the teacher what to do when it reaches the conflicting situation.

The importance of proactively resolving these ambiguous states is demonstrated by considering the behavior of a planner lacking any additional information. For instance, in a task in which an obstacle appears in the middle of the workspace, a teacher may provide some demonstrations that travel to the left of the obstacle and some to the right. These trajectories may be densely linked in the neighbor graph prior to the obstacle, but will split into two clusters to avoid it. At this point, the planner is faced with the choice of which branch to follow. A naïve approach may simply rely on safety constraints to force the robot into one branch or another. The behavior produced by this strategy is typically disconcerting, though, as the robot may attempt to split the difference between the clusters until they spread far enough apart that the intervening space can no longer be guaranteed safe. The planner then abruptly forces the

robot to one branch or the other in order to continue.

Instead, we seek to detect and resolve *bifurcations* in the neighbor graph before training has ended. A bifurcation is a region of the neighbor graph where trajectories split, and the nature of the inter-trajectory links changes so that the trajectories are divided into separate groups or *partitions*. Each partition represents a strategies demonstrated by the trajectories. Specifically, after the bifurcation region (with respect to the time-ordered intra-trajectory links), a collection of trajectories will have inter-trajectory links that are much denser within the groups than between them. This partitioning is not present (or is less pronounced) prior to the bifurcation. This criterion will be developed more fully in Chapter 7.

We detect bifurcations by examining the demonstrations through time, searching for locations where neighboring trajectories cease to be neighbors. However, the noisy example trajectories provided by the teacher produce a noisy neighbor graph, and many false positives will appear. Noisy trajectories may separate from the neighbor graph for short periods of time during imperfections in demonstrations. It would not be useful to the learner to label each of these small deviations as areas where it must choose between multiple strategies.

Furthermore, the teacher's time is valuable, and the learner should not present questions regarding all possible bifurcations in the graph. In fact, the learner's questions should be ranked so that the most important bifurcations are considered first. We consider the most important bifurcations to be those that involve a large proportion of example trajectories, produce a clean split between the branches, and occur early in the task. This allows the learner to ask the most important questions first, and the answer to a question about an early branch may obviate a question about a later branch.

Bifurcations are presented to the teacher in the form of two partial trajectory plans. The robot executes a plan from the start point to a location just before the split is encountered. The planner considers each branch separately and produces a partial plan that travels a short distance past the split. Each plan is demonstrated for the teacher, and a preference is requested. Since the first partial plan must be executed in

reverse (to return to the start point for the second branch), this approach is only feasible in domains without differential constraints, and where execution is reversible and non-destructive. If the teacher prefers one branch over the other, the planner should avoid the less-preferred branch when producing new plans in the future. If the teacher indicates no preference, the planner receives no additional information, and it may plan in either branch randomly (or in proportion to the number of examples in each branch). As a final option, though, the teacher may wish to indicate that there is no difference between the options presented. Although the learner believes that the two clusters of example trajectories are following different paths or strategies, the teacher believes they are the same. This may occur when too few demonstrations have been provided to sufficiently cover the planning space, resulting in an accidental gap between demonstrations. To resolve this situation, the learner requests an additional demonstration near the bifurcation and between the two clusters. This is done by creating a plan that averages the actions of plans following the two suspected branches. The plan stops executing when the robot reaches the unknown (and potentially unsafe) region just beyond the bifurcation, and the teacher is asked to continue the plan. The goal of this additional demonstration is to instruct the learner how to interpolate between the two clusters, and to illustrate the lack of (geometric or non-geometric) obstacles in the intervening space.

4.5 Summary

Learning motion tasks by demonstration requires the reconciling of imperfect, divergent examples. This work presents a collection of heuristics for determining correspondences between high-dimensional time-series trajectories despite these inherent imperfections. These correspondences form a neighbor graph that is used to safely interpolate between the essential elements of example trajectories and produce novel motions that complete the task. The neighbor graph also provides a high-level structure to the demonstrations that allows the learner to detect places where conflicting strategies are present and to request clarifications from the human teachers regarding these conflicts. This allows

the learner to produce safe, consistent trajectories that exceed any single demonstration by the teacher.

Chapter 5
Neighbor Graph

I have often wished to be human. I study people carefully, in order to more closely approximate human behavior.

— Data, "Hero Worship",
Star Trek: The Next Generation, 1992

The first step in learning trajectories is to understand the structure of the examples presented by the teacher. Ideally, we would like to discover the semantic equivalence between portions of these examples, that is, which portions of different trajectories represent the same part of the task. If two examples follow different strategies, we would like to identify the areas where they begin to diverge. Later, we will also seek to discover whether one of these strategies is superior to the other. Eventually, the learner should be able to produce new trajectories that mimic the examples, performing the same task as the teacher, following one of the strategies presented.

Discovering the semantic similarity between trajectories requires us to find portions of trajectories that are close to one another in space and (task) time, and follow similar paths. Given a collection of examples that meet these requirements, a new trajectory formed by interpolating between the examples can reasonably be said to perform the same task by the same strategy, but without any of the imperfections of the examples. Other collections that differ in one or more aspects may still represent the same task, but they use a different strategy, so interpolating between them may be unsafe or otherwise undesirable. For example, two trajectories that avoid an obstacle in different ways (e.g. one goes to the left and the other goes to the right) may produce paths with similar

curvature and time to reach the goal. However, they occupy substantially different parts of the workspace. Thus, knowledge of the locations of physical obstacles may help us separate trajectories into groups for which interpolation is safe.

Recalling our original goal of discovering semantic similarity between trajectories, we note that obstacle avoidance is not the only constraint in finding correspondences between demonstration trajectories. Even trajectories that follow the same strategy may significantly deviate from one another. Imperfect demonstrations may contain segments of backtracking or detours from the nominal path. Even if no obstacles exist in the midst of these demonstrations, these deviations must be identified in order to determine which portions of similar trajectories represent the same portion of the task. We will then make these semantic correspondences explicit by constructing a neighbor graph, which contains links within and between trajectories. These links connect discrete points sampled from the trajectories that are near each other in both space and (task) time.

However, unintentional jitter in otherwise smooth paths can confuse the correspondences between trajectories. Larger deviations such as backtracking motions or detours from the nominal course must be excised from the correspondence relationship. Although humans can typically discern global structure even in noisy collections of points, it is a challenging task for a robot learner. This problem, known as graph or manifold denoising, has been studied extensively for application to Isomap [75] and other graph-based learning algorithms [28,81,82]. Time-series data presents specific challenges to this effort, but it also has the advantage that part of the structure of the data is known. As ST-Isomap [34] demonstrates, better graphs may be constructed by retaining the temporal links between adjacent sample points on each individual trajectory. These points are nearby in time and space, and they are logically similar to one another since they arise through the movement of the robot.

Forming neighbor links between example trajectories is more challenging, though. The rest of this chapter examines the selection of neighbors between trajectories. Spatial proximity of points is not a sufficient

similarity metric, especially if the space is non-Markovian. A robot may visit the same configuration multiple times during the execution of a task. Even in Markovian spaces, though, spatially-proximal portions of trajectories that occur at different times in the task may produce spurious neighbor links. This challenge is illustrated by the example task discussed in Section 5.4.

Temporal proximity between example trajectories may provide some useful information if the timing of the demonstrations is carefully controlled, or if normalization techniques such as dynamic time warping [57] are used, but only in simple cases. For example, different examples may also use different strategies (and therefore different paths) to complete the task. Jitter and other imperfections (such as those discussed in Section 5.2) can create enough divergence between examples to render dynamic time warping insufficient.

The remainder of this chapter discusses the algorithm we have developed for choosing the neighbor graph links between example trajectories. The next section provides a preliminary discussion of the coordinate spaces and distance metrics used for comparing trajectories. Section 5.2 describes the heuristics developed to detect and deal with imperfections and other undesired motions in example trajectories. Section 5.3 discusses our approach for ensuring safety for plans created from this neighbor graph, given that we have no explicit model of obstacles in the environment. Finally, Section 5.4 presents the results of experiments building neighbor graphs from trajectories collected using a 7-DOF manipulator.

5.1 Coordinate Spaces

The neighbor graph itself is simply a set of relationships between points sampled from trajectories. In order to build this graph or use it for planning, though, these points must be embedded in some metric space. A distance metric is required to determine the similarity between points, and thus to build the links of the neighbor graph. Similarly, a distance metric is used to interpolate between points when planning novel trajectories.

For robot trajectories, the obvious candidates for coordinate spaces are configuration space or workspace (specifically, the workspace coordinates of the end-effector, or some other salient point on the robot). In either of these spaces, the Euclidean distance metric usually provides reasonable and intuitive determinations of which points are relatively close to one another, and which points are far apart. This provides a sensible basis for learning and planning.

Each of these coordinate spaces presents challenges of their own, though, particularly when end-effector orientation is to be considered. For the workspace, the problem is that there is no canonical distance metric for SO(3), 3-dimensional position with 3-dimensional orientation. However, when a redundant manipulator is used, orientation can provide a crucial distinction between the infinite number of manipulator poses with the same end-effector position. Finding a weighting factor to combine the differences between positions and orientations into a single distance measure can unfortunately become an empirical process with a result that does not generalize to multiple tasks. Fortunately, position alone typically provides a sufficient similarity metric for tasks performed with consistency.

Working in configuration space avoids this problem, at least for robots with a single type of joint. In our experiments, we used a manipulator with seven revolute joints, so each dimension of configuration space measured the same (angular) units. In this space, the Euclidean distance metric captures difference in position and orientation simultaneously, but the results are not necessarily intuitive. Movement in the different dimensions of this space do not contribute equally to the movement of the robot's end effector. As before, weights for each dimension may be derived to produce a more intuitive distance metric. An example of such a weighting scheme is the Joint Dominance Coefficients [38], which provides a statistical measure of the influence each joint exerts over the workspace movement of a set of points on the robot.

For pedagogical clarity, this work will calculate neighbors using the three-dimensional workspace position, unless otherwise noted. Although this space fails to distinguish multiple configurations that place the end effector in the same location, the neighbor graphs are far easier to illus-

trate and interpret than those produced using seven-dimensional configuration space. When producing plans for a redundant manipulator, though, distances should be calculated in configuration space.

One final coordinate space of note is the low-dimensional embedding discussed in Chapter 3. After constructing the neighbor graph, pairwise geodesic distances can be calculated, and Isomap (or some other dimensionality reduction technique) may be used to construct an embedding that preserves these distances as well as possible. Although this space does not present a representation of the graph useful for planning, examination of example trajectories and their neighbor graphs embedded in such a space can provide valuable qualitative information on the soundness of the neighbor links that can aid in algorithm design and debugging. A successful embedding creates a sort of *task space*, in which the distance between points approximates their separation in the performance of the task. For example, the crossing points in Figure 3.3 where the robot visits the same configurations twice during the execution of the task produce two distinct regions in the task space. Their separation is proportional to the amount of the task executed between visits to that configuration, rather than the similarity of the configurations. Visually examining features such as these in an embedding has aided in the development of the neighbor finding algorithms described in this chapter.

5.2 Trajectory Neighbor Heuristics

When searching for a point's neighbors, individual points are not considered in isolation. Instead, we consider each pair of trajectories separately, and search for subsequences of those trajectories that contain points that match in a roughly pairwise manner. That is, as the indices in both subsequences increase, the points of each trajectory maintain similar juxtapositions. Since all trajectories are initially subsampled at a uniform distance d_u between adjacent points, this means that, informally, matching subsequences travel in the same direction. The algorithm described here relies on heuristics to replicate human intuition in recognizing the correspondences between multiple noisy, imperfect

demonstrations of a motion task.

Algorithm 5.1 The Neighbor Graph algorithm

```
1:  function NeighborGraph(T)
2:      ▷ Initialize adjacency list with intra-trajectory links
3:      L ← L_a
4:      for all ordered pairs (traj_a, traj_b) ∈ T do
5:          ▷ Look for runs of roughly pairwise neighbors
6:          runs ← NeighborRuns(traj_a, traj_b)
7:          ▷ Remove runs that backtrack along the
8:          ▷ neighboring trajectory
9:          runs ← BacktrackCheck(runs)
10:         ▷ Remove many-to-one neighbors
11:         nearest ← ManyToOneCheck(traj_a, traj_b, runs)
12:         ▷ Add remaining neighbors to adjacency list
13:         for i = 0 → ||nearest|| − 1 do
14:             v_b ← traj_b[i]
15:             v_a ← traj_a[nearest[i]]
16:             d ← dist(v_b, v_a)
17:             L ← L ∪ {(v_b, v_a, d), (v_a, v_b, d)}
18:     return L
```

The construction of the neighbor graph algorithm is presented in Algorithm 5.1. The adjacency list L is initialized with the set of intra-trajectory links, as defined in Equation 4.1. To add the inter-trajectory links, each pair $(traj_a, traj_b)$ of demonstration trajectories is considered twice: first finding the nearest neighbors in $traj_a$ from the points in $traj_b$, then in the other direction. This is necessary because nearest nearest neighbor is not symmetric. The links found here are added twice since the direction is not important, as it is for the time-ordered intra-trajectory links, but we define all links to be directed for consistency.

The first subroutine, NeighborRuns (Algorithm 5.2), searches for *runs*, or subsequences of the trajectories in which the nearest neighbors roughly correspond. Because the demonstrations contain jitter and noise, we do

Algorithm 5.2 The Neighbor Graph run finding algorithm

```
1:  function NeighborRuns(traj_a, traj_b)
2:      ▷ Find nearest neighbors in b for each point in a
3:      for i = 0 → ||traj_a|| − 1 do
4:          neigh[i] ← NearestNeighbor(traj_a[i], traj_b)

5:      ▷ Search for runs of neighboring points
6:      runs ← {}
7:      run ← {}
8:      for i = 0 → ||traj_a|| − 1 do
9:          safe ← SafetyCheck(traj_a[i], traj_b[neigh[i]])
10:         if run ≠ ∅ then
11:             diff ← i − neigh[i]
12:             initial_diff ← run[0]_a − run[0]_b
13:             cond_1 ← abs(diff − initial_diff) < MAX_DIFF
14:             cond_2 ← i − run[−1]_a < MAX_DIFF
15:             cond_3 ← neigh[i] ≥ run[−1]_b
16:             cond_4 ← safe
17:             if ∃i s.t. cond_i ≡ false then
18:                 if ||run|| ≥ MIN_RUN then
19:                     runs ← runs ∪ run
20:                 run ← {}
21:         if safe then
22:             run ← run ∪ {(i, neigh[i])}
23:     ▷ Add the final run to the list
24:     if ||run|| ≥ MIN_RUN then
25:         runs ← runs ∪ run

26:     return runs
```

not enforce a strict alignment between points of the trajectories, but use a series of heuristic conditions to ensure approximate alignment. The conditions, as listed in lines 13–16 of Algorithm 5.2 are:

1 : We record the difference in point indices at the start of a run, and for each candidate pair in the run. This condition specifies that this difference varies by no more than MAX_DIFF, thus ensuring that the alignment between trajectories does not change significantly

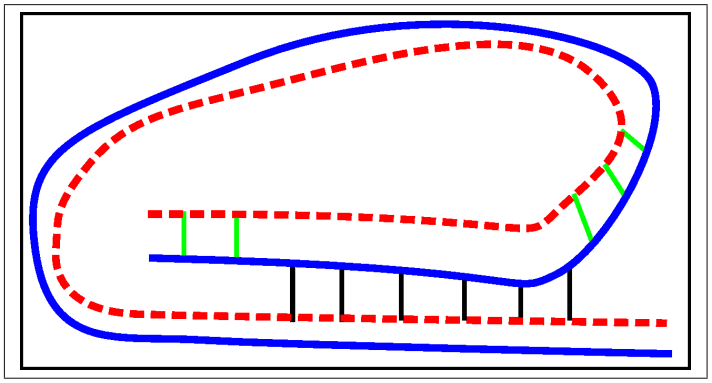


Figure 5.1: The solid (blue) trajectory drifts closer to distant sections of the dashed (red) trajectory. Pointwise nearest-neighbor alone is not sufficient to correct this problem.

within a run. Throughout our experiments, we used a value of 3 for this parameter.

2 : Similarly, we ensure that there is no consecutive sequence of MAX_DIFF points missing within a single run. Note that the array subscript -1 follows the syntax of the python language, indicating the last item in any array.

3 : This algorithm examines the points from $traj_a$ in order. This test ensures that corresponding neighbors in $traj_b$ do not appear in reverse order.

4 : A workspace safety check is executed in line 9 of the algorithm to ensure that the robot can move between the candidate neighbor points without impacting any physical obstacles. This safety check is discussed in detail in Section 5.3.

These conditions result in a list of *runs* of neighbors that ignores minor deviations separating the pair of trajectories. However, undesirable neighbor links may be formed due to larger-scale inconsistencies. For example, Figure 5.1 illustrates a case in which an interally consistent run of neighbors is clearly incorrect in the context of other runs. This situation occurs when portions of trajectories that are distant in

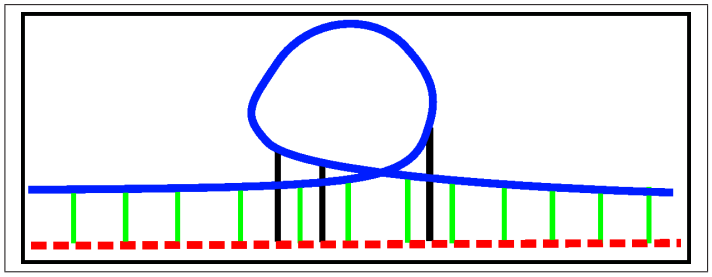


Figure 5.2: Regularly spaced neighbor links from the solid (blue) trajectory to the dashed (red) trajectory result in some undesirable links (black).

task space are near to each other in workspace. In fact, if the space is non-Markovian, distinct portions of the task may overlap in workspace. Similarly, Figure 5.2 illustrates the overlapping runs that may occur due a loop or other artifact in demonstration trajectories.

The backtracking checks of Algorithm 5.3 search for these errors and remove the offending neighbor links. The general strategy in this function is to search for subsequent indicies of $traj_b$ that *decrease* while the indices of their corresponding neighbors in $traj_a$ *increase*. The first check, in lines 5–18 of the algorithm, searches for entire runs that skip backwards along $traj_b$. Note that the run containing the regression is not always the run that is invalidated. Consider the trajectories of Figure 5.1. If $traj_a$ is the solid line, $traj_b$ is the dashed line, and the trajectories begin in the bottom right corner of the image, it is true that the incorrect neighbor links (dark lines) connect $traj_a$ to an earlier section of $traj_b$ than the previous run. However, if the trajectories move in the other direction, starting in the center of the spiral, the dark neighbor links skip far ahead along $traj_b$, and it is the following run that appears to regress. In both cases, a regression signals the presence of an invalid run, but we must check the alignment of nearby runs to determine which run is in error. The second check in lines 19–22 addresses the problem of Figure 5.2. In this case, subsequent runs monotonically advance along $traj_a$, but regress and overlap along $traj_b$. This check simply removes overlapping neighbors.

Finally, one-to-many neighbor links are not allowed. Only the one-

Algorithm 5.3 The Neighbor Graph backtracking checks

```
1:  function BacktrackCheck(runs)
2:      ▷ Remove entire runs that are out of alignment
3:      prev_diff ← runs[0][0]_a − runs[0][0]_b
4:      for all i ∈ 1.. ‖runs‖ do
5:          diff ← runs[i][0]_a − runs[i][0]_b
6:          ▷ Compare alignment of this run to nearby runs
7:          if runs[i][0]_b < runs[i − 1][0]_b then
8:              if i − 1 ≥ 0 then
9:                  other_diff ← runs[i − 2][0]_a − runs[i − 2][0]_b
10:             else if i + 1 < ‖runs‖ then
11:                 other_diff ← runs[i + 1][0]_a − runs[i + 1][0]_b
12:             else
13:                 other_diff ← 0
14:         ▷ Remove run farthest from alignment
15:         if abs(prev_diff − other_diff) < abs(diff − prev_diff) then
16:             runs ← runs − runs[i]
17:         else
18:             runs ← runs − runs[i − 1]

19:     ▷ Remove points from runs that overlap with other runs
20:     for all i ∈ 1.. ‖runs‖ do
21:         while runs[i − 1][−1]_b > runs[i][0]_b do
22:             runs[i − 1] ← runs[i − 1] − runs[i − 1][−1]

23:     return runs
```

to-one link with the shortest distance is permitted in the final neighbor graph. This restriction ensures that the geodesic distances increase quickly when trajectories deviate from one another, as in Figure 5.3. Algorithm 5.4 resolves the conflict in favor of the shortest link.

5.3 Safety

Our strategy for creating new plans from the neighbor graph will require interpolation of demonstration trajectories. The heuristics of the previous section attempt to discover the alignment of sections of these

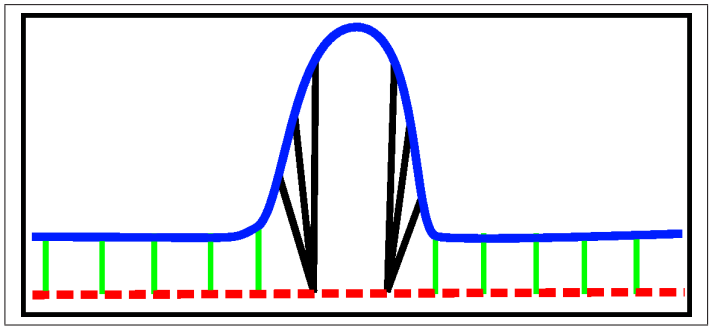


Figure 5.3: Many-to-one neighbor links result in a graph with too many links, and thus geodesic distances that are too small. Deviations such as that shown in the solid (blue) trajectory should appear distant in the neighbor graph.

Algorithm 5.4 The Neighbor Graph many-to-one check

```
1:  function ManyToOneCheck(traj_a, traj_b runs)
2:      nearest ← [∅]_||traj_b||
3:      for all run ∈ runs do
4:          for all (idx_a, idx_b) ∈ run do
5:              d ← dist(traj_a[idx_a], traj_b[idx_b])
6:              if adj_list[idx_b] ≡ ∅ then
7:                  prev_dist ← inf
8:              else
9:                  prev_dist ← dist(traj_b[idx_b],traj_a[adj_list[idx_b]])
10:             if d < prev_dist then
11:                 adj_list[idx_b] ← idx_a
12:     return nearest
```

trajectories while ignoring minor deviations and errors. However, the similarity of these linked sections does not ensure that an interpolation between them is a valid trajectory. An intervening workspace obstacle may exist, blocking the interpolated path. To help prevent our planner from creating such paths, our neighbor-finding algorithm does not create links that pass through (potential) obstacles. More than this, though, we want to prevent such neighbor links because two trajectories that pass on opposite sides of an obstacle are apparently following different motion strategies. We do not want to create plans that blend these two distinct

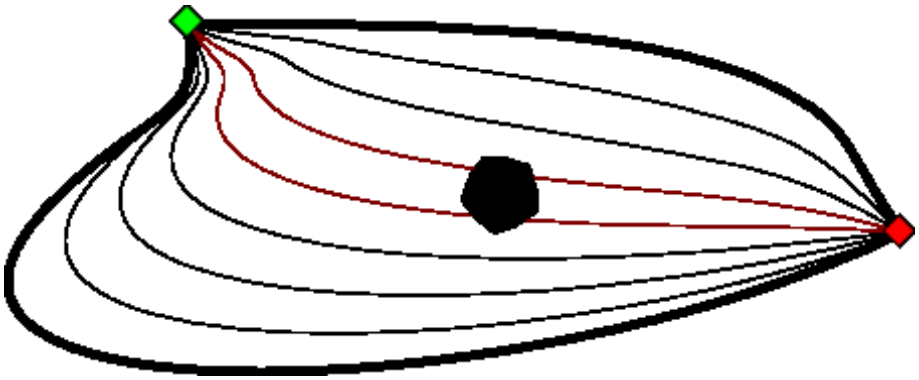


Figure 5.4: In two dimensions, interpolating between homotopic paths is always safe. In three dimensions, an interpolation may not be safe, but another deformation may be possible.

strategies. Chapter 7 describes our strategy for resolving situations in which demonstrations split into multiple strategies, but we pause here to why these splits are so hard to detect (especially in SE(3) or SO(3)), and to examine the impact on the neighbor graph we are constructing.

When operating in the plane, determining which trajectories may be safely interpolated is relatively straightforward [21]. This situation is exemplified by a planar manipulator, or non-redundant manipulator with end-effector workspace confined to a plane. Figure 5.4 illustrates the workspace paths of such a manipulator. If a safe (collision-free) continuous deformation is possible between two paths with fixed start and goal points, these paths are said to be *homotopic* [48].

In higher dimensions, though, homotopy is not sufficient to separate paths into distinct strategies. For example, if the robot in Figure 5.4 is permitted to leave the plane, and the obstacle in the center of the image in known to have finite height, then all of the illustrated paths are homotopic. There is a smooth deformation (if not a simple interpolation) between these sets of paths around any simply-connected, finite obstacle. This means that a set of example trajectories in three-dimensional space cannot be easily divided into classes representing distinct strategies such as “left of” or “right of” an obstacle. Returning to our goal of programming by demonstration, it is not even possible to ask

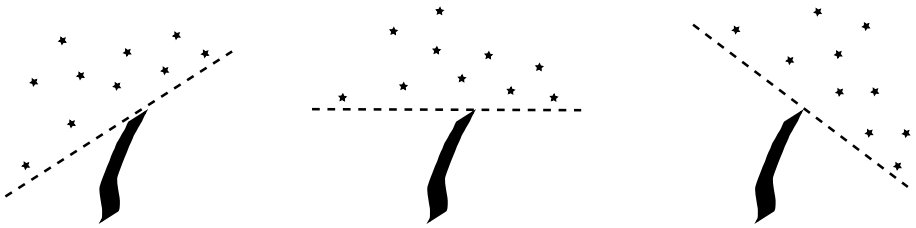


Figure 5.5: Three cross-sections from sets of homotopic example trajectories near an obstacle. Each star is a single point from a different example trajectory. In each case, the examples lie within a half-plane on one side of the obstacle, so interpolation is safe. However, if the trajectories were combined, interpolation would no longer be possible.

a teacher to restrict examples to a single such strategy. That is, we cannot rely on a human teacher to provide a set of trajectories near obstacles such that smooth interpolations between any pair of trajectories is safe. Figure 5.5 illustrates three strategies for avoiding a protruding obstacle. Each image represents a snapshot in time from a set of trajectories moving through the plane of the image. In each case, interpolation between the points is safe because each set exists in a half-plane on one side of the obstacle. This half-plane would be difficult for a person to visualize while producing demonstration trajectories, though. Since we cannot rely on the teacher to provide trajectories that may be safely interpolated, we must assume that demonstrations may follow multiple distinct strategies.

Our approach to safety requires a model of the robot, but not the environment. This requirement should be easy to fulfill since robots change far less often than the environments in which they operate, and many identical robots are generally produced with the same hardware configuration. Given the example trajectories, it is straightforward (though computationally expensive) to determine all areas of the workspace that have been occupied by the robot. For each pose in the demonstration trajectories, we mark all the voxels of an occupancy grid that are occupied by the robot. Each trajectory sweeps out *swaths* of space that cannot contain obstacles. The teacher may also elect to provide a conservative buffer around the robot's position that is also considered safe

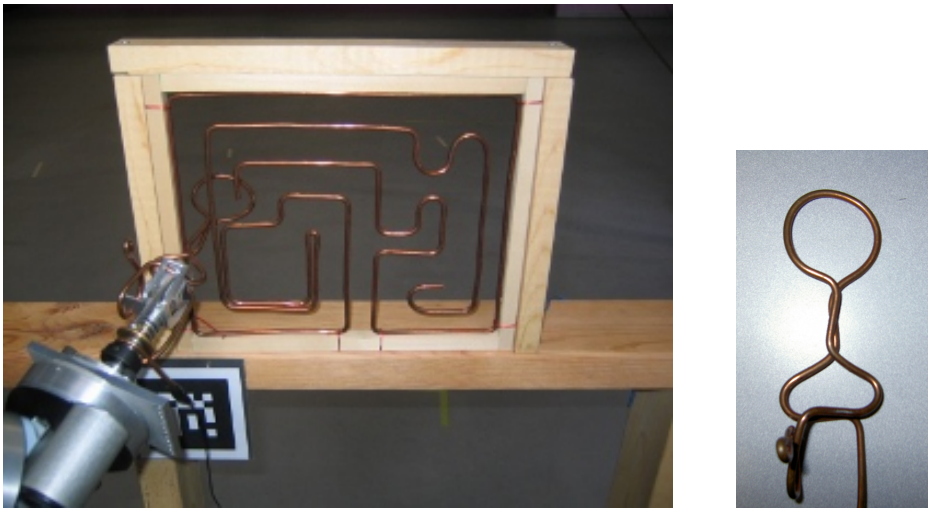
to occupy. This procedure is performed offline as a post-processing step on the demonstration trajectories. As a computational optimization during both post-processing and plan generation, we may consider only the end effector or final link of the robot, if these are the only parts of the robot likely to be in collision with the environment. We took advantage of this optimization in our experiments, where the only physical obstacle was near the end effector.

The occupancy grid constructed by this method provides a conservative estimate of the safe workspace in the robot's environment. Marked voxels are known to be free of static obstacles, but unmarked voxels are unknown. Additional robot poses may be tested against this voxel grid to determine whether they are safe. The *SafetyCheck* called in Algorithm 5.1 performs this check along a candidate neighbor link to determine whether the robot may safely move between the two endpoints without occupying any unknown space.

5.4 Experimental Results

Experiments were conducted with a 7-DOF WAM manipulator using the maze task shown in Figure 5.6. When the arm is operated in gravity-compensation mode, it can be easily moved by hand. Participants were asked to perform kinesthetic demonstrations navigating the pictured maze, without touching the edges. When the robot's copper end effector contacts the walls of the maze, a buzzer provides auditory feedback. The wire maze is effectively two-dimensional, though some of the rotational axes are relatively unconstrained, allowing additional variation in the demonstrated trajectories. This rotational variation is not strictly necessary for navigating the maze and is unlikely to be correlated between example trajectories. The linear portion of the end effector is used to navigate the maze, and the proximal and distal loops keep the end effector within the plane of the maze. Six demonstrations were performed by each of fifteen adult participants, recruited via word-of-mouth from Carnegie Mellon University and the surrounding community. Seven of these participants had previous experience operating robots.

This task is inherently two dimensional. Although the walls of the



(a) (b)

Figure 5.6: The WAM manipulator traversing the wire maze, and a closeup of the end effector. The 2-D barcode in the lower-left is a visual fiducial used to detect the location of the rig relative to the robot.

maze twist through the workspace, there is only one path from start (at the bottom center in images) to goal. The maze corridors are usually about 5 mm wide, so the task requires precise motion. The portion of the maze just to the left of the first corridor is parallel to the first section, and thus offers an opportunity to demonstrate the backtracking heuristics of Algorithm 5.3.

Figure 5.7 shows the neighbor links chosen by various methods and the resulting embeddings. In an ideal task space, we expect demonstration trajectories to be embedded such that one dimension represents time and another represents variation between examples. We expect a dimensionality reduction technique should be able to discover such an embedding. Using Isomap, which relies on geodesic distances between all pairs of points, should help evaluate the quality of the neighbor graph used as input. A correct neighbor graph (with dense connections between trajectories at the same point in the maze, and no connections crossing maze walls) should produce an embedding with dimensions as described above. Figures (a) and (b) were produced us-

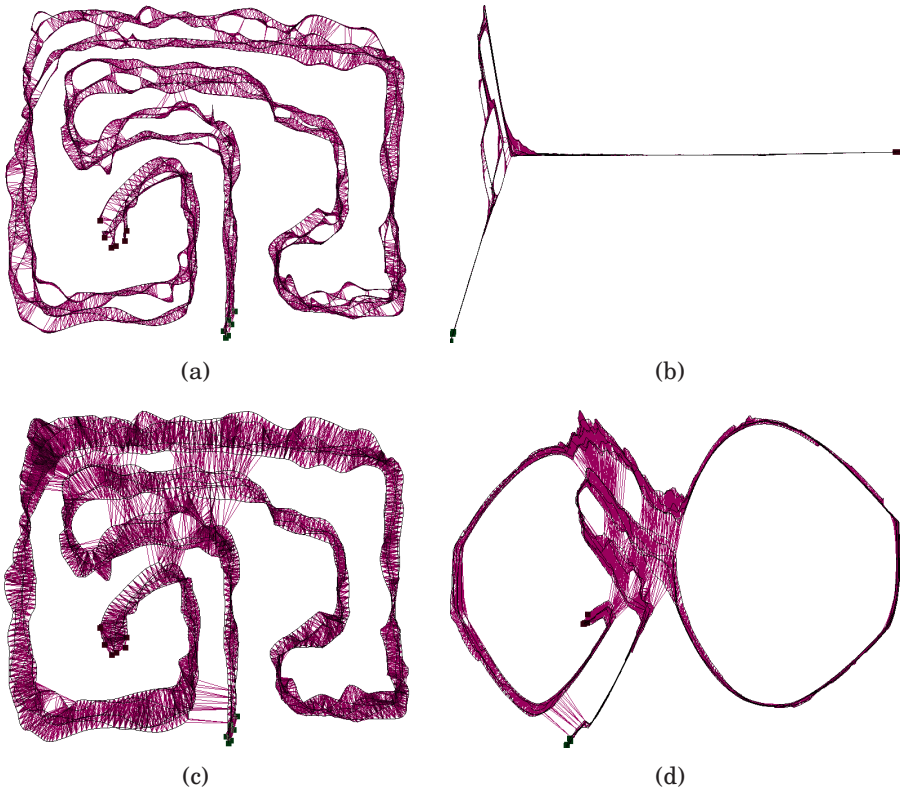
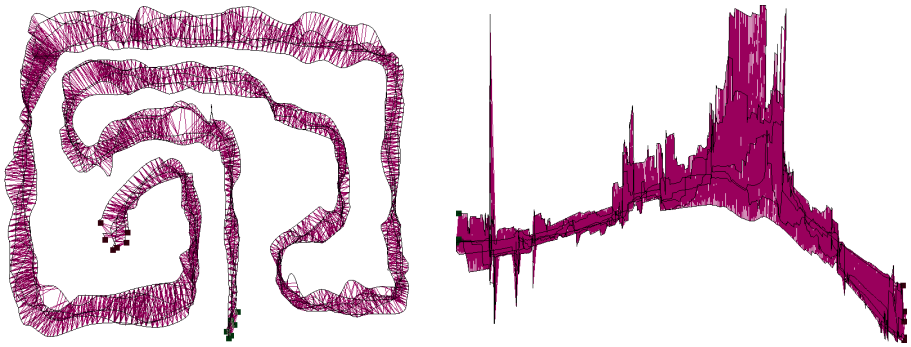


Figure 5.7: Neighbors selected by (a) k-Isomap and (c) ST-Isomap, and their embeddings (b) and (d). Spurious short-circuit neighbor links produce embeddings unusable for planning.

ing the k-nearest neighbor strategy of the original Isomap algorithm, with $k = 10$ chosen to ensure the number of neighbors per point is roughly equivalent to that of the other algorithms. The workspace plot of Figure 5.7(a) appears sparser than the corresponding images for ST-Isomap (Figure 5.7(c) below) and our algorithm (Figure 5.8(a)) because k-NN produces more neighbors between points in the same trajectory. The ST-Isomap result illustrates some of the most difficult situations for neighbor selection. In many cases, such as the spurious neighbor links near the bottom of Figure 5.7(c), links are formed between trajectories that are relatively near to one another, and even travelling in parallel directions. Without considering global information about the trajectories,



(a) (b)

Figure 5.8: A two-dimensional projection of the workspace trajectories for the wire maze task (left) and the two-dimensional embedding of its 7-DOF configuration space (right). Purple lines represent neighbor links.

these incorrect links are difficult to detect.

The neighbor links and corresponding embedding produced by our approach can be seen in Figure 5.8. Spurious links crossing maze walls have been eliminated, and dense neighbor connections are found within the maze pathways. As expected, the two-dimensional embedding contains paths that primarily move from left to right, with vertical deviations indicting transient similarity between examples. These deviations appear large and abrupt because the relative scale of dimensions in the embedding is arbitrary.

Finally, dimensionality reduction provides a quantitative measure in terms of the residual variance of each of these approaches. Residual variance is an indication of the amount of information lost by reducing the dimensionality of data. For the Isomap variants, it measures the error introduced in the original dataset by comparing the geodesic distances between all pairs of points in the original space with the Euclidean distances in the reduced dimensionality space. As a baseline measure, we also consider Singular Value Decomposition (SVD), which incorporates no task-specific knowledge. SVD simply reprojects the new points into the original space and compares Euclidean pairwise distances with the original points. Errors are normalized, and the results are presented as a fraction of the original distances.

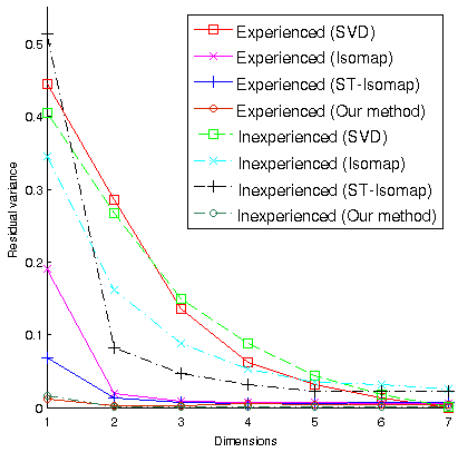
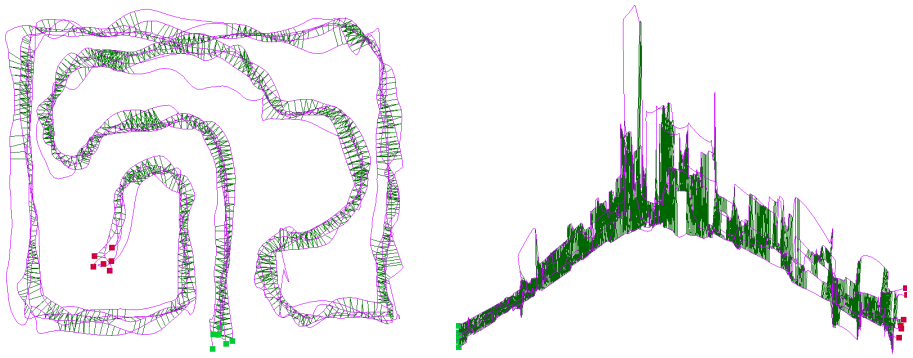


Figure 5.9: The residual variance for four different dimensionality reduction techniques. Our performs best in both cases, followed by ST-Isomap, but the examples provided by the inexperienced users retain more variance.

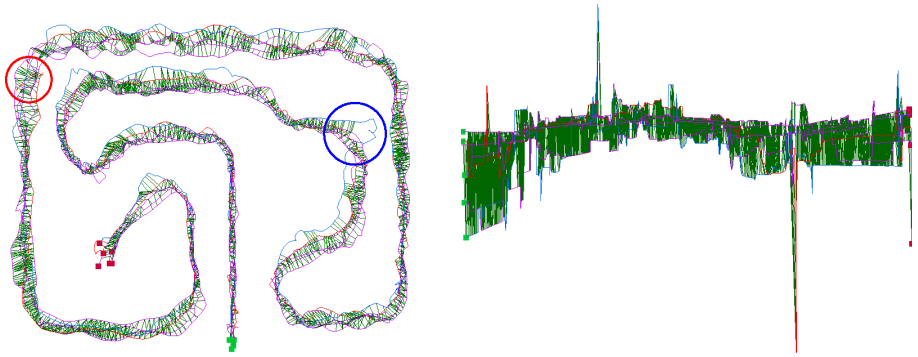
Figure 5.9 quantifies the error in SVD, Isomap, ST-Isomap, and our method for the task on the WAM arm. The data is further separated based on the level of experience that the user had in operating this device. We may draw a few conclusions about the use of ST-Isomap for this application from this plot. There is a distinct elbow in the traces for ST-Isomap for both experienced and inexperienced users at two dimensions. This suggests that two dimensions may be sufficient to represent at least the seven-dimensional trajectories from this scenario. In fact, in the experienced case, the residual has nearly reached its minimum at two dimensions, so additional dimensions would offer little improvement. The inexperienced case, though, retains far more variance initially, and seems to asymptote higher.

Qualitatively, this seems unsurprising since the collection of inexperienced trajectories (e.g. Figure 5.10) appears less consistent than those of the experienced users (e.g. Figure 5.11). The inexperienced user's trajectories tend to be more spread out, and they lack neighbor links in the most extreme cases. This causes the trajectories to separate and contract in the embedding. The experienced user's trajectories appear more



(a) (b)

Figure 5.10: A two-dimensional projection of the wire maze trajectories created by an inexperienced robot user (left) and their embedding (right).



(a) (b)

Figure 5.11: A two-dimensional projection of the wire maze trajectories created by an experienced robot user (left) and their embedding (right). The significant deviations of the red and blue trajectories visible in the embedding are marked in the workspace.

consistent in the workspace, which permits a more consistent neighbor graph and embedding. Two significant areas of deviation are highlighted in the workspace, and appear as vertical spikes in the embedding. The blue trajectory separates significantly from the other trajectories at one point, and the red trajectory backtracks briefly. In these cases, neighbor links are not created, so the offending portions of the

trajectories appear to be much farther away from the other trajectories in terms of geodesic distance. This forces them farther away in the embedding, as well, producing the spikes in the image.

The visible disparities between the neighbor graphs and embeddings of experienced and inexperienced trajectories may actually be useful in that it may allow the system to automatically distinguish the skill level of the person demonstrating the task. For example, if a human (rather than the robot) is being trained to perform a task, this may allow a quantitative analysis to determine when the person has mastered the skill.

5.5 Summary

In this chapter, we have discussed the use of neighbor graphs for understanding the structure of demonstration trajectories in high-dimensional space. We have discussed the inadequacy of path homotopy in determining which paths follow similar strategies to complete a task. Instead, we apply a series of heuristics to discover the semantic similarities between portions of trajectories, despite imperfections in the demonstrations. Our approach also eliminates the selection of the domain-dependent k or ε parameter for defining the size of a neighborhood whose optimal value can vary within a single domain. Our approach requires only the specification of d_u, the uniform step size required to represent trajectory features. Since this parameter has a straightforward physical meaning, it can be selected more intuitively by the user. Dimensionality reduction is used to illustrate and verify this approach.

Chapter 6
Planning

You have to do what someone asks you, don't you? ... Don't you? If you love them?

— Sonny, *I, Robot*, 2004

Planning, in the context of PbD, is the generalization of demonstrations to create new trajectories. Demonstrations provide only examples of the behavior that the robot should perform, and are unlikely to fill the robot's state space so densely as to provide an action for every possible state. Moreover, we acknowledge that the examples provided by humans are imperfect, and conflicting actions may be provided for the same (or similar) states in different example trajectories. Although many learning algorithms can cope with noisy demonstration data, our neighbor graph provides a basis for determining which demonstrations are likely to represent the teacher's true intentions. Rather than precisely duplicating the teacher's motions, our planner interpolates between the multiple provided trajectories, eliding the inconsistencies and errors that the neighbor-finding algorithm addresses.

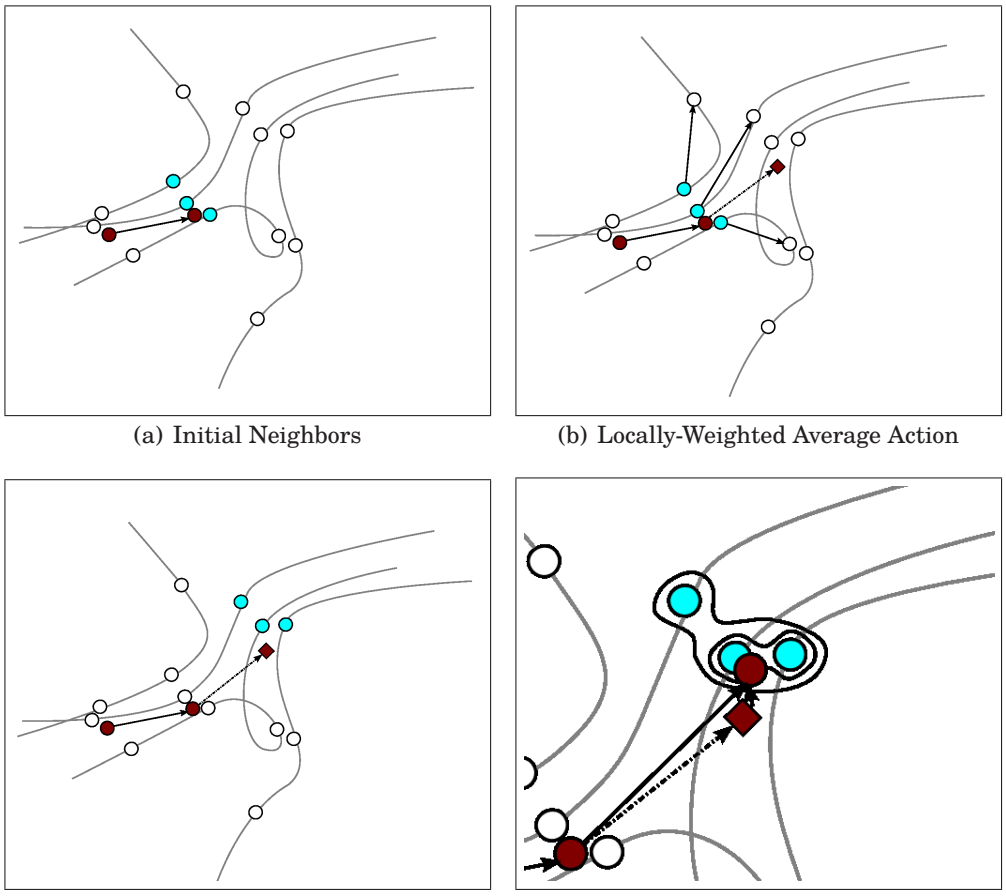
Our planning algorithm operates directly over the neighbor graph, the set of connections between discrete points sampled at a uniform interval from demonstration trajectories. For each iteration in the construction of a new plan, the planner maintains a set of neighbor points from the demonstration set. These neighbors are not merely the closest points in terms of a metric over the planning space; they should represent the portions of the training examples at semantically equivalent states during execution of the task. As before, distinct portions of example trajectories may appear in close proximity in non-Markovian regions,

Algorithm 6.1 The trajectory planning algorithm

```
1: function CreatePlan(T, p, neighbors, sdf)
2:     plan ← []
3:     while True do
4:         ▷ Find the next plan point according to Equation 6.4
5:         p+ ← WeightedPointExtension(T, p, neighbors)

6:         ▷ Find neighbors for the new point
7:         neighbors ← NeighborExtension(T, p, p+, neighbors)
8:         ▷ Terminate when neighbors cannot be extended due
9:         ▷ to end of demonstration trajectories
10:        if AtGoal(p) then
11:            break

12:        ▷ Refine the planned point using its new neighbors
13:        p+ ← Refine(T, p+, neighbors)

14:        ▷ Use the Signed Distance Field to ensure safety
15:        last_safe ← sdf.CheckLine(p, p+)
16:        if last_safe ≠ p+ then
17:            p+ ← sdf.Correct(last_safe)

18:        ▷ Record the new point and prepare for the next iteration
19:        plan.Append(p+)
20:        p ← p+

21:    return plan
```

but an internally-consistent set of neighbors will be closely related in the neighbor graph. Initializing the set of neighbors at the beginning of the task is thus straightforward: nearby points from the beginning of example trajectories are chosen.

The planning procedure, illustrated in Figure 6.1 and presented in pseudocode in Algorithm 6.1, proceeds inductively by advancing the planned position based on the actions of neighboring points, determining the next set of neighbors, then refining the position of the next step in the plan. The details of each of these operations are provided in the following sec-



(a) Initial Neighbors (b) Locally-Weighted Average Action

(c) Neighbor Extension (d) Plan Refinement

Figure 6.1: (a) The graph-based planning algorithm begins with a partial plan (red points) and a set of neighbors (solid outline points) selected from among demonstration trajectories (light blue). (b) The initial estimate for the next plan point is computed using the locally-weighted average of neighbor actions. (c) Neighbors are selected for the new plan point. (d) The pose of the new plan point is refined using the new set of neighbors.

tions. Section 6.4 then deals with an exceptional case: planning when the neighbor graph bifurcates, providing conflicting information about the teacher's desired strategy. Finally, Section 6.5 presents and evaluates some plans created using this algorithm in two experimental do-

mains.

As noted in Section 5.1, configuration space should typically be used for redundant platforms, but the workspace end-effector position may be used in many cases. Unlike the latent task space developed in Chapter 3, configuration and workspace are likely to be non-Markovian, so our planner cannot calculate a global policy. That is, the robot may visit the same regions of configuration or workspace multiple times during the execution of a task, such as the three areas in Figure 3.3 where the demonstration trajectories cross themselves.

6.1 Action Selection

The initial action is computed as a locally weighted average of the actions of its neighbors (Figure 6.1(b)). The demonstration trajectories have previously been discretely sampled at a uniform interval d_u, so actions are computed as the vector from the current neighbor point to its subsequent point in the same trajectory. The learning algorithm is not sensitive to the value of d_u, but this discretization distance should not be so large as to elide small-scale elements of the demonstrations that should be reproduced by the learner.

As a matter of notation, for an example trajectory point t_i, we represent its successor as t_i^+. Thus, given N neighbors of a plan point p, their (normalized) weights w_n and the subsequent plan point are calculated as:

$$w_n = \frac{1}{d_u + \|t_n - p\|} \tag{6.1}$$

$$W = \sum_{n=0}^{N} w_n \tag{6.2}$$

$$a_p = \sum_{n=0}^{N} \frac{w_n}{W} \left(t_n^+ - t_n\right) \tag{6.3}$$

$$p^+ = p + \frac{d_u}{\|a_p\|} a_p \tag{6.4}$$

These equations provide the functionality of the WeightedPointEx-

TENSION function used by Algorithm 6.1. The weight computation in Equation 6.1 includes d_u in the denominator to avoid division by zero when a neighbor happens to be coincident (or nearly coincident) with the plan point. Without this term, nearby neighbors are able to overwhelm any influence by more distant neighbors. The weighted average action a_p is taken as the *direction* of the desired action, but is rescaled by d_u to ensure a reasonably consistent action distance at each step in the plan. The boundary condition, when some or all of the t_n^+ points do not exist, occurs when the plan reaches the end of the demonstration trajectories. At this point, the plan has reached the demonstrated goal, and planning stops.

6.2 Neighbor Extension

Next, we advance the set of neighbors (Figure 6.1(c)). The new neighbors will be used both to refine the current planned action and to provide neighbors for the next planning step. Rather than simply advancing to the set of points subsequent to the original neighbors, the NEIGHBOREXTENSION function used in Algorithm 6.1 searches near p^+. We remove neighbors that are too far away and add new neighbors that are nearby in both metric and graph distance. In our experiments, we search for points within d_u of p^+ and no more than three graph links from the original set of neighbors. Since the graph is directed, these links may move forward in time along a trajectory, or between trajectories, but not backward in time. The allowance of three links permits extension to trajectories that are not directly linked to current points, but which are neighbors of neighbors, while still permitting a step forward in time.

We also manage the size of the neighbor set so that it does not grow too large or too small. If the neighbor extension results in fewer than three new neighbors, the planner risks following a single demonstration with no interpolation. With too many neighbors, the planner is less focused on a specific area of the graph, and may interpolate too broadly, so we limit the number of neighbors to $|T|$, the number of demonstration trajectories. This limit is easily implemented by simply choosing the closest demonstration points. When additional neighbors are required,

we expand the search radius d_u to find the nearest demonstration points, regardless of their graph distance.

Again, the effect of this strategy is that the set of neighbors marches forward through the task, even if it does not advance uniformly along the demonstration trajectories. In practice, even well-clustered sets of example trajectories meander toward and away from one another. Thus, a plan's collection of example trajectories will gain and lose members over time, as shown in Figure 6.1(c). One of the initially neighboring trajectories veers upward and away from the others. Another trajectory appears from the bottom of the image and aligns closely with the central cluster.

6.3 Plan Refinement

Finally, the locations of the new neighbor points are used to refine the position of p^+. The initial action estimate computed by the weighted average has interpolated the actions demonstrated by the teacher at this point in the task, but undesired effects due to noise and jitter are included in the average as well. Since the neighbor graph was constructed to detect and avoid undesired actions, we attempt to follow it by adjusting p^+ toward a position that is nearby (i.e. representative of the average action performed by the teacher) and close to the clusters of points that follow the original neighbors in the neighbor graph. This is necessary since the new neighbors are not, in general, the successors of the previous neighbors. If this were the case, the weighted average of their actions would produce a reasonable action for the plan. However, neighboring trajectories have been lost and gained, and portions of demonstration trajectories may have been skipped to avoid non-optimal movements. For example, the loop in one example of Figure 6.1 is never used as a neighbor for planning. Thus, we must take care to adjust the position of the newly planned point to keep it close to its new neighbors.

To refine the planned point toward clusters of nearby neighbors, the refinement algorithm (Algorithm 6.2) creates a reward function consisting of the sum of Gaussians placed over each new neighbor with standard deviation d_u. Figure 6.1(d) illustrates level sets of this reward function

Algorithm 6.2 The plan refinement algorithm

```
1: function REFINE(T, p+, neighbors)
2:     ▷ Place a reward function at each neighbor
3:     rewards ← []
4:     for all n ∈ neighbors do
5:         rewards.APPEND(N(n, d_u^2))
6:     ▷ Find a local maximum in the reward function near
7:     ▷ the planned point
8:     p* ← GRADIENTASCENT(p+, rewards)
9:     ▷ Blend the refinement with the planned point
10:    return blend * p+ + (1.0 − blend) * p*
```

around the new neighbors. We use standard gradient ascent with a step size of $d_u/10$ to find a local maximum, p^*, in this reward. Rather than replacing our planned point p^+ with this local maximum, we interpolate between the two points, moving some fraction of the distance up the gradient toward p^*. In our experiments, we found that smooth plans were created by using the mean of these two points, so a $blend$ of 50% was used throughout the work in this thesis. Increasing the blend in favor of p^* results in plans that oscillate between demonstration trajectories, resulting in higher average curvature. This is likely due to the fact that the local maximum in the reward function will be coincident with one of the neighbor points when the neighbors are widely separated. Alternately, decreasing the blend to reduce the influence of p^* can can result in planning failure as the plan deviates too far from the neighbor graph as they follow the noisy motions too closely.

This approach to planning ensures that interpolation occurs only between demonstrated points that are similar in pose and graph distance: precisely where interpolation is expected to safely respect geometric and non-geometric constraints. However, a physical model of the robot permits a more explicit safety mechanism for avoiding obstacles when operating in a static environment. Using the occupancy grid calculated in Section 5.3, plans may be checked for safety during the refinement stage of planning. The current refined plan point is used to model the robot

in the occupancy grid, and we ensure that all occupied voxels have been marked as free. This check is performed at the planned point, and at several points along the path from the previous planned point to the new one. Since the step size d_u is small, only a few points along a straight line between the planned points need to be checked. Alternately, the robot kinematics may be used to calculate the precise path followed by the robot, and all of the voxels that it will occupy during this motion.

If any voxels are not free, the planned point is adjusted to move the robot out of collision. Since the previous plan point was collision-free, we know that a safe configuration exists within the length of the action step. Rather than performing an exhaustive search near the planned point, though, we use a signed distance field (SDF) [62] to efficiently precompute corrections to poses in collision. This data structure is an occupancy grid that, in addition to a bit indicating the safety of a given grid cell or voxel, provides a vector pointing to the nearest safe cell, computed using a 1-dimensional distance transform along each dimension of the workspace. This approach is useful when the collision check is performed only on the final (rigid) link of the robot, as in our experiments, since the stored vector will be used to adjust that link's position. If the entire robot must be collision-checked, the problem is more difficult. The workspace vector stored in the SDF may not be sufficient if a different link is in collision, since the corrective motion may result in another collision. The SDF may be constructed in configuration space rather than workspace, but this introduces much larger memory and computation requirements. The distance transform computed for each dimension is the same size as the configuration space, so this SDF requires seven seven-dimensional arrays at the desired resolution.

Additional information about example trajectories may be considered at this point, and incorporated into the refinement as an extension to our algorithm. For example, the wire maze domain introduced in the previous chapter uses a copper wand to traverse a copper maze. When the wand contacts the walls of the maze, a circuit is closed and a buzzer sounds as feedback for the teacher. This information is also recorded as part of the demonstration. Since we would like the robot to learn to traverse the maze without contacting the walls, these portions of the

demonstrations may be penalized and weighted lower in the refinement stage. This may be desirable for tasks in which demonstrations can contain costly or potentially catastrophic motions. In this task, we prefer not to contact the walls of the maze, but the task may still be completed if we do. Penalization was not necessary in this case because interpolation proved sufficient for the planner to avoid contacting the walls of the maze. In practice, since teachers did not contact the walls at the same points in each demonstration, the plans remained closer to the center of the corridors.

Other cost functions may be considered at this point, as well. For example, the examples may have costs or rewards associated with them if the teacher has provided qualitative feedback regarding his own performance. The robot's workspace or configuration space may also have costs associated with different regions, and plans may be adjusted to minimize these. Refining adjustments must, however, remain small relative to the step size d_u; otherwise the set of neighbor points may not remain relevant to the planned point. The safety check must be the last refinement performed, though, since any additional adjustments to the planned point may force the robot into a potentially unsafe pose.

6.4 Bifurcations

A final concern to consider when planning is bifurcations in the neighbor graph: areas where demonstration trajectories diverge into two (or more) qualitatively different strategies. Details of the bifurcation detection algorithm are presented in the next chapter, but the robot must have a strategy for planning through bifurcations once they are known. Otherwise, plans may be created that average the behavior of the two strategies, resulting in undesired behavior. The SDF safety check will prevent planning in unknown areas, but smoother trajectories can be created by recognizing bifurcations and intentionally creating plans that choose and follow a single branch.

We accomplish this by choosing neighbors for action selection in only one branch of the bifurcation. The bifurcation detection algorithm records the demonstration points in the neighborhood of each bifurcation, so

we are able to detect when the neighbor extension step of the planner reaches a bifurcation. Neighbors belonging to demonstrations that follow the undesired branch are removed and replaced with the closest points from trajectories in the intended branch.

6.5 Experimental Results

The goal of our planning algorithm is to produce safe, smooth, and efficient trajectories that perform the task demonstrated by the human teacher. Safety in static environments is guaranteed by our conservative estimation of geometric obstacles in the workspace. Evaluating whether the robot performs the same task as the teacher is largely domain-dependent. In the example domains illustrated in this section, we rely largely upon intuition to verify that the trajectories created by the learner follow the same strategy as the examples in moving through space. Our planner does not search for a path that optimizes an easily quantifiable objective function, such as the shortest obstacle-free path to the goal region, as in most traditional motion planners. Trajectory smoothness is easier to verify quantitatively. The following examples verify path smoothness by examining the curvature of example and planned trajectories. Finally, efficiency is evaluated in terms of path length. Again, our planner does not exploit obstacle-free "shortcuts" through the workspace, so planned trajectories will not be dramatically shorter than the examples, but the elimination of noise and mistakes from demonstrations produces slightly shorter plans.

Figure 6.2 illustrates a plan in the previously described maze domain. This plan compares favorably to the jagged plan of Figure 3.10, produced by the dimensionality reduction technique detailed in Chapter 3. The plot on the right is a cumulative curvature histogram. Like a cumulative distribution function, this graph illustrates the percentage of points with curvature less than or equal to the value on the horizontal axis. The vertical axis indicates the proportion of sample points (from 0% to 100%) in each trajectory with a curvature value less than or equal to the corresponding value on the horizontal axis. Thus, curves shifted to the left side of the graph correspond to trajectories with lower overall

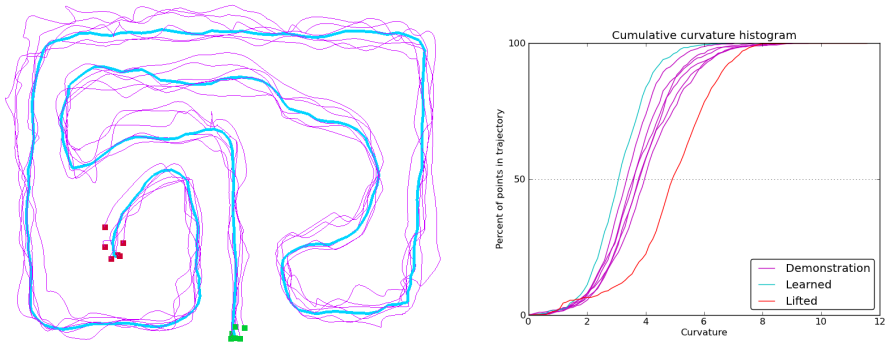


Figure 6.2: The example trajectories (purple) from Figure 3.10 with a new plan in blue. The graph on the right shows a cumulative curvature histogram plot, illustrating discontinuities of the lifted plan (in red), while the new plan (in cyan) exhibits less curvature than the examples.

curvature. The red trace on this plot, representing the earlier plan lifted from the latent space (shown in Figure 3.10) is shifted to the right of the example trajectories, indicating that more points in this trajectory have higher curvature values. This is expected since the lifted plan consists of piecewise linear segments with second-order discontinuities between them. The cyan line, representing the new plan, is just to the left of the example trajectories. This indicates that the shape of the new plan is consistent with the examples provided, but with noticeably less curvature. Also, note that the planned path tends to traverse areas of densely clustered example trajectories. The refinement step of the planning algorithm seeks to follow these clusters.

Figure 6.3 shows the results achieved with increasing numbers of example trajectories from an inexperienced robot user in this maze scenario, and Table 6.1 summarizes the statistics in the plot. Plans were created from the first two demonstrations provided, and with each subsequent demonstration, from all of the demonstrations collected so far. Since our planner is, in a sense, averaging the demonstrations from which it produces plans, it is unsurprising that increasing the number of demonstrations creates smoother plans. With a small number of demonstrations, though, the planner produces plans with higher curvature than any single demonstration. This is likely because the plan

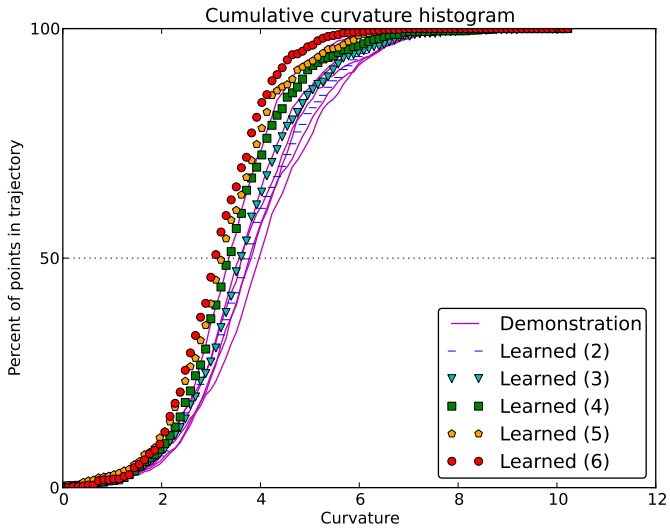


Figure 6.3: The curvature decreases as more example trajectories are added in the wire maze domain.

	Curvature	Cumulative Curvature			
Label	Mean	25%	50%	75%	90%
All Demonstrations	3.86	3.03	3.77	4.63	5.54
Demonstrations	3.46	2.80	3.43	4.11	4.74
	3.72	2.96	3.68	4.41	5.32
	3.81	2.97	3.66	4.60	5.57
	4.00	3.11	3.86	4.83	5.75
	4.10	3.22	4.03	5.00	5.87
	3.87	3.08	3.83	4.67	5.27
Learned (2)	3.86	2.94	3.80	4.71	5.66
Learned (3)	3.76	2.98	3.67	4.44	5.45
Learned (4)	3.52	2.78	3.41	4.17	4.97
Learned (5)	3.33	2.62	3.28	4.00	4.74
Learned (6)	3.20	2.55	3.16	3.83	4.41

Table 6.1: Curvature statistics for increasing numbers of example trajectories.

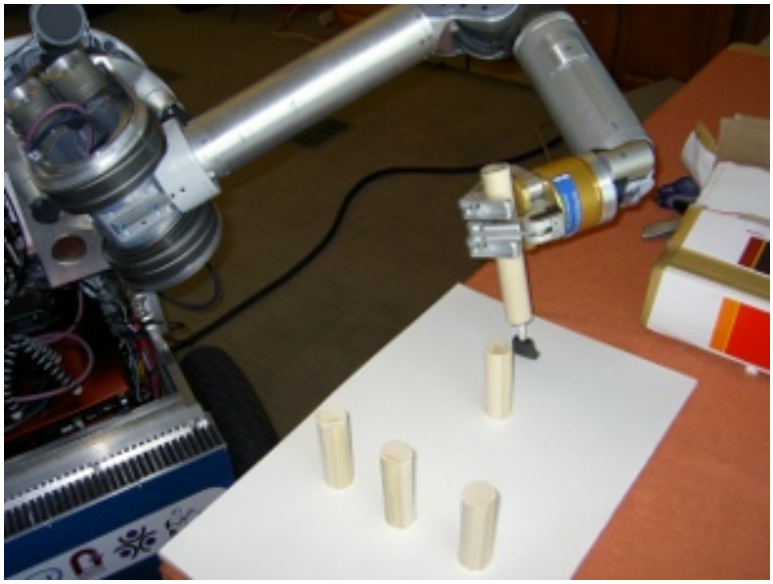
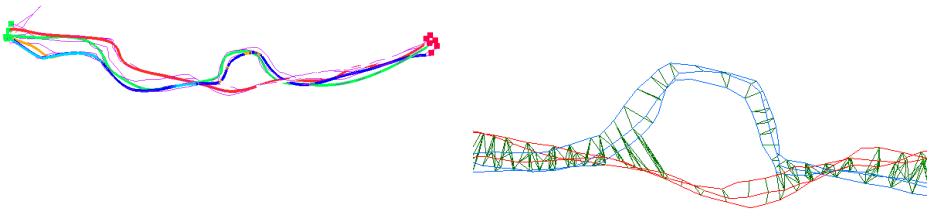


Figure 6.4: The artist domain, in which the robot was to sweep a paint brush across a board while avoiding obstacles.

oscillates between following the demonstrations in a very sparsely-filled workspace. Once more demonstrations are available, the planner is able to produce smoother plans.

In another set of experiments, the robot operated over a flat horizontal board with vertical obstacles protruding. In this *artist* domain (Figure 6.4), the robot holds a paint brush and draws a path across the board, avoiding the obstacles as demonstrated by the teacher. This provides an opportunity to investigate the detection of bifurcations in examples and the strategy for planning through them. First, though, we consider some example trajectories with a bifurcation created by a perturbation too small to represent a geometric obstacle in the robot's workspace. Figure 6.5 shows some demonstration and planned trajectories that illustrate such a perturbation. The neighbor graph, shown on the right, illustrates the separation between the two demonstrated strategies. Although the distance between the strategies is small, the deviation is sufficient to prevent neighbor links between the two sets of examples. The planner, therefore, does not interpolate between them.



(a) (b)

Figure 6.5: A simple linear trajectory in which some of the examples deviate slightly in the middle of the path. The SDF safety check is disabled for this example to show that the bifurcation is detected even without concern for obstacle avoidance. On the left, example plans are shown (thick lines) in each branch of the bifurcation, using the bifurcation detection described in Chapter 7. The image on the right shows the neighbor graph for the area of interest.

As above, the cumulative curvature histogram (Figure 6.6) shows that planned trajectories are smoother than the demonstrations.

Next, the obstacles are used to create distict strategies for moving across the board. The artist domain permits multiple strategies for traversing the board while avoiding the dowel rod obstacles. For example, Figure 6.7 illustrates a simple set of demonstrations traversing past a single obstacle. The figure on the left illustrates a set of examples that all follow the same strategy: they loop around the obstacle, then overlap themselves before continuing to the goal. This makes the workspace non-Markovian, because the correct action in the overlapping states depends on the current progress through the task. This could cause difficulties for planners that create policies that directly map states to actions. Our neighbor graph, though, captures this information, so our planner is able to correctly plan a loop around the obstacle. A planned path is shown in green.

In the right-hand figure, Figure 6.7(b), three demonstrations in red that simply pass by the obstacle are added to the three purple trajecto-

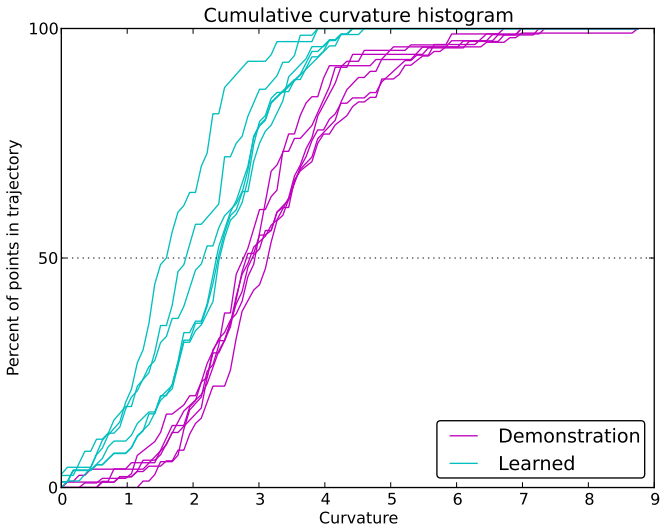
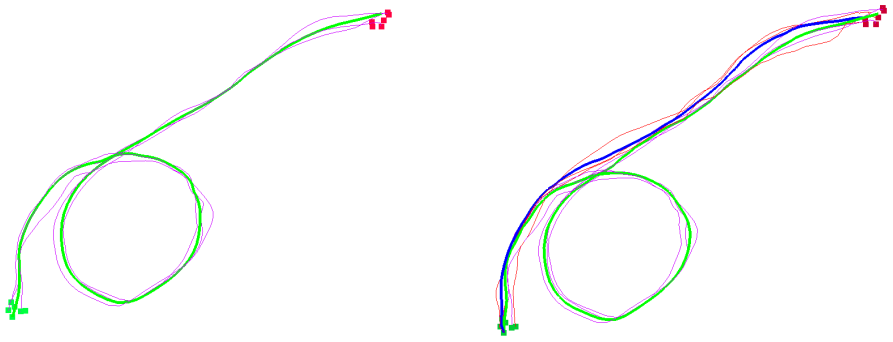


Figure 6.6: The cumulative curvature histogram for the perturbation example of Figure 6.5.

ries that looped around it. The thicker blue and green paths represent plans created by the learner following one of these two strategies. In this case, a bifurcation is detected (by the method detailed in the next chapter), and two planning strategies are possible. In essence, while passing through a bifurcation in the neighbor graph, points from trajectories in one of the strategies will be removed from the list of neighbors used for planning, thus causing the plan to follow the demonstrations from the other strategy. Once more, the cumulative curvature histogram is provided, in Figure 6.8.

As a final example, we examine a set of examples with multiple bifurcations. Figure 6.9(a) shows some demonstrations in the artist domain in which multiple strategies are used to traverse between, and even over, the dowel rod obstacles. Figure 6.9(b) shows a close-up view of a typical bifurcation, in which the demonstrations are evenly divided in their strategy for avoiding an obstacle. We have three demonstrations of each of the four strategies, and we created twelve plans distributed similarly. The plans are shown in various colors in Figure 6.10(a), and the corre-



(a) (b)

Figure 6.7: Planning in non-Markovian space. The example trajectories (thin purple lines) traverse a loop around an obstacle, which the plan (thick green line) follows. When additional examples that bypass the obstacle are provided (in red), a bifurcation is detected. The blue plan follows the new branch.

Scenario	N	Demonstrations		Learned	
		mean	median	mean	median
maze (Figure 6.2)	6	3.8261	3.7490	3.1980	3.1611
perturbation (Figure 6.5)	6				
upper path	3	3.1900	3.0265	2.3327	2.3508
lower path	3	3.0587	2.9330	2.0510	2.0373
loop (Figure 6.7)	6				
looping path	3	2.5379	2.4386	2.1028	2.1056
non-looping path	3	2.5411	2.4132	1.3797	1.4670
splits (Figure 6.9)	12				
outer paths	6	3.2020	3.0766	2.3422	2.3351
inner paths	6	3.3520	3.2021	2.3792	2.3437
artist (Section 7.3)	18	3.6442	2.8269	1.8244	1.4268

Table 6.2: Trajectory curvature statistics

sponding curvature histogram is shown in Figure 6.10(b).

Table 6.2 summarizes the numeric results of the curvature plots presented in this chapter. The final line presents results from user trials to be presented in Chapter 7, in which teachers of varying inexperience operated the robot in the artist domain. The number of demonstration

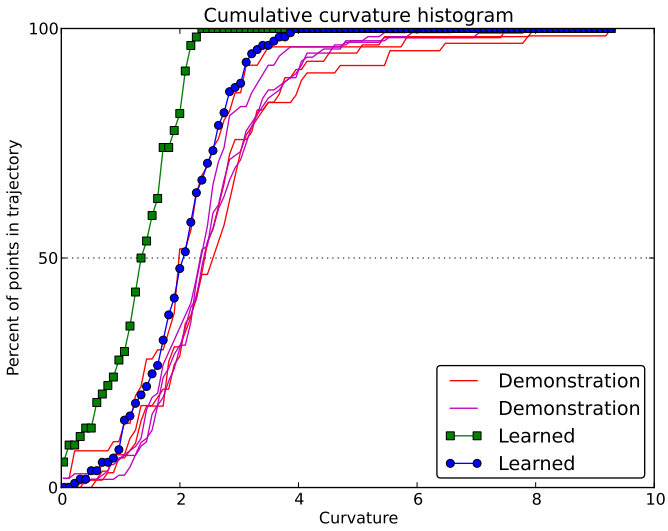
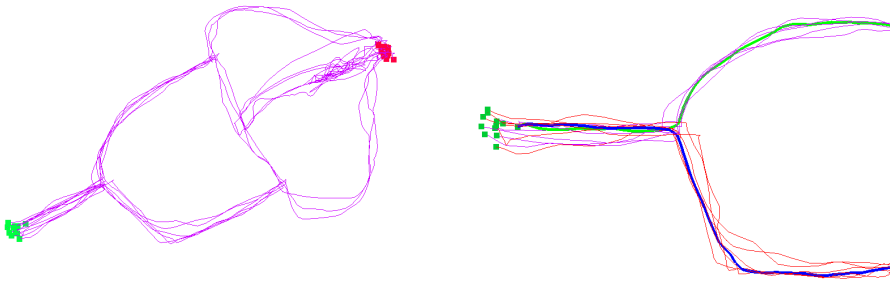


Figure 6.8: The cumulative curvature histogram for looping domain. Colors correspond to trajectory colors in Figure 6.7. The learned path following the loop is marked with green squares, and the other learned path is marked with blue circles.

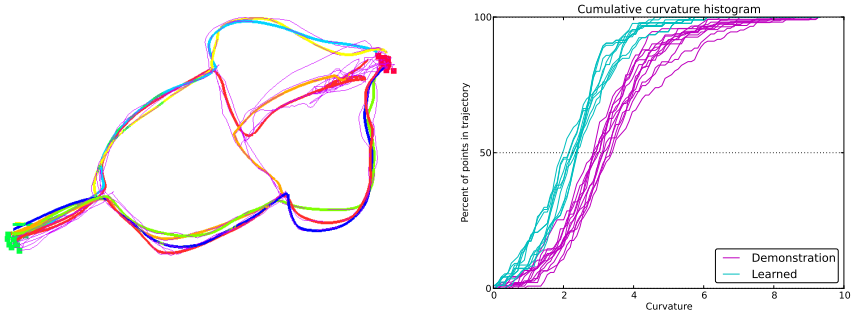
trajectories in each scenario is listed, and an equal number of plans was created in each case. One planned path was started from the start point of each demonstration path. This table presents the mean curvature for demonstration and learned trajectories, as well as the median curvature, which is the value of the corresponding plots at 50% on the vertical axis. In all cases, the reduction in curvature is substantial. On average, there is a 33.9% reduction in the mean curvature, with the most pronounced improvement of 50% in the artist domain, where the task and workspace were far less constrained.

Finally, we briefly examine the efficiency of the plans created. Table 6.3 lists statistics for the lengths of demonstration and planned trajectories for the domains discussed above. As in the previous table, the number of demonstration trajectories and learned trajectories is listed. One planned path was started from the start point of each demonstration path. The small variations in starting position results in variations



(a) (b)

Figure 6.9: An example in the artist domain with multiple bifurcations. The example trajectories are shown on the left, and the right image shows a close-up view of plans following both branches of the first bifurcation.



(a) (b)

Figure 6.10: The thick lines in this image represent plans initialized at the starting location of each of the examples, and following the branches of the corresponding example. The cumulative curvature histogram clearly demonstrates the smoothness of the planned paths.

in the trajectories of each plan. The learned trajectories are, on average, shorter in every case than the demonstration trajectories, with an average reduction of 10.5% in path length. This result makes sense in light of the trajectory examples and curvature results presented in this section. Planned paths closely reproduce the shape of example paths, but

Scenario	N	Demonstrations (mean, std. dev.)	Learned (mean, std. dev.)
maze (Figure 6.2)	6	(1.5206, 0.0529)	(1.5000, 0.0596)
perturbation (Figure 6.5)	6		
upper path	3	(0.6624, 0.0512)	(0.5963, 0.0047)
lower path	3	(0.5807, 0.0048)	(0.5689, 0.0149)
loop (Figure 6.7)	6		
looping path	3	(1.1327, 0.0086)	(1.1274, 0.0083)
non-looping path	3	(0.5950, 0.0199)	(0.5902, 0.0102)
splits (Figure 6.9)	12		
outer paths	6	(0.7397, 0.0320)	(0.6883, 0.0231)
inner paths	6	(1.0062, 0.0494)	(0.7152, 0.0902)
artist (Section 7.3)	18	(0.7015, 0.2112)	(0.6080, 0.1540)

Table 6.3: Trajectory length statistics

with less curvature. Thus, the planned paths are also slightly shorter.

6.6 Summary

The PbD motion planning algorithm described here is able to successfully create safe, smooth, novel manipulator plans using only demonstrations provided by a domain expert. The planned robot state is related to the demonstrations, and the demonstrated actions are used to compute a new plan action. However, the demonstration trajectories are imperfect and their relationships to one another change over time. Moreover, the examples can demonstrate multiple strategies for completing the task. Planned paths must follow only one branch of bifurcating strategies. Thus, planning requires more than simply averaging demonstrated behavior over time, or even learning a single behavior corresponding to each possible state.

Instead, the planning algorithm advances through the neighbor graph, which encodes information about the relationships between portions of the demonstration trajectories. While these demonstrations do not necessarily behave similarly as they progress through *time*, the neighbor graph relationships allow the planner to exploit their similarities as they progress through the *task*. The new plan follows these common features

that are essential to successfully executing the motion task.

This approach offers advantages even when a model of the environment is available. In addition to merely avoiding obstacles (as a conventional motion planner would do), non-geometric constraints are observed as well. Rather than searching for a collision-free path that optimizes an objective function articulated by a programmer, this planner follows the routes of example trajectories provided by anyone able to demonstrate the task. It respects the constraints demonstrated while eliding motions that are significantly dissimilar from other demonstrations, and thus disconnected in the neighbor graph. In our experiments, this resulted in planned paths with a 33.9% reduction in mean curvature and a 10.5% reduction in path length.

Chapter 7
Active Learning

Can the maker repair what he makes?

— Roy Batty, *Blade Runner*, 1982

The planning approach described so far is capable of creating high-quality trajectories from imperfect demonstrations. Jitter and other undesirable artifacts in the demonstrations can be avoided by the planner largely because small-scale incorrect movements in the examples are unique, and therefore poorly connected in the neighbor graph. Thus, they exert less influence on the planner.

In many situations, though, deviations in strategy are not so unique. Differing strategies are each reinforced with sets of neighboring demonstrations performing similar actions. In practical terms, the teacher has provided demonstrations that perform qualitatively different actions at the same point in the task. Whether these differences are intentional or not, the learner must decide which of the conflicting strategies to follow when planning through these states. This choice may be random, weighted by the number of examples of each strategy, or based on some statistic relating the examples in each group. However, none of these approaches is likely to provide desired behavior in all applications.

Instead, strategy decisions must be made on a case-by-case basis. The ease-of-use of PbD methods is premised on the idea that human teachers can demonstrate and recognize correct performances even if they cannot formulate the basis for their judgement in mathematic or programmatic terms. Just as the human teacher provides demonstrations that provide the basis for executing individual tasks, the advice of the teacher should be sought to resolve choices in creating new plans.

In our approach, the learner creates and demonstrates candidate plans that follow each possible strategy. The teacher is asked to specify which strategy is to be preferred for creating future plans.

This chapter first presents our algorithm for detecting *bifurcations*, states in which demonstration trajectories diverge into qualitatively distinct planning strategies. Next, we describe our strategy for interacting with the teacher and requesting advice for resolving the apparently conflicting demonstrations. Finally, Section 7.3 presents results from user trials in the artist domain. Naïve and experienced robot users taught the robot to sweep a paint brush across a poster board while avoiding obstacles. When bifurcations were detected, users were asked to resolve the learner's confusion so that new trajectories could be planned. Results show that this approach easily and effectively allowed users to train the robot to imitate their demonstrations.

7.1 Detecting Diverging Demonstrations

The planning algorithm described in the previous chapter relies heavily on finding and exploiting commonality between multiple demonstration trajectories. The points of demonstration trajectories are organized by a neighbor graph that ensures connected portions of trajectories are close to one another, moving in the same direction, and may safely be interpolated without concern for physical obstacles. The planner, in turn, creates trajectories that remain close to clusters of demonstrations. These algorithms have been designed to ensure that novel plans duplicate the characteristics common to the majority of demonstrations and avoid features unique to a single trajectory. These unique features are considered to be errors in the examples.

Section 5.2 discussed some typical errors that separate individual example trajectories from their neighbors, including backtracking and transient deviations. More obvious bifurcations, such as that illustrated in Figure 7.1, also result in a lack of neighbor links between demonstrations in different branches of the bifurcation. When these gaps appear in the neighbor graph, the planner is not able to follow both branches at once. Specifically, the plan refinement stage of the planner will cause

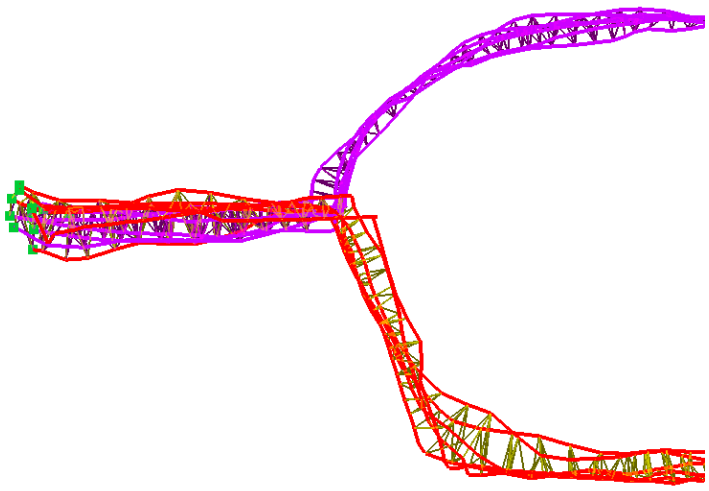


Figure 7.1: This bifurcation in demonstrations is intuitively obvious from the paths of the trajectories, and is represented by the neighbor graph. Plans created in each branch of the bifurcation are shown in Figure 6.9(b).

new trajectory points to gravitate toward one cluster of demonstration points, and subsequent neighbor extensions will select neighbors nearby and following this cluster in the graph. Since the refinement seeks only a local optimum in terms of nearby clusters, small changes in the previous plan state can cause the planner to choose a different branch of the bifurcation. This seemingly chaotic behavior provides another motivation for purposefully detecting and resolving bifurcations.

Obvious bifurcations, such as the one shown above, are typically caused by constraints that restrict paths into a limited number of discrete options. Although there is variation within the options (as there is between all demonstrations), it is noticeably less than the variation between options. These sorts of bifurcations may occur due to physical obstacles in the workspace or other geometric or non-geometric constraints. For example, constraints on end-effector orientation coupled with manipulator redundancy may provide two or more options for following a particular

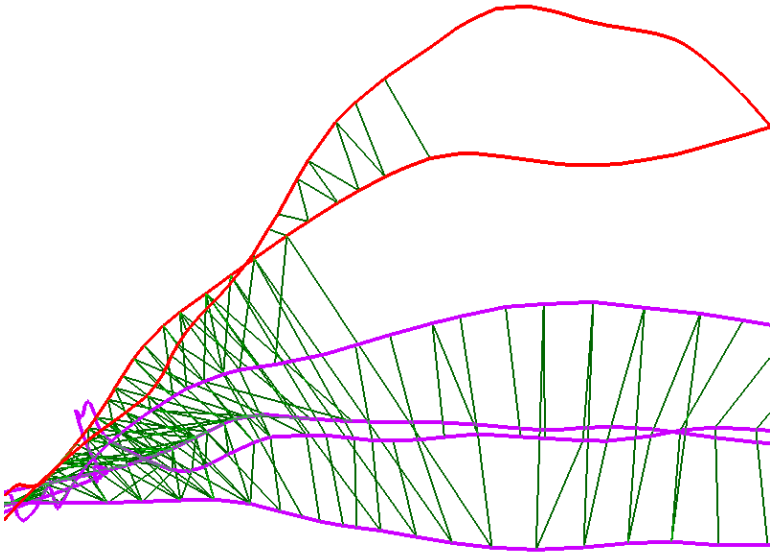


Figure 7.2: Some bifurcations appear due to unintentional clustering of demonstrations in tasks without true constraints. In this example, a bifurcation has been detected between the red and the purple paths.

workspace path. In this case, no physical obstacle may be present between the two clusters of trajectories, but their directions of movement are so widely separated that they must represent different strategies.

In other cases, such as the demonstrations shown in Figure 7.2, example trajectories may be widely separated despite a lack of obstacles forcing them apart. In this case, the freedom to vary the path taken from left to right in the image has resulted in an artificial distinction between the similar red and purple paths. These demonstrations are so widely separated that a physical obstacle may exist between them. The bifurcation detection algorithm described below thus detects two different strategies in the demonstration trajectories, and the teacher will be asked to resolve the apparent conflict. This example raises an interesting complication to the task of resolving bifurcations: it is not sufficient to allow the teacher to simply choose between the two strategies detected by the learner. In this case, the bifurcation appears only due to lack of information, a deficiency in the demonstrations provided to the learner.

The solution to this dilemma is to provide additional information. The mechanism for doing so will be discussed in Section 7.2.

Finally, we note that the presence of an obstacle is not as reliable an indication of bifurcation as it may seem. As discussed in Section 5.3, the concept of homotopy makes planar trajectories in the presence of obstacles easy to distinguish. In higher dimensions, though, obstacles are more easily circumvented. For example, even if an obstacle truly exists between the differently colored trajectories of Figure 7.2, a bifurcation does not necessarily exist between the sets of trajectories. As illustrated in Figure 5.4, if the obstacle were confined to the page, or the half-space below the page, a safe and valid path may exist just above the page. This new trajectory would be similar (in fact, homotopic) to one of the red trajectories, and one of the purple trajectories. A simple interpolation between red and purple paths would be confined to the plane of the page, and thus doomed to pass through the hypothetical obstacle. A smooth interpolation through the new trajectory would avoid the obstacle, though, and show that all the trajectories are, in fact, homotopic.

Using the SDF of Section 6.3 to detect bifurcations is a tempting, but incomplete, approach. We might produce trial robot poses by interpolating between two trajectories, then test whether those poses are safe. However, this will detect only bifurcations caused by physical obstacles. For a more general approach, we rely upon the neighbor graph. A bifurcation in demonstration strategies should appear as a localized partition in the graph. Example trajectories that are linked in the graph at some point in time will no longer be linked after the bifurcation.

Thus, the presence of a physical obstacle in the robot's workspace, even when it appears between demonstration trajectories, is neither necessary nor sufficient to require a bifurcation between these trajectories. An obstacle is only one indication that a bifurcation may exist. Groups of example trajectories may diverge for other reasons, so we should search for the presence of divergences between clusters of trajectories. Typical approaches to clustering such as k-means [24] or hierarchical clustering [66] are difficult to apply since they do not account for time-series data such as trajectories. If the split between strategies is not sufficiently abrupt, these approaches cannot precisely localize the bifurcation – if it

is detected at all.

Our strategy for detecting bifurcations depends on the neighbor graph constructed in Chapter 5. This graph contains links between similar portions of nearby trajectories, but no links are formed when the trajectories diverge from one another. The neighbor graph captures the separation between demonstration trajectories caused by physical obstacles as well as other causes. As shown in Figure 7.1, the neighbor graph also captures separations caused by trajectories moving in divergent directions, as well as the errors and imperfections described in Section 5.2. Of course, not all of these situation correspond to diverging strategies. In particular, the learner should not ask for clarification regarding every imperfection and missing neighbor link between nearby trajectories.

Instead, the learner must determine which missing neighbor links represent bifurcations worthy of investigation. Such bifurcations are characterized by an abrupt loss of neighbors between sets of trajectories. Prior to the bifurcation, the presence of links between the trajectories indicates that the same strategy is being followed in all cases. Following the bifurcation, links should cease between trajectories following different strategies, but should still exist within a single strategy.

This description suggests the use of a graph partitioning algorithm to detect bifurcations. Such algorithms are frequently used for image segmentation and other applications in which divisions must be found between collections of interconnected nodes. We use Shi and Malik's normalized cuts [63] graph partitioning algorithm to detect these locations. This algorithm compares the number of links within a partition, or cluster, to the number of links between clusters. The links between clusters are cut to form partitions. A good partition should cut relatively few links, but a large number of links should exist within clusters. NormalizedCut computes a score based on the number of cut links and the number remaining within clusters. The procedure, presented in Algorithm 7.1, takes an variant of an adjacency matrix as input, computes a score for every possible partitioning, and returns the highest-scoring partition.

The adjacency matrix used does not describe links between all the points in the trajectory collection, T, because we are not searching for

Algorithm 7.1 The Normalized Cut algorithm

```
1: function NormalizedCut(adj)
2:     n ← size(adj)
3:     best_score ← −∞
4:     for all permutations p ∈ {0, 1}^n do
5:         ▷ Determine degree of each vertex for normalization
6:         degree[0] = Σ_{i∈{p[i]=0}} adj[i][i]
7:         degree[1] = Σ_{i∈{p[i]=1}} adj[i][i]
8:         assoc_links ← cut_links ← {0, 0}

9:         ▷ For each pair of vertices...
10:        for all i ∈ 0..n do
11:            for all j ∈ i..n do
12:                if p[i] == p[j] then
13:                    ▷ ... accumulate a score in assoc_links if
14:                    ▷ in the same partition
15:                    assoc_links[p[i]] += 2 * adj[i][j] / degree[p[i]]
16:                else
17:                    ▷ ... or cut_links otherwise.
18:                    cut_links[p[i]] += adj[i][j] / degree[p[i]]
19:                    cut_links[p[j]] += adj[j][i] / degree[p[j]]

20:        ▷ Keep track of the best score
21:        score ← assoc_links[0] + assoc_links[1] − cut_links[0] −
   cut_links[1]
22:        if score > best_score then
23:            best_score ← score
24:            best_cut ← p

25:    return best_cut
```

an arbitrary division between these points. We are only interested in a subset of the neighbor graph. In addition, all the points of a single trajectory must be assigned to one partition or the other. To simplify the information provided to the NormalizedCut algorithm, we build a *meta-graph* in which each vertex is a trajectory. The adjacency matrix passed to Algorithm 7.1 contains the number of iter-trajectory links between each pair of trajectories in each off-diagonal element, and the *degree* (total number of links) of within each vertex on the diagonal (for use in nomalization).

Typically, normalized cuts are computed on large graphs, so an initial partition must be estimated by some means, and simulated annealing is used to find a locally optimal score. In our case, though, the size of the partitioning problem is significantly reduced. Although the neighbor graph of demonstration trajectory points may contain tens of thousands of points, we are only searching for partitions between trajectories. As in most approaches to PbD, our approach seeks to operate with a small number of examples, thus limiting the amount of time the teacher needs to spend during training. Our experiments contained a dozen or fewer example trajectories, so we can easily enumerate and test all possible partitions to find the globally optimal cut.

As described so far, the NormalizedCut algorithm could be used to separate entire demonstration trajectories from one another. However, we expect the demonstrations to be similar (and thus well-connected) during some parts of the task, and to diverge in other parts. Our goal is to find the areas of divergence. Thus, we want to compute cuts within a sliding window of trajectory points. This presents some considerable challenges, though. The foremost is determining which points from which trajectories are included in the sliding window. Since trajectories are different lengths and unaligned in time, we cannot simply choose the same time indices from all examples. Furthermore, even the neighbor graph does not appear (at first) to be useful since we are searching for areas in which trajectories are *not* linked. However, the trajectories must be linked prior to the bifurcation. We hope to detect situations in which trajectories that were well-connected at some point in time become disconnected later. Our sliding-window strategy, therefore, is to compare

the connections between trajectories at each given point to the connections in the future. We then look for clusters of points that lose these connections to neighboring trajectories. These clusters, which we call *cut neighborhoods*, indicate regions of the example trajectories just prior to bifurcations.

The procedure is detailed in Algorithm 7.2. The first loop in this pseudocode searches for likely bifurcation points: places where the number of neighboring demonstration trajectories decreases. Because the neighbor graph is noisy, we search for a local minimum in a small range of points following a local maximum. We only record values for points with more neighboring trajectories than *all* of the next δ_1 points. This approach allows us to ignore transient or intermittent connections between trajectories. In our experiments, $\delta_1 = 3$ worked well for all tested scenarios. The $missing$ array keeps track of the differences between these local extrema.

In the next loop, the potential cut neighborhoods are collected in the $neighborhoods$ set. These neighborhoods keep track of the demonstration trajectory points within them and a score, which is the sum of differences calculated in the previous loop. The FindOrCreate create function called from this loop (and listed in Algorithm 7.3) is responsible for coalescing suspected bifurcation points into neighborhoods. Specifically, a point is added to an existing neighborhood if it is less than δ_1 links away (in the original neighbor graph represented by adj) from any point in that neighborhood. Otherwise, a new neighborhood is created. Collecting points into neighborhoods serves two purposes: it prevents multiple detections of the same bifurcation, and ensures that the meta-graph contains information from all trajectories involved in the bifurcation. Each neighborhood computes and stores its adjacency matrix, representing the meta-graph of connections between demonstration trajectories up to δ_2 points in the future. A larger δ value ($\delta_2 = 10$ in our experiments) produces a denser meta-graph for NormalizedCut to examine, and leads to more accurate results. We create a symmetric adjacency matrix with the degree of each vertex in the diagonal elements.

In the final loop, the NormalizedCut algorithm (listed in Algorithm 7.1) is used to compute the partitioning of demonstration trajectories as de-

Algorithm 7.2 The learning procedure

```
1: function DetectBifurcations(T, adj)
2:     ▷ Compute path lengths through adjacency graph
3:     adj.AllPairsShortestPaths( )

4:     ▷ Find local maxima followed by local minima in number of
5:     ▷ neighboring demonstrations
6:     for all pt ∈ T do
7:         nn[0..δ1] ← NumNeighbors(pt..pt + δ1)
8:         if nn[0] > nn[1..δ1] and nn[0] ≥ NumNeighbors(pt − 1) then
9:             missing[pt] ← nn[0]− min(nn[1..δ1])
10:        else
11:            missing[pt] ← 0

12:    ▷ Coalesce neighborhoods of points with missing neighbors
13:    for i ∈ T where missing[i] ≠ 0 do
14:        pts ← {T[i]} ∪ adj[T[i]]
15:        neighborhood ← FindOrCreate(T, neighborhoods, adj, T[i])
16:        neighborhood.pts ← neighborhood.pts ∪ pts
17:        neighborhood.score ← neighborhood.score + missing[i]
18:        ▷ Build a meta-graph for the neighborhood
19:        for all pt ∈ pts do
20:            n ← T.demonstration_number(pt)
21:            for all p ∈ pt..pt + δ2 do
22:                for all neighbor ∈ adj[p] do
23:                    m ← T.demonstration_number(neighbor)
24:                    neighborhood.adj_matrix[n][n] + +
25:                    neighborhood.adj_matrix[n][m] + +
26:                    neighborhood.adj_matrix[m][n] + +

27:    ▷ Use Normalized Cuts to determine bifurcations
28:    bifurcations ← {}
29:    for all neighborhood ∈ neighborhoods do
30:        labels ← NormalizedCut(neighborhood.adj_matrix)
31:        if labels[0] ≠ ∅ and labels[1] ≠ ∅ then
32:            bifurcations ← bifurcations ∪ {neighborhood.pts, labels}

33:    return bifurcations
```

scribed above. Recall that the $labels$ array returned by NormalizedCut provides the meta-graph vertices (that is, trajectories of T) that belong to the two partitions. Note that the final test ensures that at least one demonstration has been assigned to each of the two branches of the bifurcation. This is necessary because some neighborhoods may be found with large values in the $missing$ array even though the highest-scoring normalized cut keeps all demonstrations in the same group. This is likely to occur in situations in which the neighbor graph is especially noisy. If many neighboring demonstration trajectories are becoming disconnected in the neighbor graph in a random fashion, high values will appear in the $missing$ array. However, a good normalized cut will still have strong interconnections with branches with fewer connections between them.

Algorithm 7.3 The learning procedure helper function

```
1:  function FindOrCreate(T, neighborhoods, adj, pt)
2:      for all neighborhood ∈ neighborhoods do
3:          for all n ∈ neighborhood do
4:              if adj.ShortestPath(pt, n) < δ_1 then
5:                  return neighborhood
6:      empty_neighborhood ← new Neighborhood
7:      empty_neighborhood.pts ← {}
8:      empty_neighborhood.score ← 0
9:      empty_neighborhood.adj_matrix ← Ø_{|T|×|T|}
10:     neighborhoods ← neighborhoods ∪ empty_neighborhood
11:     return empty_neighborhood
```

7.2 Requesting Advice

Having located areas of diverging demonstrations, the robot learner must consider which strategy to use when planning. However, it is unlikely to be able to make a choice that is reasonable in all scenarios without additional input from the teacher. Without any additional information, the learner may choose randomly among available strategies. Any strategy will allow the learner to reach its goal and complete its task, even if suboptimally. Indeed, our planner has no notion of optimality beyond

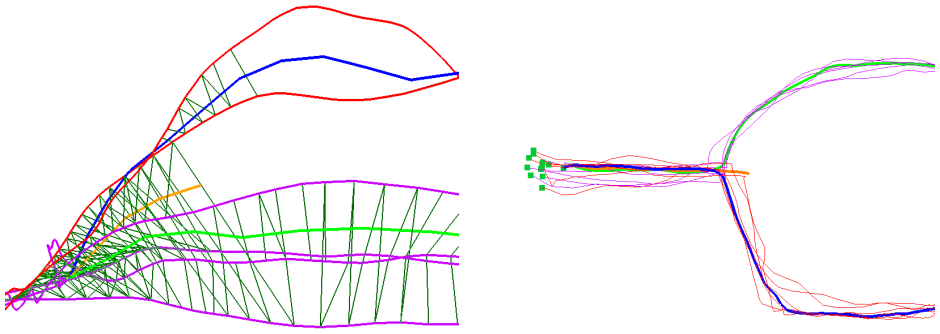
similarity to demonstrations. Since all candidate strategies are defined by demonstrations, this formulation of optimality does not help choose between them. Additionally, we would prefer to create predictable robot behaviors rather than random ones.

Furthermore, unlike a traditional motion planner, our learner has no objective function to optimize, nor do we wish to impose one to solve the current dilemma. Having the teacher specify a cost function over workspace or configuration space may provide a solution, but would be a daunting task for non-programmers who have turned to PbD methods for teaching tasks to robots. A simple global objective such as path length may provide the desired outcome for some tasks, but not all.

Alternately, we may consider computing some statistic of the diverging sets of demonstrations. As a simple (apparent) improvement to the random choice strategy just described, we might bias the random choice based on the number of demonstrations in each set. Or, the learner may choose the more consistent set of examples by determining which demonstrations are more tightly clustered. Again, though, there is no reason to believe that such simple heuristics provide preferable behavior in all situations. The teacher may demonstrate a greater number of more consistent examples in some cases by chance, or even due to the convenience of demonstrating certain strategies. No single statistic can predict the preferable behavior in all areas of diverging strategy.

Thus, we rely on an active learner, which requests additional information when required. When suspected bifurcations are detected, the robot learner asks the human teacher which branch of the bifurcation is to be preferred. Rather than present ambiguous graphical representations of the trajectories on a computer screen, the robot learner executes partial trajectory plans to demonstrate the branches of the bifurcation. This way, the teacher is given a clear conception of what the robot plans to do in the uncertain situation. The teacher may indicate that one plan or the other is to be preferred, and future plans will follow that branch of the bifurcation. That is, when a future plan encounters points from the *cut neighborhood*, points from the preferred branch are used as neighbors, but points from the other branch are not. The teacher may also indicate no preference, in which case the planner can do no better than

to choose a random branch, weighted by the number of demonstrations provided in each branch.



(a) (b)

Figure 7.3: If the teacher asserts no difference between the branches of a bifurcation, an averaged plan is executed until it is no longer safe.

A final possibility is that the teacher discerns no distinction between the two branches. This may occur in situations such as that illustrated by Figure 7.2, where the provided trajectories are widely separated, causing the robot to infer a constraint where none exists. In this case, the planner would benefit from receiving additional information, so the robot requests a new example demonstrating the possibility of planning in the unknown region. The learner solicits this additional demonstration by creating a partial plan that approaches the bifurcation and proceeds through it, averaging the two demonstrations just performed, until it reaches a point no longer known to be safe. At this point, the teacher is asked to continue the demonstration, essentially proving the equivalence of the two strategies by providing a trajectory that bridges the gap between the previous examples. If the learner continues to detect a bifurcation, additional examples may be required. Figure 7.3 includes two cases in which an orange plan is produced as the average of the two plans in the detected bifurcation. The plan is cut short because the average of the two branches leads into an area of the workspace not explored by demonstration trajectories. Since obstacles may be present, the learner must stop and ask the teacher to continue the demonstration. The new

Figure 7.4: Demonstrated strategies may bifurcate multiple times.

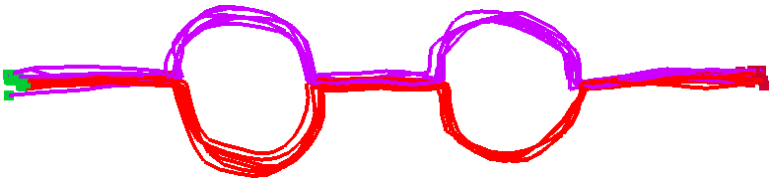


Figure 7.5: Example trajectories that split into the same groups twice.

demonstration will be added to the set of training examples.

When multiple bifurcations appear in a scenario, such as the one in Figure 7.4, questions are presented to the teacher in topological order. The first encountered bifurcation is examined first because the resolution to the learner's uncertainty here may obviate the need for additional questions. For example, the teacher may express a preference to avoid a branch of a bifurcation with additional bifurcations. If the resolution to an early question requires an additional example, later bifurcations may be formed or resolved by the new example. If later bifurcations appear in both branches of a bifurcation in which the teacher has indicated no preference, the order of the questions is not critical.

There is a minor optimization in question-asking policy that may appear advantageous in scenarios such as that shown in Figure 7.5. The examples split into two groups to avoid an obstacle, then rejoin on the other side. Upon approaching another obstacle, the examples again split into the same groups to avoid it. In this illustration, it appears that the desired behavior at the second bifurcation is precisely correlated to the behavior at the first. However, we avoid making this determination during our learning procedure. If the planner is able to find a safe path transitioning from one strategy to the other, the learner is allowed to consider such an option. This allows the learner to combine portions of examples that follow distinct strategies into combinations never demonstrated by the teacher. This may be advantageous, for example, if the teacher remained on one side of the obstacles due to visibility concerns during training.

The procedure for requesting advice from the teacher requires the creation and execution of two plans, or partial plans, in the vicinity of the bifurcation. Since the bifurcation specifies the neighborhood of points that is the area of interest, we can begin the planner at the start (as usual), plan through one branch, and stop some short distance after the bifurcation. Because we operate in domains that do not have dynamic constraints, we can then execute the plan in reverse until a short distance prior to the bifurcation, then create and execute a partial plan in the other branch. To extend this approach for handling dynamic constraints, we could execute full plans, though some other means would be necessary to draw the teacher's attention to the area in which the bifurcation is traversed. In the experiment described below, we demonstrated plans that extended 10 cm before and after the bifurcation, but this distance can easily be tuned to match the physical characteristics of a given task.

To plan through a single branch, we need only modify the neighbor extension procedure from Section 6.2 to remove neighbors that are neighborhood points in the *other* partition. Note that this does not ignore *all* points from the trajectories assigned to the other branch, only those in the vicinity of the bifurcation.

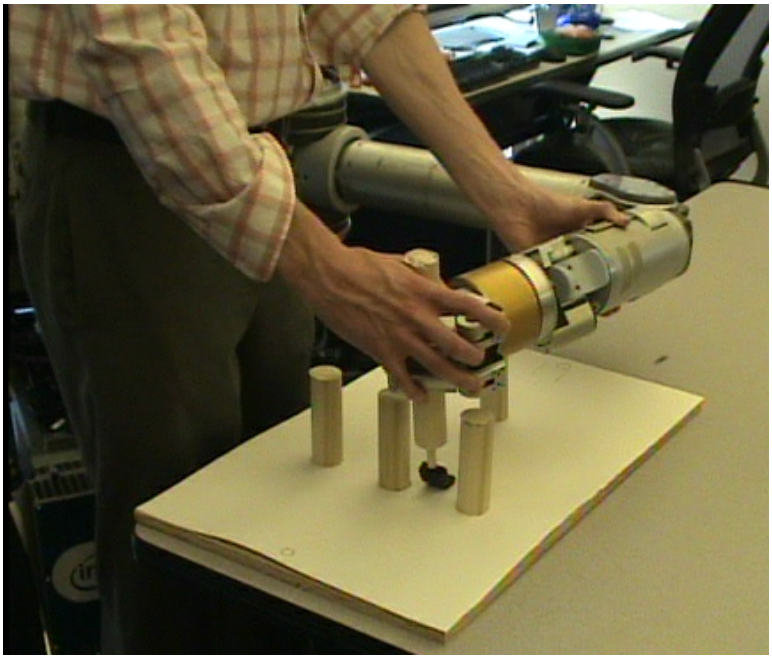


Figure 7.6: User trials were conducted in the artist domain.

7.3 Experimental Results

User trials were conducted in the artist domain described in the previous chapter. Experiment participants were asked to perform the same task described earlier: to teach the robot to traverse the poster board with a paint brush while avoiding obstacles. Kinesthetic demonstrations were performed using one of the 7-DOF WAM arms of the HERB robot [67] built by Intel and pictured in Figure 7.6. Participants were asked to move the paint brush from a fixed start to a fixed goal, each about 5 mm in diameter, to provide at least three examples of each strategy (if they chose to demonstrate multiple strategies), and to provide at least six examples overall. The demonstration trajectories were processed by the learner, and detected bifurcations were presented to the participants, who were asked to indicate which strategy the robot should prefer. Finally, the robot created and performed a new plan. Participants were then asked to fill out a brief questionnaire about their experience.

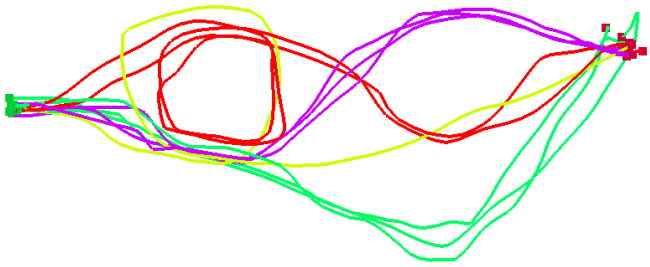


Figure 7.7: A set of demonstrations from a single teacher. Distinct strategies are shown in different colors. Our system detected two bifurcations, but did not detect the singleton strategy shown in yellow.

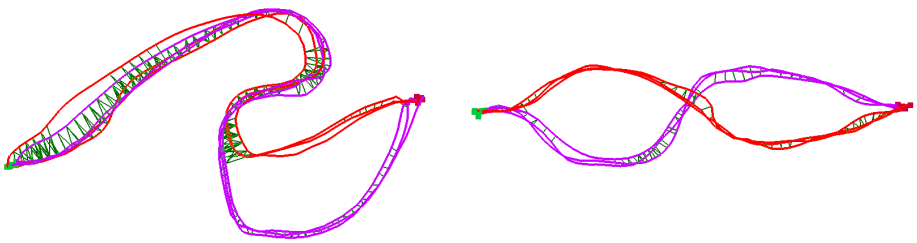


Figure 7.8: Some typical teacher demonstrations. The teacher on the left had specific robot manipulator experience. The teacher on the right reported no previous experience with robots.

Eighteen participants were recruited via word-of-mouth from Carnegie Mellon University and the surrounding community. They received no compensation for their participation. Participants were asked about their previous experience with robots, and were categorized based on

Ease of Programming	Chronbach's alpha = 0.7021
Training the robot was easy.	
I felt in control of the robot during training.	
I feel comfortable working with the robot.	
Quality of Plans	Chronbach's alpha = 0.7586
The robot's final plans were predictable.	
The robot's final plans were safe.	
The robot's final plans were aggressive.*	
The robot's final plans were surprising.*	
Training Questions	Chronbach's alpha = 0.7280
The questions asked by the robot were predictable.	
The questions asked by the robot were important to its training.	
The questions asked by the robot were surprising.*	
Effectiveness of Learning	Chronbach's alpha = 0.9216
The robot's final plans were natural.	
The robot's final plans were similar to mine.	
The robot's learned from my demonstrations.	

Table 7.1: Questions from the user trial. All questions were asked on a 5 point scale ranging from "Strongly Agree" to "Strongly Disagree". For analysis, the scales for questions marked with a * were reversed.

their response into groups with **general** robotics experience (four participants), **specific** experience with robot manipulators (three participants), or **none** (eleven participants). Almost all participants provided three demonstrations of each of two strategies, so a single bifurcation was detected and presented to the teacher. One participant (in the specific experience category) demonstrated a single strategy, and one participant (with general experience) provided nine examples in four different strategies. This set of trajectories is pictured in Figure 7.7. As requested, two of the strategies contain three examples each, but the remaining three trajectories demonstrate two strategies. Our system detected two bifurcations in this cases, but was unable to distinguish the singleton strategy. Figure 7.8 illustrates some more typical teacher demonstrations.

Survey questions are listed in Figure 7.3 and grouped into categories that measure similar qualities of the teacher's experience and percep-

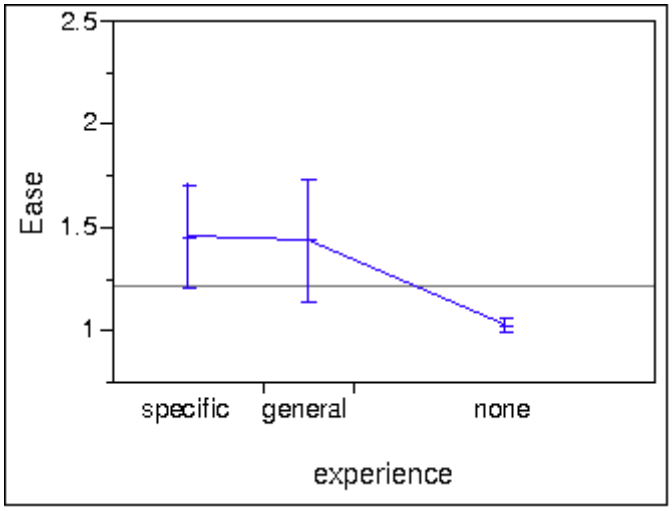


Figure 7.9: Results for the **Ease of Programming** scale. The overall mean response is 1.222.

tions. The "Chronback's alpha" listed for each scale is a measure of the internal consistency of responses within a category. A measure of 0.70 or higher is generally taken to indicate that the questions measure the same variable. For each scale, we compute the mean of responses in each experience category, and perform an analysis of variance (ANOVA) to determine the significance of the differences.

The "Ease of Programming" scale (Figure 7.9) exhibits the most significant difference between experience levels ($F = 3.24, p = 0.068$). Although the overall average of 1.222 indicates that users generally agreed with the questions in this category, there was some variance in responses from experienced users. Inexperienced users were nearly unanimous in their strong agreement with all of these questions.

The next category of questions asks for the teacher's evaluation of the "Quality of Plans" (Figure 7.10) in terms of safety and predictability. The difference between experience levels on this scale falls short of statistical significance ($F = 2.577, p = 0.110$), but the means are more widely distributed. The users with specific experience were nearly ambivalent, while the other groups expressed more confidence in the plans created.

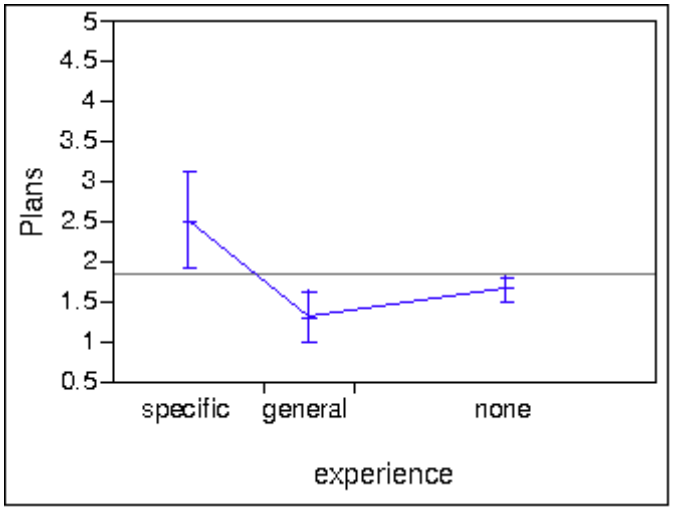


Figure 7.10: Results for the **Quality of Plans** scale. The overall mean response is 1.861.

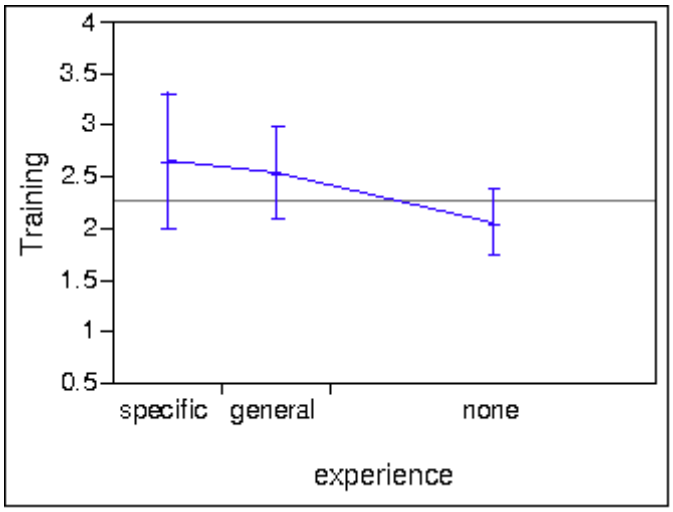


Figure 7.11: Results for the **Training Questions** scale. The overall mean response is 2.289.

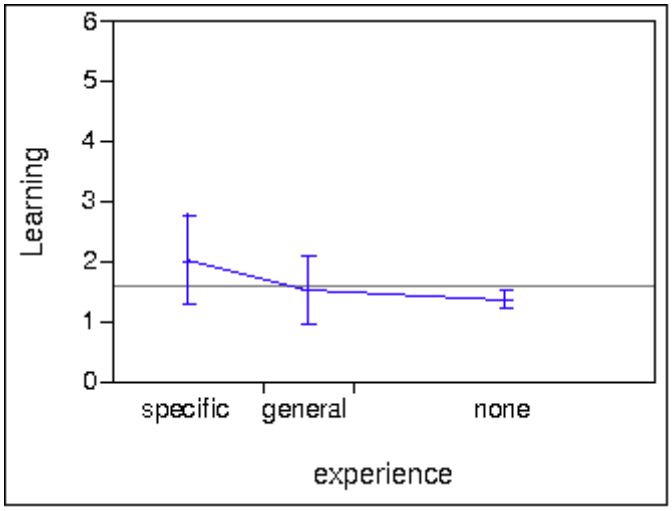


Figure 7.12: Results for the **Effectiveness of Learning** scale. The overall mean response is 1.611.

Results in the other categories were not statistically significant ($p > 0.1$). The "Training Questions" scale (Figure 7.11) refers to the robot's demonstration of the teacher's bifurcating strategies and the request (spoken by the experimenter) for a choice of strategy to be used in the robot's own plan. The responses are slightly more positive than neutral, but display large variance. This presents a potential problem for active learning. Although it is not strictly necessary for the teachers to understand the significance of this feedback cycle, the questions asked by the robot are essential for resolving the ambiguity presented by diverging demonstrations. Since the learner is capable of planning when it asks its questions (indeed, the questions involve the execution of partial plans), a teacher who thinks the questions are unimportant may skip this phase of training. The final scale, "Effectiveness of Learning" (Figure 7.12) also shows no statistically significant difference between the groups, but the overall mean of 1.611 is further from neutral. This indicates a general agreement that the robot's final plan followed the strategy that the teacher chose to convey.

7.4 Summary

The active learning approach described in this chapter allows a robot learner to detect and resolve ambiguous or conflicting advice that it receives during initial training. The detection algorithm, based on a graph cut procedure, searches for areas of disagreement among demonstration trajectories that were previously following similar strategies. When their strategies diverge, the lack of neighbor links between the strategies may indicate the presence of a physical obstacle or other constraints, or it may simply result from inconsistent and widely varying examples.

Since the robot learner does not possess enough information to determine either the cause or the solution to the inconsistent strategies in demonstration trajectories, it requests additional help from the teacher. As argued earlier, there is no single path statistic or objective criteria for automatically choosing one branch over another in all tasks and scenarios. Instead, by demonstrating the options that the learner has distilled from example trajectories, the teacher is given an opportunity to evaluate its knowledge. Additional examples may be provided to improve the learner's model of the task. The demonstration of plan options provides an intuitive means to solicit additional information from the teacher. Teachers are able to see the execution of proposed planning strategies in situ, which facilitates their decision regarding which strategy the robot learner should adopt for future plans. Teacher feedback then allows the robot to create safe, natural plans that exhibit the desired strategy.

User trials showed that people with all levels of experience with robots were able to easily and effectively teach a robot manipulator to perform a motion task through kinesthetic demonstrations. Although experienced robot programmers were generally less enthusiastic about the ease of Programming by Demonstration and the quality of the plans generated, all users considered the robot's final plans to be representative of the strategies they had demonstrated.

Chapter 8
Conclusions

> *Don't you see, when the imitator is perfect, so must be the imitation, and the semblance becomes the truth, the pretense a reality!*
>
> — Klapaucius, *The Cyberiad*, 1965

The thesis of this work is that a Programming by Demonstration approach to planning is able to construct safe, efficient, natural motion plans for robotic manipulators from imperfect demonstrations and advice requested when demonstrations are found to conflict. To support this claim, we have developed algorithms for organizing demonstration trajectories in a neighbor graph, planning new trajectories using this graph, and detecting and resolving conflicting information in demonstration trajectories through interaction with the teacher. This PbD system was implemented and tested in two experimental domains through user trials with novice and expert robot users.

Our approach to Programming by Demonstration begins with the construction of a neighbor graph that properly relates common portions of a collection of demonstration trajectories. Our algorithm builds upon the approach of ST-Isomap's neighbor finding to more reliably avoid erroneous neighbor links such as short-circuits between distinct portions of the task. We have formulated a series of heuristics to elide neighbor links in inconsistent areas of the trajectories that are likely to represent errors in the demonstrations. These heuristics were derived from observation of common errors in previous approaches to neighbor graphs for time-series data. Although this neighbor graph was originally intended for planning in a reduced-dimensionality space, we have demonstrated

inherent obstacles that confound this approach. Instead, the sensitivity to spurious neighbor links and the non-linearity of the embeddings led us to develop the alternate approach of planning directly over the neighbor graph.

The planner contributed in this thesis creates new trajectories using the neighbor graph in the original high-dimensional space. The correspondences between common portions of example trajectories – and the lack of correspondences in areas of deviation – are used to safely and smoothly interpolate between examples to create new trajectories. We have demonstrated this planner in various domains through experiments and user trials, and shown that inconsistencies and deviations in the demonstrations are avoided in the resulting plans, which are known to be safe in static environments. Trajectories generated by the system were shown to be 10.5% shorter and 33.9% smoother (in terms of curvature) than demonstration trajectories in experimental domains.

Finally, we have developed an algorithm for detecting conflicting information in demonstrations and formulated a user interaction strategy for resolving the conflicts. Our learner locates bifurcations in the neighbor graph using a normalized graph cuts algorithm and queries the teacher for the strategy it should follow when creating new plans by demonstrating the distinction between the branches of the bifurcation. When the teacher indicates a preference for one strategy over another, the learner respects this choice for future plans. If the teacher indicates no preference, the learner is free to plan using either strategy. Finally, the teacher may assert that there should be no distinction between the branches. In this case, the learner will begin executing a plan that approaches the bifurcation, then ask the teacher to continue, thus filling in the gap in the learner's knowledge. We conducted user trials with expert and novice robot users demonstrating the efficacy of this approach. Even participants with no prior experience working with robots were able to train a robot to perform a motion task and resolve the learner's question regarding a bifurcation in their strategies. Users of all skill levels found the system easy to use, and recognized the effectiveness of the system in conveying their motion strategies to the robot.

8.1 Future Work

Although the approach developed in this thesis has been successfully demonstrated in multiple domains, additional areas of research remain unexplored. Our work to date suggests possibilities for improvement in the computational complexity of our algorithms, and additional mechanisms for incorporating feedback from the teacher to learn better plans. We also envision additional applications for this work beyond teaching a skill to a single robot, and extensions to support lifting some of our assumptions and limitations on the applicability of this approach.

The most computationally demanding portion of the system described in this thesis is the assurance of safety for generated plans. This requires the construction of a high-resolution occupancy grid in the robot's workspace and calculation of the Signed Distance Field (SDF). Simplifying assumptions such as the assertion that only the final link of the robot's kinematic chain needs to be checked for collisions certainly help reduce the size of the problem, but the construction of the occupancy grid for a set of nominally similar demonstration trajectories entails many redundant operations. A more efficient algorithm, such as approaches that analytically compute the swept volume of a robot's geometric primitives through a trajectory [80], would speed the process.

The planning approach presented in this work creates a plan for execution in a static environment. In fact, the lack of ability to deal with dynamic obstacles is often a perception problem, as well. In cluttered, unstructured environments such as automobile assembly lines, it is usually difficult to sense the entire workspace of the robot and detect dynamic obstacles. If this perception problem were solved, though, our planner could be improved to dynamically replan trajectories when obstacles are detected. In particular, bifurcating strategies could be useful: when obstacles are detected in one branch, the planner could use the other strategy.

Our planning approach is also purely kinematic. It ignores the speed of demonstrated trajectories, and does not consider dynamic constraints. In some cases, simply interpolating demonstrated velocities during the action selection step of planning may produce plans that respect the

same dynamic constraints demonstrated by the teacher. However, to guarantee that these constraints are respected, an explicit check would be required after the refinement step, similar to the current SDF safety check.

Additionally, we could relax the constraint imposed by our definition of the start region, namely, that plans may safely start from any interpolation of initial demonstration points. Start regions could be explicitly delineated by the teacher, determined by the SDF, or, more simply, we could detect multiple disjoint clusters of initial demonstration points. For example, a redundant manipulator may start a task in distinct configurations that produce the same workspace pose. If these clusters of initial points were detected before building the neighbor graph, multiple start regions could be defined, and new plans could be started from the most appropriate region.

Interpolation is usually, but not always, a safe and desirable method for generalizing trajectories with non-geometric constraints. For example, random motions may be necessary for some kinds of manipulation tasks, resulting in small, uncorrelated motions in demonstrations. Our algorithm will discard these uncorrelated motions as noise, in favor of planning a smooth trajectory for that task. To correctly plan in these situations, it may be necessary to explicitly detect areas of uncorrelated noise and preserve it.

Similarly, there may be other objective functions employed by teachers in manipulation tasks that our planner should detect. Our current approach produces shorter, smoother plans as an effect of its interpolation strategy. Although we have not formulated the precise objective function being optimized by our planner, it would be useful to do so, and to provide a means to alter this objective function if we can determine the criteria that a human teacher is seeking to optimize in demonstrations.

Other work in Programming by Demonstration has explored alternate methods of active learning for soliciting and incorporating teacher feedback. For example, *Binary Critiquing* and *Advice Operators* [2] are two mechanisms for providing quality estimates and corrections to trajectories – both teacher demonstrations and plans generated by the learner. Binary Critiquing allows a teacher to identify areas of poor performance

in motion trajectories. Advice Operators provide additional information by indicating how trajectories (or portions of trajectories) could be improved. These concepts could be incorporated in our framework, possibly by providing scores or weights to elements of the neighbor graph. The refinement stage of the planner could use these scores to adjust the generation of new trajectories.

We also expect the improved neighbor-finding approach presented here to be useful for other applications in which motion trajectories are compared. In addition to Programming by Demonstration, this technique may prove useful for activity recognition, robot fault detection, and other applications in which time-series trajectories are compared. Activity recognition compares a given motion trajectory to collections of examples representing distinct activities. Our neighbor-finding technique is likely to construct a much denser neighbor graph between a query trajectory and additional demonstrations that represent the same activity. Similarly, a robot that frequently repeats a similar task or set of tasks is likely could compare a new trajectory to a database of prior executions. If the new trajectory is sufficiently dissimilar to normal executions, the robot might request operator assistance.

Finally, we may be able to distinguish between novice and expert demonstrations, so we can determine when a human teacher has learned a particular skill. In Chapter 5, we found that the demonstrations provided by inexperienced robot operators were typically more widely separated and contained variance not easily accounted for through dimensionality reduction. If this observation holds true even when a robot is not involved, such as using motion capture of human movements, we may be able to detect the decreased variance between examples when a person becomes proficient at a particular activity. This could be useful, for instance, when astronauts train to perform an Extra-Vehicular Activity (EVA). Due to the danger and difficulty of working in spacesuit outside of a spacecraft, astronauts practice these activities extensively in neutral-buoyancy underwater simulations. A quantitative assessment of the astronaut's proficiency in performing an EVA could be a useful tool for determining when the necessary skills have been mastered.

8.2 Summary

We have presented a Programming by Demonstration algorithm for teaching robots to perform repeatable motion tasks safely, efficiently, and smoothly. We have developed heuristics for relating imperfect demonstrations in a neighbor graph, and an algorithm to create new plans using this graph. We have also developed an algorithm for detecting bifurcations in the teacher's plans, demonstrating the distinction between the strategies, and eliciting advice for creating future plans. User trials have demonstrated the capabilities of this system for novice and expert robot users. While additional avenues of research remain, the contributions of this thesis provide an effectual means for programming a robot by demonstration.

References

Can't we just talk to the humans? A little understanding could make things better. Can't we talk to the humans and work together now?

— "The Humans are Dead", *The Distant Future*, Flight of the Conchords, 2007

[1] J. Aleotti, S. Caselli, and G. Maccherozzi. Trajectory reconstruction with nurbs curves for robot programming by demonstration. In *Computational Intelligence in Robotics and Automation, 2005. CIRA 2005. Proceedings. 2005 IEEE International Symposium on*, pages 73–78, 2005. 2.2, 2.3

[2] Brenna Argall. *Learning Mobile Robot Motion Control from Demonstration and Corrective Feedback*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, March 2009. 8.1

[3] Brenna Argall, Brett Browning, and Manuela Veloso. Learning by demonstration with critique from a human teacher. In *ACM/IEEE international conference on Human-robot interaction*, pages 57–64, Arlington, Virginia, USA, 2007. ACM Press. 2.3

[4] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robot. Auton. Syst.*, 57:469–483, May 2009. 2

[5] H. Asada. Teaching and learning of compliance using neural nets: representation and generation of nonlinear compliance. In *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 1237–1244 vol.2, 1990. 2.1

[6] H. Asada and H. Izumi. Automatic program generation from teaching data for the hybrid control of robots. *Robotics and Automation, IEEE Transactions on*, 5:166–173, 1989. 2.1

[7] Atkeson, Moore, and Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11:11–73, February 1997. 2.2

[8] James (Drew) Bagnell and Jeff Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*. IEEE, May 2001. 2.2

[9] D. C. Bentivegna. *Learning from Observation Using Primitives*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2004. 2.3

[10] D.C. Bentivegna and C.G. Atkeson. Learning from observation using primitives. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 1988–1993 vol.2, 2001. 2.2, 2.4

[11] Marshall Bern and David Eppstein. Mesh generation and optimal triangulation. *Computing in Euclidean Geometry*, 1:23–90, 1992. 3.3

[12] A. Billard, S. Calinon, R. Dillmann, and S. Schaal. Robot programming by demonstration. In B. Siciliano and O. Khatib, editors, *Handbook of Robotics*, pages 1371–1394. Springer, Secaucus, NJ, USA, 2008. 2

[13] Christina Louise Campbell, Richard Alan Peters, Robert E. Brodenheimer, William J. Bluethmann, Eric Huber, and Robert O. Ambrose. Superpositioning of behaviors learned through teleoperation. *IEEE Transactions on Robotics*, 22(1):79–91, February 2006. 2.2, 2.5

[14] J. Chen and A. Zelinsky. Programing by demonstration: Coping with suboptimal teaching actions. *Int. Journal of Robotics Research*, 22(5):299–319, 2003. 2.1

[15] Sonia Chernova and Manuela Veloso. Tree-based policy learning in continuous domains through teaching by demonstration. In Gal Kaminka, David Pynadath, and Christopher Geib, editors, *Modeling Others from Observations: Papers from the AAAI Workshop*, pages 24–31. American Association for Artificial Intelligence, 2006. 2.3, 2.4

[16] Adam Coates, Pieter Abbeel, and Andrew Y. Ng. Apprenticeship learning for helicopter control. *Communications of the ACM*, page 2009. 2.2

[17] D. A Cohn, Z. Ghahramani, and M. I Jordan. Active learning with statistical models. *Journal of Artificial Intelligence Research*, 4:129–145, feb 1996. 2.3

[18] Kerstin Dautenhahn and Chrystopher L. Nehaniv. *Imitation in Animals and Artifacts*. MIT Press, 2002. 2.5, 3.2

[19] N. Delson and H. West. Robot programming by human demonstration: the use of human inconsistency in improving 3d robot trajectories. In *Intelligent Robots and Systems '94. 'Advanced Robotic Systems and the Real World', IROS '94. Proceedings of the IEEE/RSJ/GI International Conference on*, volume 2, pages 1248–1255 vol.2, 1994. 2.3

[20] N. Delson and H. West. Robot programming by human demonstration: the use of human variation in identifying obstacle free trajectories. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 564–571 vol.1, 1994. 2.3, 2.6

[21] N. Delson and H. West. Robot programming by human demonstration: adaptation and inconsistency in constrained motion. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 30–36 vol.1, 1996. 2.3, 5.3

[22] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 3.1

[23] E. Drumwright, O.C. Jenkins, and M.J. Matarić. Exemplar-based primitives for humanoid movement classification and control. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 1, pages 140–145 Vol.1, 2004. 2.2, 2.5

[24] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973. 7.1

[25] Imola Fodor. A survey of dimension reduction techniques. Technical report, U.S. Department of Energy, 2002. 2.5

[26] H. Friedrich, J. Holle, and R. Dillmann. Interactive generation of flexible robot programs. In *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, volume 1, pages 538–543 vol.1, 1998. 2.3

[27] Daniel H Grollman and Odest Chadwicke Jenkins. Dogged learning for robots. In *2007 IEEE International Conference on Robotics and Automation (ICRA)*, 2007. 2.3

[28] M Hein and M Maier. Manifold denoising as preprocessing for finding natural representations of data. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1646–1649, Menlo Park, CA, July 2007. AAAI Press. 5

[29] M. Hersch, F. Guenter, S. Calinon, and A. Billard. Dynamical system modulation for robot learning via kinesthetic demonstrations. *IEEE Transactions on Robotics*, 2008. 2.2

[30] G. Hovland, P. Sikka, and B. McCarragher. Skill acquisition from human demonstration using a hidden markov model. In *IEEE International Conference on Robotics and Automation*, pages 2706–2711, Minneapolis, MN, 1996. 2.4

[31] Soshi Iba. *Interactive Multi-Modal Robot Programming*. PhD thesis, Robotics Institute, Carnegie Mellon University, May 2004. 2.4

[32] A.J. Ijspeert, J. Nakanishi, and S. Schaal. Trajectory formation for imitation with nonlinear dynamical systems. In *Intelligent*

Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on, volume 2, pages 752–757 vol.2, 2001. 2.2

[33] Odest Chadwicke Jenkins and Maja J. Matarić. Performance-derived behavior vocabularies: Data-driven acqusition of skills from motion. *International Journal of Humanoid Robotics*, 1:237–288, June 2004. 2.5

[34] Odest Chadwicke Jenkins and Maja J. Matarić. A spatio-temporal extension to isomap nonlinear dimension reduction. In *The Twenty-first International Conference on Machine Learning*, page 56, Banff, Alberta, Canada, 2004. ACM Press. 1.2, 2.5, 3.2, 5

[35] M. Kaiser and R. Dillmann. Building elementary robot skills from human demonstration. In *Proceedings of 1996 IEEE International Conference on Robotics and Automation*, volume 3, pages 2700 – 2705, apr 1996. 2.2

[36] Sing Bing Kang. *Robot Instruction by Human Demonstration*. PhD thesis, Robotics Institute, Carnegie Mellon University, December 1994. 2.3

[37] L.E. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proceedings of the International Conference on Robotics and Automation*, pages 2138–2145, San Diego, CA, 1994. 2.1

[38] Klaas Klasing, Dirk Wollherr, and Martin Buss. Joint dominance coefficients: A sensitivity-based measure for ranking robotic degrees of freedom. In Torsten KrÃűger and Friedrich M. Wahl, editors, *Advances in Robotics Research*, pages 1–10. Springer Berlin Heidelberg, 2009. 5.1

[39] V. Klema and A. Laub. The singular value decomposition: Its computation and some applications. *Automatic Control, IEEE Transactions on*, 25(2):164 – 176, apr 1980. 3.1

[40] Y. Kuniyoshi, M. Inaba, and H. Inoue. Learning by watching: Extracting reusable task knowledge from visual observation of human

performance. *Transactions on Robots and Automation*, 10:799–822, 1994. 2.1

[41] S. M. Lavalle and J. J. Kuffner. Randomized kinodynamic planning. *International Journal of Robotics Research*, 20(5):378–400, May 2001. 2.1

[42] C.H. Lee. A phase space spline smoother for fitting trajectories. *Systems, Man and Cybernetics, Part B, IEEE Transactions on*, 34:346–356, 2004. 2.2

[43] A. Lockerd and C. Breazeal. Tutelage and socially guided robot learning. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 4, pages 3475–3480 vol.4, 2004. 2.3

[44] Pattie Maes and Rodney A. Brooks. Learning to coordinate behaviors. In *National Conference on Artificial Intelligence*, pages 796–802, 1990. 2.2

[45] H Mayer, I Nagy, A Knoll, E.U. Braun, R Lange, and R Bauernschmitt. Adaptive control for human-robot skilltransfer: Trajectory planning based on fluid dynamics. In *ICRA*, Rome, Italy, 2007. 2.3

[46] M.Berlin, J. Gray, A. L. Thomaz, and C. Breazeal. Perspective taking: An organizing principle for learning in human-robot interaction. In *Proceedings of the 21st National Conference on Artificial Intelligence*, 2006. 2.4

[47] Andrew Moore, Jeff Schneider, Justin Boyan, and M.S. Lee. Q2: memory-based active learning for optimizing noisy continuous functions. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 4, pages 4095 – 4102, April 2000. 2.4

[48] J. R. Munkres. *Topology*. Prentice Hall, Upper Saddle River, NJ, 2000. 5.3

[49] Koichi Ogawara, Jun Takamatsu, Soshi Iba, Tomikazu Tanuki, Hiroshi Kimura, and Katsushi Ikeuchi. Acquiring hand-action models

in task and behavior levels by a learning robot through observing human demonstrations. In *The IEEE-RAS International Conference on Humanoid Robots*, September 2000. 2.4

[50] R.A. Peters and O.C. Jenkins. Uncovering manifold structures in robonaut's sensory-data state space. In *Humanoid Robots, 2005 5th IEEE-RAS International Conference on*, pages 369–374, 2005. 2.5

[51] Dean A. Pomerleau. Efficient trairing of artificial neural networks for autonomous navigation. *Neural Computation*, 3:88–97, 1991. 2.2

[52] Polly K. Pook and Dana H. Ballard. Recognizing teleoperated manipulations. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 578–585, 1993. 2.2

[53] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992. 2.4

[54] Nathan Ratliff, David Bradley, James Bagnell, and Joel Chestnutt. Boosting structured prediction for imitation learning. In *Advances in Neural Information Processing Systems 19*, Cambridge, MA, 2007. MIT Press. 2.2

[55] Jens Rittscher, Andrew Blake, Anthony Hoogs, and Gees Stein. Mathematical modelling of animate and intentional motion. *Philosophical Transactions: Biological Sciences*, 358:475–490, March 2003. 2.5

[56] Charles Rose, Michael F. Cohen, and Bobby Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Comput. Graph. Appl*, 18:32–40, 1998. 2.5

[57] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26:43–49, 1978. 5

[58] Marcos Salganicoff, Lyle H. Ungar, and Ruzena Bajcsy. Active learning for vision-based robot grasping. *Mach. Learn.*, 23(2-3):251–278, 1996. 2.4

[59] Stefan Schaal, A.J. Ijspeert, and Aude Billard. Computational approaches to motor learning by imitation. *Philosophical Transactions of the Royal Society*, 358:537–547, March 2003. 2.5

[60] B. Sellner, F. Heger, L. M. Hiatt, R. Simmons, and S. Singh. Coordinated multi-agent teams and sliding autonomy for large-scale assembly. *Proceedings of the IEEE*, 94(7), July 2006. 1

[61] Brennan Sellner, Frederik W. Heger, Laura M. Hiatt, Nik A. Melchior, Stephen Roderick, Dave Akin, Reid Simmons, and Sanjiv Singh. Overcoming sensor noise for low-tolerance autonomous assembly. In *Proceedings of the IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems*, Nice, France, September 22-26 2008. 1

[62] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999. 6.3

[63] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, Washington, DC, USA, 1997. IEEE Computer Society. 7.1

[64] Marjorie Skubic, David Noelle, Mitch Wilkes, Kazuhiko Kawamura, and James M. Keller. A biologically inspired adaptive working memory for robots. In *AAAI Fall Symp., Workshop on the Intersection of Cognitive Science and Robotics: From Interfaces to Intelligence*, October 2004. 2.4

[65] William D. Smart and L. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceeding of the International Conference on Machine Learning*, 2000. 2.2

[66] P. H. A. Sneath and R. R. Sokal. *Numerical Taxonomy*. Freeman, London, UK, 1973. 7.1

[67] S. Srinivasa, D. Ferguson, C. Helfrich, D. Berenson, A. Collet, R. Diankov, G. Gallagher, G. Hollinger, J. Kuffner, and M. VandeWeghe. Herb: A home exploring robotic butler. 2009. doi:10.1007/s10514-009-9160-9. 7.3

[68] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 4, pages 3310 – 3317, May 1994. 2.1

[69] M. Stolle and C.G. Atkeson. Policies based on trajectory libraries. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 3344–3349, 2006. 2.2, 2.3

[70] N. Tatematsu and K. Ohnishi. Tracking motion of mobile robot for moving target using nurbs curve. In *Industrial Technology, 2003 IEEE International Conference on*, volume 1, pages 245–249 Vol.1, 2003. 2.2

[71] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290:2319–2323, December 2000. 2.5, 3.1

[72] A.L. Thomaz, G. Hoffman, and C. Breazeal. Reinforcement learning with human teachers: Understanding how people want to teach robots. *Robot and Human Interactive Communication, 2006. ROMAN 2006. The 15th IEEE International Symposium on*, pages 352–357, Sept. 2006. 2.3

[73] J.G. Trafton, N.L. Cassimatis, M.D. Bugajska, D.P. Brock, F.E. Mintz, and A.C. Schultz. Enabling effective human-robot interaction using perspective-taking in robots. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 35:460– 470, 2005. 2.4

[74] A. Tsoli and O. C. Jenkins. Neighborhood denoising for learning high-dimensional grasping manifolds. In *International Conference on Intelligent Robots and Systems (IROS 2008)*, pages 3680–3685, Nice, France, Sep 2008. 3.2, 3.4

[75] Aggeliki Tsoli and Odest Chadwicke Jenkins. 2d subspaces for user-driven robot grasping. In *Robotics, Science and Systems Conference: Workshop on Robot Manipulation*, 2007. 5

[76] A. Ude, C.G. Atkeson, and M. Riley. Planning of joint trajectories for humanoid robots using b-spline wavelets. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 3, pages 2223–2228 vol.3, 2000. 2.2

[77] Michael van Lent and John E. Laird. Learning procedural knowledge through observation. In *The 1st international conference on Knowledge capture*, pages 179–186, Victoria, British Columbia, Canada, 2001. ACM Press. 2.4

[78] Y. Wang, M. Huber, V.N. Papudesi, and D.J. Cook. User-guided reinforcement learning of robot assistive tasks for an intelligent environment. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 424–429, October 2003. 2.2, 2.3

[79] C. Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge,England, 1989. 2.3

[80] John D. Weld and Ming C. Leu. Geometric representation of swept volumes with application to polyhedral objects. *The International Journal of Robotics Research*, 9(5):105–117, 1990. 8.1

[81] Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. In *Exploring artificial intelligence in the new millennium*, pages 239–269. Morgan Kaufmann Publishers Inc., 2003. 5

[82] Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, Bernhard SchÃűlkopf, and Bernhard Sch Olkopf. Learning with local and global consistency. *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 16*, 16:321—328, 2003. 5

[83] R. Zollner, O. Rogalla, R. Dillmann, and M. Zollner. Understanding users intention: programming fine manipulation tasks by demonstration. In *Intelligent Robots and System, 2002. IEEE/RSJ International Conference on*, volume 2, pages 1114–1119 vol.2, 2002. 2.3