UNIVERSITY OF CALIFORNIA, SAN DIEGO

Interactive Motion Planning with Motion Capture Data

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Wan-Yen Lo

Committee in charge:

Professor Matthias Zwicker, Chair
Professor Henrik Wann Jensen, Co-Chair
Professor Samuel Buss
Professor Charles Elkan
Professor Victor Zordan

2012

The dissertation of Wan-Yen Lo is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California, San Diego

2012

DEDICATION

To Mom and Dad.

EPIGRAPH

Your task is not to foresee the future,
but to enable it.
—Antoine de Saint Exupéry

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Finally, it is time to write the Acknowledgments section. It has been a long journey and I am standing right in front of the destination. I could not have made such a long way without the support and help from so many people.

First of all, I would like to thank my family, especially my parents, to whom this dissertation is dedicated. They always give full support to my decisions, so that I can pursue my dreams wholeheartedly in the past few years. Without them, I would not be here today.

I would like to thank my advisor Prof. Matthias Zwicker. He gave me the opportunity to start this amazing journey, and provided me with constant support all along the way. I will definitely miss the time during which I could walk directly into his office and then spend hours discussing research. I really appreciate the guidance I received from him, and I believe it will continue to have a positive effect on my career path in the future. I would also like to thank Prof. Henrik Wann Jensen for many valuable discussions and inspirations. Finally, I would like to thank the other members of my dissertation committee, Samuel Buss, Charles Elkan, and Victor Zordan, for taking their time to read this dissertation and to come to my defense.

I would also like to express my gratitude to Jeroen van Baar from Disney Research Zurich. I had a great time working with him as a summer intern. Without his inspiration, I would not have started and published a stereoscopic project.

Many thanks go to all my colleagues at the UCSD graphics lab, in particular: Will Chang, Craig Donner, Toshiya Hachisuka, Wojciech Jarosz, Neel Joshi, Arash Keshmirian, Krystle de Mesa, Iman Mostafavi, and Iman Sadeghi. I enjoyed the countless discussions we had, either about research or life. I appreciate the fact that they made the working environment stimulating and cheerful, and that they always gave me a hand whenever I was stuck with research problems. Further, special thanks go to Kuei-Chun Hsu and Yen-Lin Lee for volunteering to be my MOCAP actors.

My thanks also go to my colleagues in the computer graphics group at University of Bern: Daljit Singh Dhillon, Daniel Donatsch, Claude Knaus, Fabrice Rousselle, and Sonja Schär. They have always created a very sociable and friendly atmosphere.

In the end, I would like to thank Julie Conner from UCSD and Dragana Esser from University of Bern for helping me with the administrative work. My involving with three countries, Taiwan, USA, and Switzerland, have caused quite some trouble for them, but they have always been patient and helpful.

Parts of this dissertation are based on papers co-authored with my collaborators:

- Chapter 3 is based on "Bidirectional Search for Interactive Motion Synthesis", Wan-Yen Lo and Matthias Zwicker, *Computer Graphics Forum (Proceedings of Eurographics EG'10)*, 2010.
- Chapter 5 is based on "Real-Time Planning for Parameterized Human Motion", Wan-Yen Lo and Matthias Zwicker, *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2008.
- Chapter 6 is based on "Learning Motion Controllers with Adaptive Depth Perception", Wan-Yen Lo, Claude Knaus, and Matthias Zwicker, currently under review.

I was the primary researcher and author of these papers.

VITA

1982	Born, Taiwan
2004	B. S., National Taiwan University, Taiwan
2006	M. S., National Taiwan University, Taiwan
2009-2011	Graduate Research Assistant, University of Bern, Switzerland
2012	Ph. D., University of California, San Diego, USA

PUBLICATIONS

Wan-Yen Lo, Jeroen Van Baar, Claude Knaus, Matthias Zwicker, and Markus Gross, "Stereoscopic 3D Copy & Paste", *ACM SIGGRAPH Asia*, 2010.

Wan-Yen Lo and Matthias Zwicker, "Bidirectional Search for Interactive Motion Synthesis", *Computer Graphics Forum (Proceedings of Eurographics EG'10)*, 2010.

Wan-Yen Lo and Matthias Zwicker, "Real-Time Planning for Parameterized Human Motion", *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2008.

Ying-Ruei Chen, Wan-Yen Lo, Yu-Pao Tsai, and Yi-Ping Hung, "Generation of Binocular Object Movies from Monocular Object Movies", *SPIE Conference on Stereoscopic Displays and Virtual Reality Systems XIV*, 2007.

Pang-Hung Huang, Yu-Pao Tsai, Wan-Yen Lo, Sheng-Wen Shih, and Yi-Ping Hung, "Calibration of Motorized Object Rig and Its Applications", *Journal of Information Science and Engineering*, 2007.

Pang-Hung Huang, Yu-Pao Tsai, Wan-Yen Lo, Sheng-Wen Shih, Chu-Song Chen, Yi-Ping Hung, "A Method for Calibrating Motorized Object Rig", *Asian Conference on Computer Vision*, 2006.

Wan-Yen Lo, Yu-Pao Tsai, Chien-Wei Chen, and Yi-Ping Hung, "Stereoscopic Kiosk for Virtual Museum", *International Computer Symposium*, 2004.

ABSTRACT OF THE DISSERTATION

Interactive Motion Planning with Motion Capture Data

by

Wan-Yen Lo
Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Matthias Zwicker, Chair
Professor Henrik Wann Jensen, Co-Chair

Realistic character motion is an important component in media production, such as movies and video games. More lifelike characters enhance storytelling and immersive experience. To date, the most common approach to offer high degree of realism is based on large databases of motion capture data. The motion capture process, however, is expensive and time-consuming, while only a limited number and range of motions can be captured at a time. As a consequence, realistic motion synthesis has become a core research topic in computer animation.

Many of the most successful techniques are based on fragmenting and recombining motion capture data. The connectivity among the motion fragments is encoded with a graph structure, and novel motions can be generated with graph traversals. In addition, most systems allow a user to provide a number of constraints to specify the desired motion. By formulating the constraints as a cost function, motion synthesis is cast as a graph search problem, and the optimally-synthesized motion corresponds to the path through the graph that minimizes the total cost. The search complexity for an optimal or near-optimal solution, however, is exponential to the connectivity of the graph and the length of the desired motion sequence. Synthesizing optimal or near-optimal motions is thus challenging for interactive applications. In this dissertation, we explore the two most significant research directions toward near-optimal motion synthesis, including graph search and reinforcement learning, and present algorithms for interactive and real-time character animation.

This dissertation begins by reviewing previous work on searching motion graphs. In particular, A* search is optimally efficient and considered the state-of-the-art technique for optimal motion synthesis. However, applying A* search on motion graphs is challenging when interactive performance is demanded. To make A* search more applicable to interactive applications, we present a bidirectional search algorithm to improve the search efficiency while preserving the search quality. This can reduce the maximal search depth by almost a factor of two, leading to significant performance improvements. We further demonstrate its application to interactive motion synthesis using an intuitive sketching interface.

The second part of the dissertation consists of reinforcement learning frameworks for real-time character animation. The character controller makes near-optimal decisions in response to user input in real-time. The controller is constructed in a pre-process by exploring all possible situations. We introduce a tree-based regression algorithm, which is more efficient and robust than previous strategies for learning controllers. In addition, we extend the learning framework to include parameterized motions and interpolation for precise motion control. Finally, we show how to leverage character controllers by letting the character "see" the environment directly with depth perception. We derive a hierarchical state model and a regression algorithm to avoid the curse of dimensionality resulting from raw vision input. The controller can be generalized to allow a character to navigate or survive in environments containing arbitrarily shaped obstacles, which is hard to achieve with previous reinforcement learning frameworks.

Chapter 1

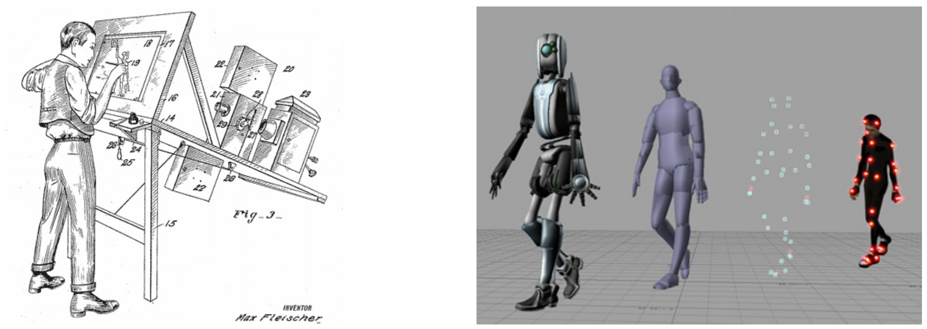
Introduction

“One sees clearly only with the heart. Anything essential is invisible to the eyes."

The Little Prince
Antoine de Saint Exupéry

Realistic character motion plays a vital part in media production, such as films and computer games. More lifelike characters make more believable special effects and enhance immersive experiences. However, creating realistic motions is a very challenging task. Since humans are adept at discerning the subtleties of common movements, even small unnaturalness might break the realism of the production. A few decades ago, rotoscoping was a popular way to animate lifelike characters, where the animators draw the motion frame by frame, by tracing over a live-action film, as illustrated in Figure 1.1a. In the last decade, thanks to the invention of motion capture (mocap) systems, motion details of a live performer can be accurately and continuously recorded. By placing a set of markers on a performer’s joints, the motion capture system can record the 3D positions of the markers for reconstructing the whole body motion, as shown in Figure 1.1b. Alternatively, some approaches build a physical model of the character and simulate the animation under the laws of physics. Although pure physically-based approaches can greatly reduce the costs of softwares, equipments, and personnel required to perform motion capture, the generated motions tend to appear robotic.

With motion capture data, we are able to replay the movement on a virtual character, and the resulting motion exhibits a high degree of realism. However, with limited time and resources, it is not practical to capture every possible body movement. Starting a capture process is also expensive and time-consuming. Even though more and more organized mocap databases are



(a) Rotoscoping [Max Fleischer] (b) Motion capture [Wikipedia]

Figure 1.1: Motion capture techniques. (a) Fleischer's rotoscope machine allows an artist to draw on the transparent panel, onto which the images of a film are projected. (b) Digital motion capture systems enable tracking the markers placed on an actor's body in 3D space. The recorded data can then be used to animate digital characters.

available online, obtaining motions for a particular application is still difficult, as not all desirable motions are accessible. How to make the best use of the existing data for synthesizing novel but realistic motion has thus become an important research topic.

One way that is commonly used in practice is to cut the capture motions into pieces, and to synthesize novel motion by reassembling motion fragments. Realism can be directly preserved in this way, as long as the motion pieces are well concatenated. Users can also have controls over the final motion, such as moving direction, or performing a specific action at a specific time. In the computer game industry, connections among motion pieces are manually constructed into *move trees*, which are elaborate state machines that determine the best motion based on the environment and user input. Although this approach is reactive, it involves a large amount of manual work to craft carefully the state machines and the rules specifying all potential state transitions. This becomes particularly cumbersome in modern games that use thousands of individual motion clips.

In recent years, several methods have been proposed to automatically construct a graph for representing smooth transitions among motion fragments. Most systems allow a user to provide a number of constraints to specify the desired motion. Such constraints are formulated as a cost function. Motion synthesis is thus cast as a search problem for a path through the motion graph that minimizes the total cost. The search complexity for an optimal or near-optimal solution, however, is exponential to the connectivity of the graph and the length of the desired motion sequence. Hence, there is often a tradeoff between performance and optimality, although both are critical for interactive applications. In interactive applications, user inputs arrive continuously and the environment changes dynamically, so in order to make the character responsive, time lag is not allowed. The quality of the solution is equally important, since the optimal path on a motion graph often leads to the most natural motion. Nevertheless, despite many advances in character animation, generating highly responsive and realistic motions in real-time remains a difficult problem.

In this dissertation, we explore the two most significant research directions toward near-optimal motion synthesis, and present algorithms for interactive and real-time character animation. We first explore techniques on searching with motion graphs, and present a bidirectional search algorithm to improve the search efficiency while preserving the search quality. This is orthogonal to many existing search algorithms and can reduce the maximal search depth by almost a factor of two, leading to significant performance improvements. We demonstrate its application to interactive motion synthesis using an intuitive sketching interface.

We also explore reinforcement learning frameworks for real-time character animation. We introduce a tree-based regression algorithm, which is more efficient and robust than previous strategies for learning character controllers. We also extend the existing frameworks to include parameterized motions and interpolation for precise motion control. Finally, we show how to leverage character controllers by letting the character "see" the environment directly with depth perception. We derive a hierarchical state model and a regression algorithm to avoid the curse of dimensionality resulting from raw vision input. The controller can be generalized to allow a character to navigate or survive in environments containing arbitrarily shaped obstacles, which is hard to achieve with previous reinforcement learning frameworks.

1.1 Related Work

In this section, we give an overview of related research work in character animation. We first summarize methods based on splitting and reassembling motion capture data, among which motion graphs and reinforcement learning are most related to our work, and will be further introduced in Chapter 2 and Chapter 4 respectively. Next we review methods for synthesizing novel poses that cannot be generated by reassembling input data. These methods include interpolation and dynamic approaches. We have used interpolation in our work (Chapter 3 and Chapter 5) for precise motion control, but the dynamic approaches are summarized here for completeness. In the end, we discuss varied motion control interfaces, and we will present our sketch interface in Chapter 3.

1.1.1 Reassembling Motion Fragments

Schödl et al. [69] develop the concept of *video textures* in image-based graphics, allowing a continuous, infinite video stream to be synthesized from a short video clip. They achieve this by analyzing the input video to identify appropriate transitions between two non-subsequent video frames. By rearranging the video frames with respect to the transition rules, novel but similar looking video of arbitrary length can be synthesized. Video textures subsequently inspired many approaches in computer animation that synthesize realistic motions by piecing together example motions from a database.

In computer animation, Pullen and Bregler [63] break the captured data into fragments and rearrange them to meet user-specified keyframes. Motion graphs and related approaches [31, 1, 38] encode how the motion fragments may be reassembled in different ways using a graph structure, where nodes correspond to poses from the database, and edges represent transitions between similar poses. These techniques generate new motions by building walks on the graph, transforming motion synthesis into a graph search problem. Kovar et al. [31] use depth-first search to obtain graph walks. They improve the efficiency of naive depth-first search using a branch-and-bound strategy and incremental search. Lee et al. [38] use greedy best-first search and traverse only a fixed number of frames to maintain a constant rate of motion. Arikan et al. [2] use dynamic programming to rearrange the motion fragments in a hierarchical way, and allow users to specify what should happen at what time. Lee et al. [41] use smaller environment-specific motion graphs as building blocks, with which the users can compose a large and complex virtual environment. The characters can navigate through and interact with the environment in real-time.

In the systems where the users can specify constraints on the synthesized motion, the results are evaluated with a cost function. The optimal solution corresponds to the fragment rearrangement that minimizes the total cost. However, the complexity of finding the optimal solution is proportional to the size of database. To speed up the computation, Lau and Kuffner [35] manually created a behavior finite-state machine, which defines the movement capabilities of a virtual character. They define the finite-state machine with only a very small number of nodes, so that the search space is highly reduced. They also show how pre-computation can be leveraged to increase runtime performance [36]. Zhao et al. [92] propose an automatic approach to select a good subset from the original database, in order to reduce the size and improve the performance of a motion graph.

More recently, reinforcement learning approaches have got considerable attention for real-time character animation. Reinforcement learning can be used in a pre-processing stage to create an optimized motion controller, which assembles a motion stream from motion fragments in real-time to minimize a long-term cost [39, 22, 81]. McCann and Pollard [50] integrate a model of user behavior into reinforcement learning, enabling highly responsive real-time character control. Lee et al. [43] present methods for constructing complex individual and connecting controllers over an automatically selected compact set of motion clips. Lee and Popović [42] present a method to determine the appropriate reward function in the reinforcement learning framework, so as to infer the behavior styles from a small set of examples. Wampler at el. [82] extend the reinforcement learning framework with game theory to generate controllers in two-player adversarial games. Levine et al. [46] combines reinforcement learning with heuristic search to enable space-time planning in a highly dynamic environment.

1.1.2 Synthesizing Novel Poses

Although the approaches that rely on reassembling motion fragments have emerged as primary sources of realistic character animation, their major limitation is the lack of continuous properties of motion. Since the motion fragments only constitute a discrete representation of all possible motions, it is difficult to gain precise character control, e.g. grasping an object at a specific location. Therefore, a number of methods have been proposed to utilize the captured data to synthesize novel but realistic motions.

Interpolation is a common approach in many areas, and can be used to create novel motions that have specific kinematic or physical attributes [86, 65]. Kovar and Gleicher [29, 30] proposed an automated method for identifying and registering logically similar motions. They also build a continuous parameterized motion space for similar motions that provide efficient control for interpolation. Mukai and Kuriyama [55] improve motion interpolation with the use of geostatistics, treating interpolation as statistical prediction of missing data in the parametric space. Safonova and Hodgins [67] analyze interpolated human motions for physical correctness and show that the interpolated results are close to the physically-correct motions. Cooper et al. [12] proposed active learning to adaptively sample the parametric space so that the space can be well sampled with a reduced number of clips. Researchers have also combined motion graphs with parametric synthesis to create parameterized motion graphs [71, 20, 68]. In addition, Zhao and Safonova use motion interpolation to increase the connectivity of motion graphs [93].

Another group of approaches integrates dynamics of human motion to compute joint angles and velocities to animate a character. Since physical correctness itself does not guarantee motion naturalness, researchers have proposed learning dynamic controllers automatically from captured motions [94, 75, 13, 54]. Statistical models can also be used to approximate human motion dynamics for estimating a character's pose at each possible state [19, 8, 83, 89]. Liu et al. [48] incorporate several biomechanical properties into the dynamic model, and introduce an algorithm for extracting physical parameters from motion capture data, in order to generate novel motions in the same style. Lau et al. [34] use a dynamic Bayesian network to synthesize natural spatial and temporal variations from a small set of captured data. Recently, Ye and

Liu [88] present an optimal feedback controller, with which a motion capture sequence can be adapted in real-time to changes of environment and physical perturbations.

1.1.3 Motion Control Interfaces

Intuitive user interfaces are essential for effective motion synthesis applications. With the increasing popularity in motion capture devices and the improvements in character animation research, today's applications often include characters with a rich set of behaviors. Accordingly, it becomes increasingly difficult to control varied motions with conventional interfaces, such as joysticks, game pads, mice, or keyboards, since the actions have many more degrees of freedom than can be specified directly. In this section, we review some research works that utilize non-conventional user interfaces to make more intuitive control.

Sketch-based approaches allow a user to generate a wider variety of motions [79, 60] or to edit existing motions [53] by drawing simple strokes. Thorne et al. [79] design a gesture vocabulary, where each gesture alphabet corresponds to a specific motion, such as jump, skate, or flip, and allow a user to specify a motion by combining gesture alphabets. The input sketch is automatically segmented, parsed, and converted into a sequence of motion. Instead of building a gesture vocabulary, Oshita [60] determines a motion from the semantics associated with the source and target of a stroke. Both methods are simple to use for novice users, but as the number of motion primitives increases, building an intuitive mapping from motion primitives to stroke patterns becomes very difficult. Min et al. [53] apply statistical analysis techniques to a set of capture data containing similar motions, and construct a low-dimensional deformable model to represent the motion. They allow the user to sketch timed trajectories in screen space, and the system automatically generates similar, but novel, motions that match the speeds and trajectories defined by the input sketches.

Performance-based approaches allow the user to control the full-body motion of the avatar by directly acting out the desired motion in a performance animation system. Some approaches utilize computer vision techniques to reconstruct motions from video streams [38, 64, 7], while some others rely on special devices to ease the process of performance capture. Yin et al. [90] developed the FootSee system that estimate the user's pose by measuring the foot pressure. Slyper and Hodgins [74] sew multiple accelerometers on the performer's shirt and animate the avatar by continuously matching the accelerometer readings with the motion capture data. Shiratori and Hodgins [72] make use of the acceleration sensing provided by the Wiimote, allowing the user to control the avatar by acting with Wiimote in hands or on the legs. Although it is intuitive for the user to control the motion with direct acting, the avatar's ability is limited to what can be performed. On the contrary, with puppet interfaces [25, 57, 91], the user can specify difficult motions effortlessly by manipulating puppets equipped with sensors.

1.2 Contribution

The contributions of our work can be summarized as follows:

- **Bidirectional search on motion graphs:** We present a bidirectional search algorithm to improve the search efficiency for near-optimal motion synthesis using motion graphs. Our approach can reduce the maximum search depth by almost a factor of two, leading to significant performance improvements. To fully exploit the potential of bidirectional search, we propose to dynamically divide the search space into two even halves, and use efficient data structures to limit the overhead required to merge the search results.

- **Intuitive sketching interface for interactive motion synthesis:** As motion databases grow bigger, controlling varied motions with conventional interfaces becomes increasingly difficult. Even with traditional sketching systems, users still need memorize a list of mappings from inputs to motions. Our system, however, does not require any memorization but allows users to control the motion intuitively by sketching a trajectory on a specified body part. This is achieved by searching and composing a sequence of motions whose projected trajectory best matches the input.

- **Precise and real-time motion control:** We present an approach to incorporate parameterized motions and interpolation into the reinforcement learning framework. This allows us to control characters precisely with a limited amount of input data. We use a tree-based fitted iteration algorithm to learn control policies. This approach is more flexible and more robust than previous methods to construct motion controllers using reinforcement learning.

- **Autonomous characters with depth perception:** In traditional reinforcement learning frameworks, the state models are explicit descriptions of the environment, which need to be carefully parameterized to limit the dimensionality. We propose a method to skip this design phase, by letting the character "see" the environment directly with depth perception. We avoid the "curse of dimensionality" by introducing a hierarchical state model and a novel regression algorithm. We demonstrate that our controllers allow a character to navigate or survive in environments containing arbitrarily shaped obstacles, which is hard to achieve with existing reinforcement learning frameworks.

1.3 Dissertation Overview

This dissertation is organized as follows. Chapter 2 reviews the fundamental concept of motion graphs and summarizes varied graph search algorithms for online and offline motion synthesis. Chapter 3 introduces our bidirectional search strategy using motion graphs, and demonstrates its efficiency with our intuitive sketch interface for interactive motion synthesis. Chapter 4 reviews reinforcement learning and describes the details of constructing a motion controller with reinforcement learning. Chapter 5 introduces our approach to incorporate parameterized motions and interpolations in reinforcement learning frameworks for precise motion control. Chapter 6 presents our motion controllers with adaptive depth perception. Chapter 7 concludes the dissertation with a summary and directions for future work.

Chapter 2
Search with Motion Graphs

"Again, you can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future."

Steve Jobs

Realistic motion synthesis is a core topic in computer animation. Many of the most successful techniques are based on recombining motion fragments using motion graphs. In this chapter, we first clarify the notations in Section 2.1. We review the basic concept of motion graphs in Section 2.2 and formulate motion synthesis as a search problem in Section 2.3. We then discuss several search algorithms that are used in previous work for motion synthesis, and analyze their optimality and time efficiency in Section 2.4. In Section 2.5, we address the practical issue of handling repeated states to avoid wasting time on searching the same states over and over again. Finally, we conclude this chapter in Section 2.6.

2.1 Motion Representation

In character animation, the human body is typically modeled with a rigid skeleton. A skeleton is represented hierarchically as a tree, in which joints are nodes and bones are links, and there is no cyclic bone connection. The root is usually located at the pelvis joint, or at one of the foot joints. A *pose* or a *motion frame* is thus defined as,

$$\mathbf{f} = (x_0, q_0, q_1, \ldots, q_n), \tag{2.1}$$

where x_0 and q_0 denote the translation and rotation of the root joint, and $q_1, \ldots, q_n$ denote the rotations of all the other joints. From this representation, we can use forward kinematics to derive all the joint positions,

$$\mathbf{p} = (x_0, x_1, \ldots, x_n). \tag{2.2}$$

A *motion* consists of a series of poses over time, $\mathbf{m} = (\mathbf{f}_1, \ldots, \mathbf{f}_T)$, where $\mathbf{f}_t$ denotes the pose at discrete time step t, and T is the length of the motion. The motion can be cut into smaller pieces for reassembling, and the terms *motion segments*, *motion clips*, or *motion fragments* can be used interchangeably to refer to the pieces.

2.2 Motion Graphs

Inspired by video textures [69], which are seamless streams of video synthesized from example footage, motion graphs are developed to allow a long sequence of motion to be synthesized from short motion clips. A motion graph is a directed graph, constructed by taking a set of motion capture clips as input. Each frame in the input motion clips is represented as a graph node, and two nodes are connected by a directed edge if there exists a smooth transition from one frame to another. An example of a motion graph is shown in Figure 2.1.

A few approaches have been proposed to build the motion graphs automatically, by computing smooth transitions among frames. To compute the similarity of two frames i and j, Kovar et al. [31] propose to evaluate the minimal weighted difference in joint positions:

$$\Delta(i, j) = \min_{\theta,x,z} \|\mathbf{w}(\mathbf{p}_i - T_{\theta,x,z}\mathbf{p}_j)\|^2 \tag{2.3}$$

where the linear transformation $T_{\theta,x,z}$ aligns the two sets of joint positions $\mathbf{p}_i$ and $\mathbf{p}_j$ with a rotation around the up axis of θ degrees and a translation of (x, z) on the ground. The weight vector $\mathbf{w}$ assign difference importance for each joint.

If $\Delta(i, j)$ is low, node i and node j are similar, then node i can be connected to node $j + 1$ (successor of node j), and node j can be connected to node $i + 1$ (successor of node i). Originally, nodes are only connected to their

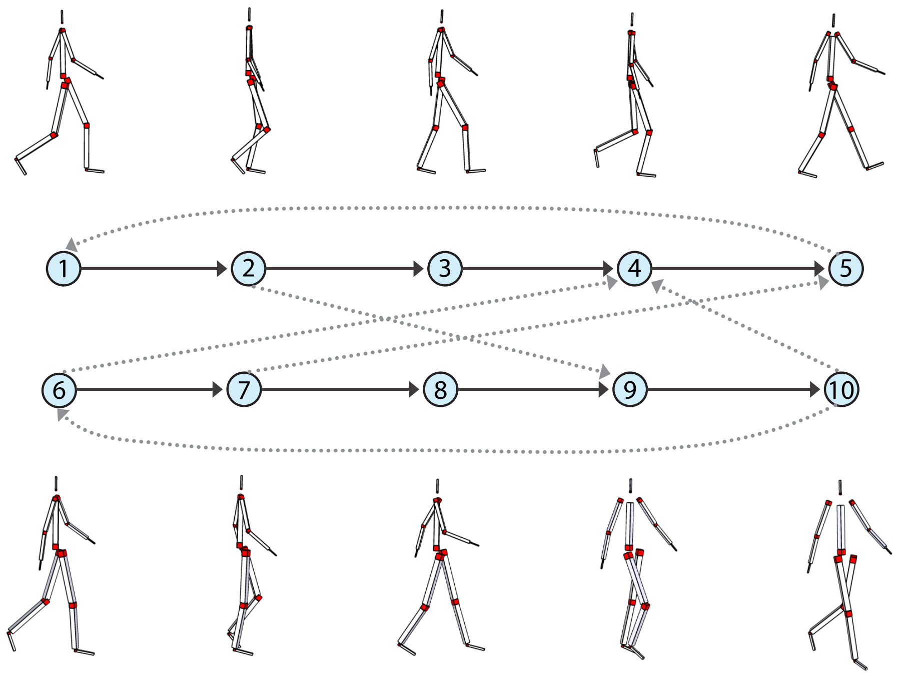
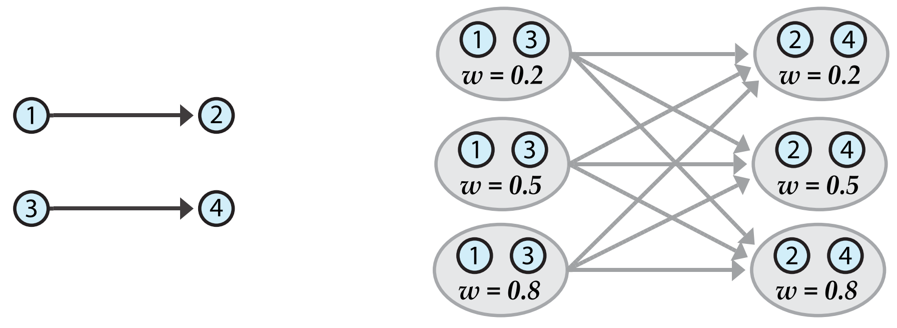


Figure 2.1: An example motion graph formed with two motion clips. Each graph node corresponds to a pose (frame) of the input motion clips and an edge represents a smooth transition between two nodes. The solid edges are original from the motion sequences, and the dotted edges are additional edges found between similar poses.

successor nodes according to the motion sequence. The similarity metric is used to compute a dense transition matrix for every pair of motion clips. The transition value at row i and column j is $\Delta(i, j)$ thresholded by a user specified value for naturalness. To make the set of new edges compact, only transitions that represent local minima in the matrix are added to the graph. These new edges create natural transitions that are new to the motion capture data. Since not every node is guaranteed to have outgoing edges, in the end the largest strongly connected component is extracted from the graph to remove all dead ends.

Once the motion graph is constructed, a path on the graph corresponds to a motion generated by connecting the poses along the path. Traversing the graph randomly will generate a continuous stream of motion. However, to have control over the final motion, e.g. requiring the character to perform some



(a) Motion graph (b) Interpolated motion graph

Figure 2.2: An example interpolated motion graph (b) built from a motion graph (a).

action at some specific time, we need to search a path on the graph that fulfills the constraints. Motion synthesis is thus cast as a graph search problem.

Before explaining how to formulate motion synthesis as a graph search problem in the next section, we introduce two variations of motion graphs: interpolated motion graphs and well-connected motion graphs. Once built, these variations can be treated as ordinary motion graphs in any graph search framework. In Chapter 3, we will show comparisons by applying our bidirectional search algorithm on well-connected motion graphs and interpolated motion graphs.

2.2.1 Interpolated Motion Graphs

Safonova and Hodgins [68] construct an *interpolated motion graph* (IMG) that supports interpolation of paths through the original motion graph. Constructing interpolated motion graphs is like taking the product of two identical motion graphs, as illustrated in Figure 2.2. Each node in the IMG is defined as (a_1, a_2, w), where a_1 and a_2 correspond to two nodes in the input motion graph, and w is the interpolation weight. Hence, the maximum number of nodes in the IMG is N^2W, where N is the number of nodes in the input motion graph, and W is the number of possible weight values. An IMG node (a_1, a_2, w) is a successor of another IMG node (a'_1, a'_2, w') if a_1 is a successor of a'_1 and a_2 is a successor of a'_2 in the original motion graph. The strength of this representa-

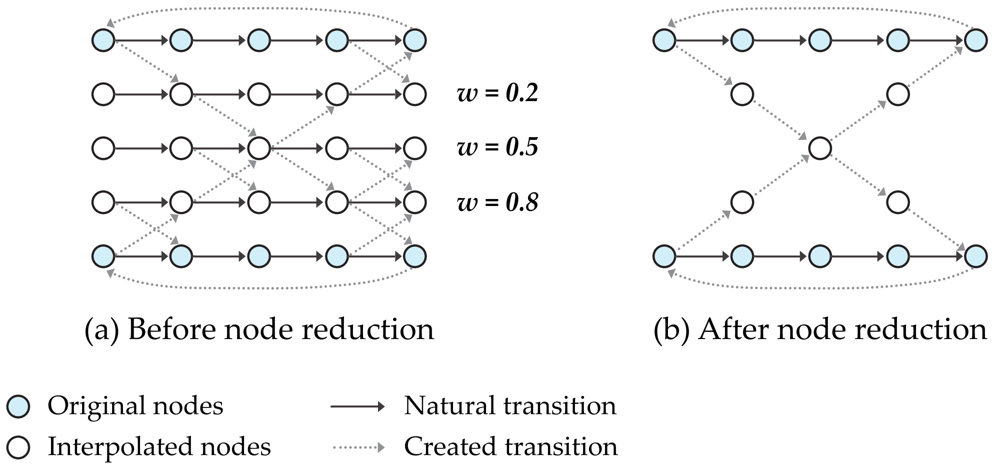


Figure 2.3: An example well-connected motion graph before (a) and after (b) the node reduction step. The input motions are interpolated with three different weights to generate the interpolated nodes. A transition is created between two nodes (either original or interpolated) if the poses are similar.

tion is that it allows the adaptation of existing motions through interpolation while also retaining the natural transitions present in a motion graph. However, the result IMG is inherently very large. Safonova and Hodgins present a few methods to reduce the graph size, and please refer to their paper for more details.

2.2.2 Well-connected Motion Graphs

Zhao and Safonova [93] introduce a new method for building motion graphs, and the outcome of their algorithm is a motion graph called a *well-connected motion graph*, which contains only smooth transitions but has very good connectivity. One fundamental problem of traditional motion graphs is that the quality of the generated motion depends largely on the connectivity of the graph and the quality of transitions in it. Achieving both criteria simultaneously, however, is difficult. Good connectivity requires transitions to exist between most pairs of poses, but this might degrade the motion quality, which can only be obtained when transitions happen between very similar poses. To resolve the tradeoff, their algorithm builds a set of interpolated motions, which contain many more similar poses than the original data set, and then constructs a graph from all original poses and interpolated poses, as shown in Figure 2.3a. Finally, they reduce the graph size by minimizing the number of interpolated poses present in the graph, as shown in Figure 2.3b. The result graph is still a motion graph, which can benefit from all the standard techniques for motion graphs, but has good connectivity without sacrificing the transition quality.

2.3 Problem Formulation

In this section, we describe how to formulate the search problem for synthesizing a motion that satisfies user specifications while minimizing a cost measure. Before going into the details, we first provide some notation. The motion graph is denoted as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes or poses, and $\mathcal{E}$ is the set of edges. The *state space*, $\mathcal{S}$, is defined as the range of all possible states the character might be in. The definition of the state $X \in \mathcal{S}$ is problem dependent, but at least includes the pose of the character. For example, in the commonly addressed *ground navigation* problem, where the character is required to move from a start position to a goal position, the state can be defined as:

$$X = (a, x, z, \theta), \tag{2.4}$$

where $a \in \mathcal{V}$ is the index of the current pose, x and z are the global position of the character on the ground plane, and θ is the global orientation of the character around the up axis.

The search problem is defined by the following components:

- The **initial state** that the character starts with, which includes the start pose and other problem dependent properties.

- The set of **actions** available for the character and a **successor function** that returns all successor states of a given state. Given a state $X = (a, \ldots)$, the set of available actions is decided by the motion graph G and is defined as $A(X) = \{a' | (a, a') \in \mathcal{E}\}$. The successor function $F(X)$ returns a set of states that can be reached from X by applying actions in $A(X)$.

- The **goal test**, which determines if a state is a goal state. For example, in the ground navigation problem, a state, $X = (a, x, z, \theta)$, is a goal state if

a corresponds to a resting pose, $\|x - x_g\| < \varepsilon$, and $\|z - z_g\| < \varepsilon$, where (x_g, z_g) is the goal position, and ε represents the tolerable deviation from the goal position.

- The **cost** function that assigns a numeric measure to each *path*, which is defined as a sequence of states starting from the initial state. Typically, the cost function consists of a *transition cost* to ensure smooth transitions between motion fragments and a *state cost* to evaluate how well the constraints are fulfilled. The transition cost can be defined using Equation 2.3, while the state cost is problem specific. For example, if the character is expected to follow some path provided by a user, the state cost consists of the deviations from the path. Also, the state cost can be set to be proportional to the distance traveled to favor shorter and more intuitive solutions.

With the problem formulated, we can start the search by building an explicit *search tree*. The *search node* in the search tree is defined as

$$N = (X, f(X), N'), \tag{2.5}$$

where $X \in S$ is the corresponding state, $f : \mathcal{S} \rightarrow \mathbb{R}$ is the cost function that returns the accumulated cost from the initial state to X, and N' denotes the parent node. The root of the search tree is a search node containing the initial state. Let us denote $\mathcal{Q}$ as the collection of leaf nodes to be *expanded*. Expanding a search node means generating a set of successor nodes that contain the states returned by the successor function. Initially $\mathcal{Q}$ consists of only the root node. The search is done by iteratively removing a node from $\mathcal{Q}$ for expansion until $\mathcal{Q}$ is empty. The goal test should be applied before expanding the selected node. If the selected node contains a goal state, a *feasible* solution is found and its expansion is skipped. Otherwise, the selected node is expanded, and all the successor nodes are inserted into $\mathcal{Q}$. A number of expansions of a search tree are illustrated in Figure 2.4. Finally, the *optimal* solutions are defined as those with the lowest cost among all feasible solutions.

The choice of which search node to be removed from $\mathcal{Q}$ is determined by the search strategy. In the next section, we will discuss a few popular search algorithms for motion synthesis.

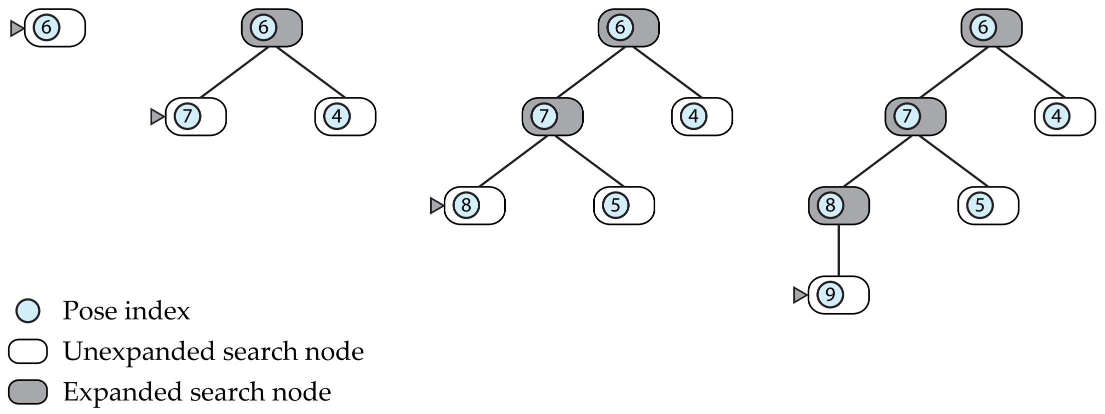


Figure 2.4: Four expansions of a search tree, built with the motion graph in Figure 2.1. The state is defined as $X = (a)$, where a denotes the current pose of the character, e.g. the initial state contains pose 6 in the motion graph. At each stage, the search node to be expanded is indicated by a marker. The expansion order is decided by the search strategy, and this example illustrates the depth-first search.

2.4 Search Algorithms

Once the problem of motion synthesis is formulated as a search problem, it can be solved with a standard search algorithm. The search algorithms are extensively used in practice, and among the best-known applications are route-finding (e.g. in computer networks or travel planning systems), robot navigation, and protein design.

Generally, the search strategies can be classified into two categories: *uninformed search* and *heuristic search* [66]. With uninformed search strategies, there is no additional information about the states beyond that provided in the definition. These strategies can only generate successors and distinguish a goal state from a nongoal state. On the contrary, the heuristic search strategies utilize problem-specific knowledge to prioritize the search nodes. We will discuss examples of uninformed search in Section 2.4.1, and examples of heuristic search in Section 2.4.2 and Section 2.4.3. All these algorithms can be used to find the optimal solution. However, the search complexity grows exponentially with the depth of the search tree. A few variations of the search algorithms are thus proposed for trading optimality for efficiency. We will discuss these variations along with the original search algorithms. Finally, in Section 2.4.4, we discuss some randomized search algorithms aiming at finding a *feasible* solution quickly, but with no guarantee in the result quality.

2.4.1 Breadth-first Search & Depth-first Search

Breadth-first search (BFS) and *depth-first search* (DFS) are the two most popular uninformed search strategies. BFS always chooses the shallowest node in $\mathcal{Q}$ for expansion, while DFS always chooses the deepest one, until a solution is found (which may or may not be optimal) or until $\mathcal{Q}$ is empty. Figure 2.4 illustrates a few steps of the DFS algorithm. Hence, either approach needs to expand the full search tree to find the optimal solution, and these naive approaches are very demanding in both computation and memory requirement.

In order to apply DFS for motion synthesis with motion graphs, Kovar et al. [31] improve the search efficiency with a *branch and bound* strategy and *incremental search*. The branch and bound strategy is used to prune branches of the search that are incapable of yielding the optimum, by keeping track of the current best goal state X_{opt}. Thus, a partial path can be halted if the accumulated cost is already bigger than $f(X_{opt})$. Branch and bound strategy reduces the number of search node expansions without scarifying the optimality. However, the search complexity is still exponential to the search depth.

Incremental search (or limited-horizon search) is further applied to ensure interactive performance, and the solution is generated incrementally. At each step, DFS is used to find an optimal subsequence of n frames, where n is used to maintain a constant rate of motion, and the final node is used as a starting point for another search. Although each partial solution is optimal, the final solution is sub-optimal. The value of n also controls the tradeoff between efficiency and optimality.

2.4.2 Greedy Best-first Search

Greedy best-first search and A* search (Section 2.4.3), belong to the family of heuristic search algorithms. Heuristic search finds solutions more efficiently than uninformed search by using problem-specific knowledge. The key component of heuristic search is a *heuristic function* that imparts additional knowledge to the search algorithm. The heuristic function is denoted as $h : \mathcal{S} \rightarrow \mathbb{R}$ that returns the estimated cost of the cheapest path from the given state to a goal state.

Greedy best-first search always expands the node in $\mathcal{Q}$ that is closet to the goal, i.e. the node with lowest value of $h(X)$. This algorithm is named greedy for that at each step it tries to get as close to the goal as it can. This algorithm may find a feasible solution more efficiently than uninformed search algorithms. To find the global optimal solution, however, it also suffers from the fact that expanding the full search tree is necessary.

Lee et al. [38] allow a user to sketch a path through the environment in their system, and use greedy best-first search to find a motion that follows the path. Their heuristic function consists of two parts. The first part rewards motion making progress to the goal, and the second part rewards facing directions that point toward the goal. To maintain a constant rate of motion, they also apply incremental search by diving the sketched path into many sub goal points. They noted that this approach could result in local minima where the character can no longer reach the goal, but in practice it is not a problem, since the user does the high-level planning by sketching an appropriate path.

2.4.3 A* Search

A* is the most widely-known heuristic search, and it evaluates a node by combining $f(X)$, the accumulated cost to reach the node, and $h(X)$, the estimated cost to get from the node to the goal,

$$g(X) = f(X) + h(X). \tag{2.6}$$

The function $g : \mathcal{S} \rightarrow \mathbb{R}$ returns the estimated cost of the cheapest solution through state X from the initial state to the goal. A* search always expands the node in $\mathcal{Q}$ that minimizes g, and the search terminates either when $\mathcal{Q}$ is empty, or when

$$\forall N = (X, f(X), N') \in \mathcal{Q} : g(X) \geq f(X_{opt}), \tag{2.7}$$

where X_{opt} is the current best goal state. A* search is optimal if the heuristic function h is *admissible*, that is, if $h(X)$ never overestimates the cost to reach the goal. Since $f(X)$ is the actual cost, with an admissible heuristic h, $g(X)$ never overestimates the real cost through X. Hence, we can conclude that when Condition 2.7 holds, X_{opt} provides the optimal solution. Furthermore, A* is *optimally efficient* for any given heuristic function: no other optimal algorithm is guaranteed to expand fewer nodes than A* [66].

A* search is optimal and optimally efficient, so it is becoming a favorable approach in computer animation for searching an optimal sequence of motions [35, 68]. However, the quality of the heuristic function is a critical factor of the search performance. Informative heuristic functions can significantly prune the search tree, guiding the search toward the optimal path, while uninformative heuristic functions require exhaustive exploration of the search space. Safonova and Hodgins [68] show that heuristic functions usually speed up the search by several orders of magnitude and is often the determining factor in whether a solution can be found. In their system, they allow a user to sketch a path through the environment, and a motion is synthesized to follow the path. To estimate the lower bound of the cost-to-goal value $h(X)$, they apply Dijkstra's algorithm to find the shortest path to the goal, and then multiply the shortest distance (in meters) by an estimate of the minimum cost function value required to traverse one meter. Additional heuristics are also provided to support environmental constraints (e.g. picking, jumping, ducking, and sitting).

Although A* search is optimally efficient, it is still an exponential search method. To increase the size of motion graphs or to apply the search in interactive applications, a few variants of A* search have been used to improve search efficiency at the expense of optimality:

- **Inflated A*** inflates the heuristic with $\delta > 1$ to speed up the search, and the new heuristic function becomes $h'(X) = \delta h(X)$. The inflated heuristic is not necessarily admissible, so the solution is no longer optimal. However, the cost of the solution is bounded above by δ times the lowest cost. Since inflated A* search has a bound on the sub-optimality, it is *near-optimal*. Lau and Kuffner [35] compare the synthesized motions generated from A* search and inflated A* search, and note that there is a treadoff between the search time and the quality of the output motion. Please refer to their work for the detailed comparisons.

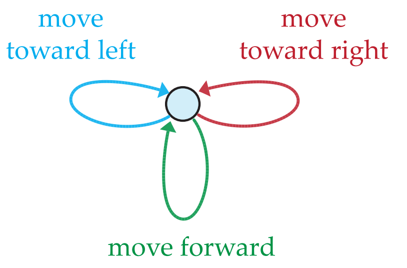
- **ARA*** (Anytime Repairing A*) is an anytime heuristic search that tunes the sub-optimality bound based on available search time [47]. ARA* starts by finding a suboptimal solution quickly using a progressively inflated heuristic and then gradually decreases δ until it runs out of time or finds a provably optimal solution. While improving its bound, ARA* reuses search efforts from previous executions in a way that the sub-optimality bounds are still satisfied. Safonova and Hodgins [68] apply ARA* in their work to find a globally optimal or near-optimal solution, and show that the optimal path through the motion graph often leads to the most intuitive result.

2.4.4 Randomized Search

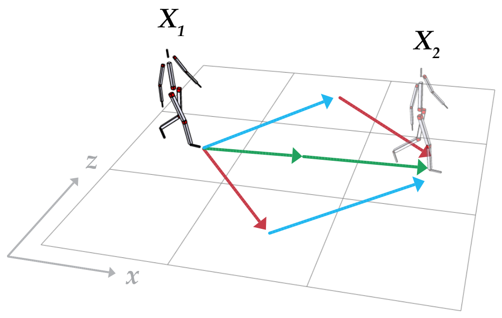
Applying optimal or near-optimal search in online applications is challenging, especially when the search space is large. Hence, some algorithms employ randomness and aim at finding a feasible solution as quickly as possible.

Arikan and Forsyth [1] propose a randomized search algorithm to allow interactive synthesis of human motions. The algorithm starts by building a set of valid random solutions. In each iteration, the solutions are mutated by replacing some portion with another set of edges, and the mutations are accepted only if they are better than the original solutions. The solutions are iteratively refined until no better results can be obtained through mutations or until a given time runs out. The search can be stopped anytime to return the best solution found so far, and if the result is not good enough, the search can also be resumed to get better solutions through further mutations and inclusion of random solutions.

The *Rapidly-Exploring Random Tree* (RRT) planner is popular in motion planning [26]. The basis of the RRT method is the incremental construction of search trees that attempt to rapidly and uniformly explore the state space. This can be considered as a Monte-Carlo way of biasing the search into the largest Voronoi regions. Choi et al. [11] use RRT for global path planning in their work. A RRT is rooted at the start point and grows to explore the free space by iteratively sampling new nodes. They randomly sample a free configuration (position and direction) and find its closest node belonging to the search tree. If there exists a valid motion from the closet node to the sampled configuration,



(a) A simplified motion graph



(b) Ground navigation

Figure 2.5: (a) A simplified motion graph that contains only one node. (b) A ground navigation problem where the state is defined as $X = (a, x, z)$. The initial state of the character is X_1. With the motion graph in (a), there are multiple ways to reach state X_2, resulting in multiple search nodes containing repeated states.

a new branch is added to the tree. This process repeats until the search tree reaches the goal location. In addition, Shapiro et al. [70] use a bidirectional RRT algorithm to correct input motions to obey constraints such as being collision-free.

2.5 Pruning Repeated States

One of the most important complications to the search process is the possibility of wasting time on expanding states that have been encountered and expanded before [66]. However, repeated states are unavoidable when searching with motion graphs, since the actions defined by the motion graph are reversible, as shown in Figure 2.5. Repeated states can cause a solvable problem to become unsolvable if the search algorithm does not detect them. Detection usually means comparing the search node about to be expanded to those have been expanded: if a match is found in the state space, then the algorithm has discovered two paths to the same state and can discard one of them [66].

One common solution to prune the repeated states is to use a set $\mathcal{C}$ to keep track of the explored states. The set $\mathcal{C}$ can be implemented with a hash table to allow efficient look up for repeated states: every time a node N is expanded, the associate state X is inserted into $\mathcal{C}$. To preserve optimality, we also need to bookkeep the current lowest costs for reaching every state in $\mathcal{C}$, so whenever repeated states are detected, the more expensive path is discarded. We use a similar approach in our work (Chapter 3) for pruning repeated states and for merging two partial searches.

If the search algorithm is A*, we can avoid the overhead of bookkeeping the lowest cost of expanded states, but ensure the optimal path to any repeated state is always the first one by imposing an additional constraint on the heuristic function. This requires the heuristic function to be *consistent*. A heuristic function $h(X)$ is consistent if for every node $N = (X, \ldots)$ and every successor $N' = (X', \ldots)$ of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to N' plus the estimated cost of reaching the gaol from N',

$$h(X) \leq f(X') - f(X) + h(X'). \tag{2.8}$$

Following the definition of consistency and Equation 2.6, the sequence of nodes expanded is in non-decreasing order of g,

$$g(X') = f(X') + h(X') \geq h(X) + f(X) = g(X). \tag{2.9}$$

Hence, in the search process, if the node selected for expansion is found repeating in $\mathcal{C}$, it can be directly discarded, and the first expanded goal node is guaranteed to be an optimal solution [66].

Although pruning repeated states is necessary, it introduces additional computation overhead in run-time. In the rest of this section, we will discuss some approaches that utilize precomputation to avoid handling repeated states at run-time.

2.5.1 Precomputed Search Trees

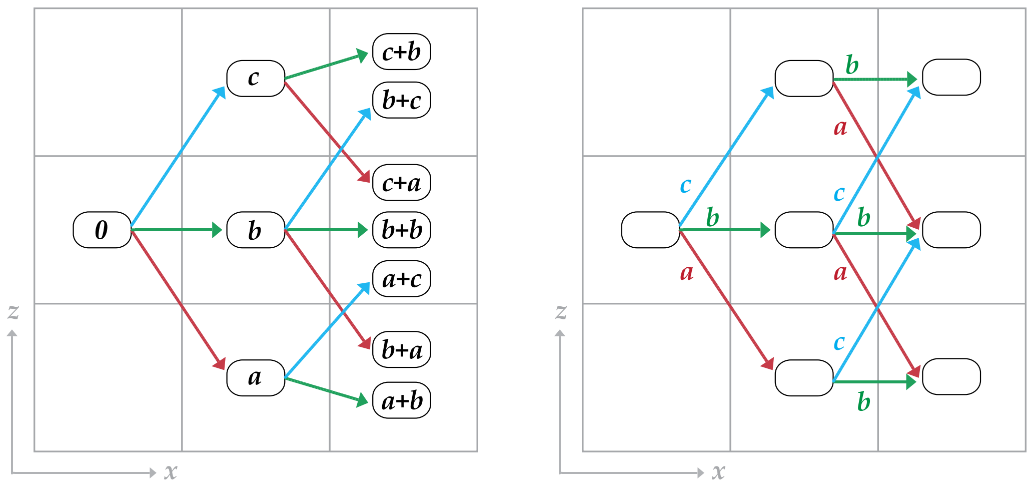
Lau and Kuffner [36] precompute the search trees so that the optimal motion sequence can be efficiently extracted in run-time through a series of table lookups. The precomputed search tree is an exhaustive search tree built off-line up to a certain depth d. They apply the precomputed search tree for navigating characters in complex environments: given a goal position, the objective is to animate the character from the start position to the goal in a best way without colliding with any obstacle. The state is defined as in Equation 2.4. The search nodes are organized with a *gridmap*, which is a discretization of the ground plane, and each node is stored in a cell according to the discretized values of x and z. The discretization level and the scale of the gridmap are parameters of the system. A cell in the gridmap may contain more than one node, due to the repeated states, and they sort the nodes with respect to the cost. An example precomputed search tree and the associated gridmap are shown in Figure 2.6a.

At run-time, the precomputed tree is aligned with the character's initial state and is mapped onto the environment: if a cell in the gridmap is occupied by an obstacle, the tree nodes corresponding to this cell and their descendant nodes are *blocked*. A blocked node means it is not reachable. The search algorithm starts by extracting the tree nodes stored in the cell that contains the goal position. The algorithm then iterates through the nodes, in the ascending order of cost, to see if any of them can successfully backtrack to the root without being blocked. The first complete path is returned as the solution, which is near optimal up to the resolution of the gridmap.

A downside of this approach is that if the goal position is distant from the start position, the precomputed search tree may not be able to cover both of them since it is only expanded up to a certain depth d. Increasing the value of d, however, results in exponential growth of memory requirements. Hence, for long sequences of motion, Lau and Kuffner use a fast global planner to generate a coarse path from the start to the goal. This path is divided into several sub-goals such that any two consecutive sub-goals are covered in the range of the precomputed search tree. The final solution is a composition of several partial solutions, and is no longer globally optimal.

2.5.2 Precomputed Search Graphs

Given a large state space, where most of the states can be reached from many paths, a *search graph* is more favorable than a search tree, since the latter needs to trade optimality for memory usage. The search graph is precomputed by building an exhaustive search tree to cover the entire state space, but the search nodes with same states are merged and the costs are stored along the links, as shown in Figure 2.6b. The search graph can be traversed with any



(a) Precomputed search tree (b) Precomputed search graph

Figure 2.6: A precomputed search tree (a) and a precomputed search graph (b) generated with the motion graph in Figure 2.5a in the ground navigation problem in Figure 2.5b. The environment is discretized into 3×3 grids. The transition costs are a, b, and c for “move toward right”, “move toward left”, and “move forward” respectively. In the precomputed search tree (a), the accumulated costs are stored in the search nodes, and there may be more than one search node stored in a grid (containing a same discretized state). In the precomputed search graph (b), the transition costs are stored along with the links, and only one search node is kept in a grid.

standard search algorithm in run-time to find a solution. However, in order to find the optimal solution, the search algorithm needs to check whether a node has been visited and whether the newly discovered path to the node is better than the original one. If so, the search queue needs to be updated according to the new cost, otherwise the optimal solution might be missed. On the contrary, some search algorithms under some conditions can guarantee that the newly discovered path is never better than the original one, e.g. A* search with consistent heuristic or breadth-first search with constant link costs. In these situations, there is no overheads in tracking the visited nodes, as the newly discovered paths can be directly discards, and the first path reaching the goal state is the optimal solution. Finally, the precomputed search graph can take advantage of graph pruning algorithms, so that the graph size can be much reduced in a preprocessing stage to speed up the search in run-time.

Safonova and Hodgins [68] build the search graph off-line by unrolling the motion graph into the environment (Figure 2.5 illustrates how to unroll a motion graph in Figure 2.5a into an environment in Figure 2.5b). The state is defined as in Equation 2.4, and the values of the position and orientation are discretized uniformly. Nodes with the same discretized values in the state can then be merged into a single node. The search graph is inherently large, with a size proportional to the size of the motion graph and the scale and resolution of the environment. To make the on-line search more efficient, Safonova and Hodgins propose to compress the motion graph before unrolling it into the environment. They first remove poses and transitions that are sub-optimal, which will not appear in any optimal solution because there exists a lower cost alternative. Next, they remove redundant poses and transitions in the motion graph. Since motion graphs often include similar data in order to capture natural transitions between behaviors, many poses and transitions are redundant. They report that these compression steps can significantly reduce the size of the motion graph without affecting the its functionality. In run-time, they apply ARA* search with a consistent heuristic function to search the precomputed graph.

A *probabilistic roadmap* is another form of a search graph, which can be built in the preprocessing stage and efficiently searched at run-time to achieve interactive performance [10]. The nodes in a probabilistic roadmap are random samples in state space. A pair of nodes are connected by an edge if there exists a motion to animate the character from one node to another. The probabilistic roadmap could be constructed to densely cover the entire environment. Precomputing a dense roadmap, however, demands significant computational resources. Alternatively, Choi et al. [11] present a lazy evaluation approach. Given start and goal configurations, they incrementally add random samples to the roadmap until a path between the start and the goal is found. Each new sample is connected to the existing roadmap with limited-horizon A* search algorithm. This approach gives a trade-off between enormous precomputation costs and runtime performance.

2.6 Summary

In this chapter, we review the concept of motion graphs and explain how to formulate the problem of motion synthesis as a standard search problem. We also introduce a few search algorithms used in character animation for solving the problem, including depth-first search, greedy best-first search, A* search, and some randomized search techniques. Depth-first search and greedy best-first search need to exhaust the search space to find the optimal solution, so researchers usually apply these search techniques with some heuristic strategies to trade in optimality for search efficiency. A* search is able to find the optimal solution without exhausting the search space, but the search efficiency strongly depends on the quality of the heuristic function. Two variants of A* search, inflated A* and ARA*, are thus more applicable for on-line applications. Although these variants only provide sub-optimal solutions, there is a bound on the sub-optimality, so the solution quality can be controlled. Randomized search techniques, on the contrary, aim at finding a feasible solution as quickly as possible, so there is no guarantee about the quality of the solution.

Finally, we discuss two types of precomputed structures, precomputed search trees and precomputed search graphs, that are built off-line to make on-line search more efficient. In a precomputed search tree, all the search nodes containing the same state are stored explicitly, so given a goal state, a solution can be found efficiently by tracing from the goal state back to the root. However, the precomputed search tree can only grow up to a certain depth, due to memory limitation. If the distance between the start and goal states is bigger than the range of the tree, a solution can only be found incrementally, and there is no guarantee on the optimality. On the other side, in a precomputed search graph, the search nodes containing the same state are merged into a single node, so a node in the precomputed search graph may contain more than one successor or predecessor. In run-time, a search algorithm is applied on the precomputed graph to decide which successor node should be visited first. The optimal solution can be efficiently found if the search algorithm ensures that the first path to a node is always the cheapest, otherwise a data structure is required for bookkeeping the lowest costs to reach the visited nodes. A main advantage of the precomputed search graph is that it can be further pruned in the preprocessing stage to speed up run-time search.

In the next chapter, we will present a bidirectional search framework that is orthogonal to many existing search algorithms (e.g. BFS, DFS, A* and its variants), and can reduce the maximal search depth by almost a factor of two, leading to significant performance improvements. Since A* search is proved to be optimally efficient, we will apply A* search in our bidirectional framework to demonstrate the performance improvement.

Chapter 3

Bidirectional Search with Motion Graphs

“Where both reason and experience fall short, there occurs a vacuum that can be filled by faith."

Sophie's World
Jostein Gaarder

In this chapter, we develop a novel method to improve the search efficiency for near-optimal motion synthesis using motion graphs. An optimal or near-optimal path through a motion graph often leads to the most intuitive result. However, as discussed in Chapter 2, finding such a path can be computationally expensive. Our main contribution is a bidirectional search algorithm. We dynamically divide the search space evenly and merge two search trees to obtain the final solution. This cuts the maximum search depth almost in half and leads to significant speedup. Many applications can thus be improved to achieve interactive performance. To illustrate the benefits of our approach, we present an interactive sketching interface that allows users to specify complex motions quickly and intuitively.

3.1 Contributions

A key contribution of this chapter is a bidirectional search algorithm to improve the search efficiency for near-optimal motion synthesis. Search algorithms have been widely applied on the motion graph for searching a sequence of motion that fulfills user constraints [31, 1, 38, 10, 35, 68, 11]. The search complexity for an optimal or near-optimal solution, however, is exponential to the connectivity of the graph and the length of the desired motion sequence.

Applying these techniques for interactive applications is thus challenging. To make the search algorithms more applicable for interactive applications, we present a novel approach to run two searches simultaneously from both ends of the desired motion. This allows us to reduce the search depth by a factor of almost two, leading to a significant performance improvement. A core component of our bidirectional search algorithm is a novel technique to efficiently merge the two search trees to obtain one continuous motion sequence. We dynamically divide the search space to ensure both trees have similar height, and we cache partial paths using hash tables so that we can merge the two trees efficiently.

Our second contribution is an intuitive sketch interface for interactive motion synthesis. Today's video games include characters with a rich set of behaviors. It becomes increasingly difficult to control varied motions with conventional interfaces, such as joysticks, game pads, mice, or keyboards, since they often require memorizing awkward keystroke combinations or gestures. Some early works allow a user to sketch the path of the desired motion [38, 31], while more recent works allow a user to generate a wider variety of motions by drawing simple strokes [60, 79]. With these systems, however, users still need to memorize a list of mappings between stroke patterns and motions. On the contrary, our system does not require any memorization but allows users to control the motion intuitively by sketching a trajectory on a specified body part. Given a stroke input, we search and compose a sequence of motions whose projected trajectory best matches the input. Interactive performance is made possible with our bidirectional search algorithm.

3.2 Bidirectional Search

Bidirectional search has been explored in artificial intelligence and path planning [66, 37]. The critical step is to design a mechanism to merge the two partial searches, and to prevent two search frontiers from intersecting with each other. Without proper designs, bidirectional search may lead to worse performance than unidirectional search [62, 33]. Kandl and Kainz [28] presented one of the first successful approaches to bidirectional search and demonstrated that it can be more efficient than unidirectional search. They allow the search direction to change once, and hash the frontier nodes in the first search to cut off some branches in the reverse search meeting the opposite frontier. Although their approach is not afflicted by the problem of search frontiers passing each other, the search efforts cannot be well balanced unless the direction is changed exactly when the search is halfway. Predicting the halfway search in advance, however, is very difficult.

In our work, we exploit properties of the state space in motion synthesis to design a bidirectional framework, where the search space is dynamically divided into two even halves, and the two search trees can be efficiently merged to obtain the final solution. We define a cut to split the search space so that two search frontiers cannot pass each other. Since it is difficult to predict the halfway search in advance, we allow the cut to be dynamically adjusted at run-time. We hash the frontier nodes from both frontiers so that two partial solutions can be efficiently merged. Finally, to accommodate the dynamically adjusted cut, we provide a mechanism to efficiently update the hashing information of both frontiers. Since A* search is optimally efficient and considered the state-of-the-art technique for near-optimal motion synthesis (Section 2.4.3), we demonstrate our approach using bidirectional A* search.

3.2.1 Overview

We propose a bidirectional algorithm that can make existing search algorithms, especially A* and its variants, more efficient. If unidirectional search has a branching factor, i.e., an average number of successors of any node, of b, then the search space for finding a path of length d is on the order of $O(b^d)$. A bidirectional algorithm, however, can reduce the size of search space to $O(b^{\frac{d}{2}})$. In our application, this translates into significant speedups as we will show in Section 3.4.1 and Section 3.4.3.

The performance of bidirectional search strongly depends on two factors:

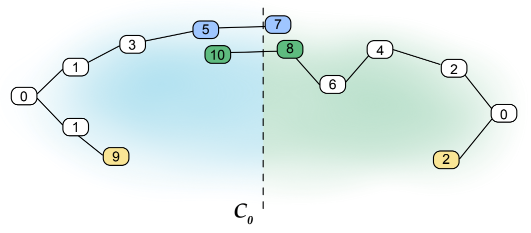
- The **stopping criteria** that determines when to stop the searches from each direction.
- The **merge query** that efficiently evaluates potential merge points between the two searches.

If they are not properly designed, the search performance may be worse than unidirectional A*. The core component of our bidirectional search algorithm is a novel strategy to design these factors for motion synthesis. We address the first issue by defining and dynamically adjusting a global *cut*, which determines where the search is stopped. To solve the second issue, we introduce an efficient data structure that we call a *merge hash table* (MHT).

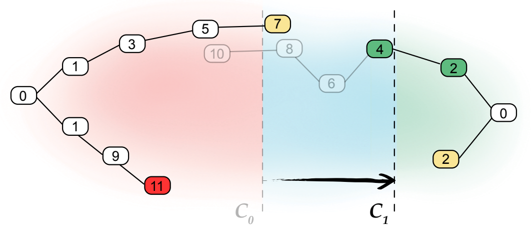
3.2.2 Dynamic Cut Adjustment

The search starts by alternately expanding two search trees from both ends by maintaining two priority queues. We expand both trees until they reach the *cut*. The idea is that the cut halves the *search space* so the work required to expand both trees is well balanced. The search space is defined as the collection of all possible solutions, and a solution is a sequence of search nodes $(N_1, \ldots, N_l)$, where a search node N_i is as defined in Equation 2.5 and l is the length of the solution. Ideally, the cut is placed in a way that for any solution of length l, no further expansion is made beyond $N_{\frac{l}{2}}$ The search depth d is thus halved, and the search performance is improved from $O(b^d)$ to $O(b^{\frac{d}{2}})$. The midpoints of the solutions, however, cannot be determined in advance, so we utilize the property of general motion synthesis problems to approximate the cut in the state space $\mathcal{S}$ (the definition of a state can be found in Section 2.3). Even with the state space, without sufficient information about how the search will proceed, it is still difficult to place the cut properly at the beginning of the search. Hence, we place an initial cut between the start and end states as a hyperplane that splits the state space into two equal halves. Whenever a node passes the cut, no further expansion can be made, since the other side will be explored by the other search tree, as illustrated in Figure 3.1a

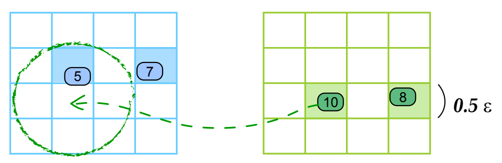
The performance of bidirectional A* largely depends on where the cut is placed, since this determines how the search space is split. The initial cut, estimated in the state space, may not divide the search space evenly, because search depth is not directly proportional to distances in state space. For example, some arcs in the search tree, such as the jumping motion, span a longer distance in state space than others, such as a small step. Hence, one tree may surpass the cut and terminate much earlier than the other. It is also possible



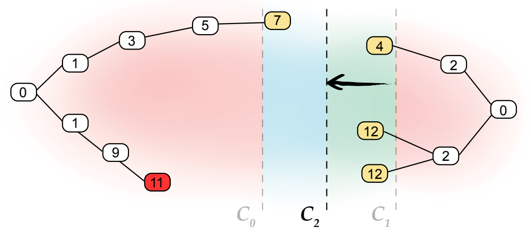
(a) Bidirectional search and initial cut



(c) First adjustment of cut



(b) Merge hash tables



(d) Second adjustment of cut

Figure 3.1: Bidirectional search: (a) We expand two search trees alternately from the start and end states. The numbers in the nodes indicate the time they are created. C_0 indicates the initial cut, and we visualize the two halves of state space in blue and green. Nodes in search queues are yellow. When links cross the cut we insert the nodes on both sides (e.g. 5-7 and 8-10) into the corresponding MHT. (b) When inserting a node into a MHT, we check its neighboring states in the opposite MHT to find all possible matches. (c) The left tree stops growing since all its frontier nodes either pass the cut (node 7) or have costs exceeding the current minimal total cost (node 11). The fully explored space is shaded in red. To keep both trees growing, we move the cut to the right. We update the queues and MHTs accordingly, by putting nodes across the new cut (nodes $2, 4$ in green) into the MHTs, and deleting nodes beyond these pairs (nodes $6, 8, 10$ in grey). Node 7 in the left tree is inserted into the queue again. (d) Some time later, the right tree stops growing before the left tree (all the frontiers pass C_1), so we move the cut to the left again, splitting the unexplored space into two equal halves. Nodes $4, 12$ are then inserted into search queue again.

that one search converges more quickly to the optimal solution than the other. Unfortunately, after one tree's termination the search behavior resembles unidirectional A*, so we would like to keep both trees growing for as long as possible.

We propose an algorithm to adjust the cut dynamically so that it eventually converges to the halfway cut in the search space. If one of the trees terminates before the other, we move the cut toward the other tree so that it splits the remaining, not yet fully explored space into two equal halves. This procedure is iterated every time one tree stops growing (all the frontier nodes either pass the cut or have costs exceeding the current minimal cost). We repeat this process until both trees terminate, or until there is not enough space remaining to make the adjustment. Two iterations of dynamic cut adjustment are illustrated in Figure 3.1c and Figure 3.1d.

3.2.3 Merging Two Search Trees

When a node passes the cut, we need to detect whether the current path can be merged with any existing paths from the opposite search. We say that two paths p_1 and p_2 are *mergeable* if they contain any pair of search nodes N_1 and N_2 that are close enough in state space. More precisely, they are mergeable if $\exists N_1 \in p_1, N_2 \in p_2$ such that $a_1 = a_2$, $|x_1 - x_2| < \varepsilon_x$, $|z_1 - z_2| < \varepsilon_z$, and $|\theta_1 - \theta_2| < \varepsilon_\theta$, where $\varepsilon = (\varepsilon_x, \varepsilon_z, \varepsilon_\theta)$ is the tolerable deviation in (x, z, θ) between two states. Instead of evaluating all pairs of nodes for merging, which takes linear-time computation, we would like to perform the query efficiently.

To achieve efficient merging, we maintain *merge hash tables* (MHT) individually for each search tree. Each MHT records a set of candidate nodes for merging from each side. The main idea is to *quantize* the state space into a grid of cells, and to use the quantized state of each node to compute the key for hashing. Pairs of nodes that can be merged will be guaranteed to fall into neighboring cells. The hash table allows us to retrieve nodes in neighboring cells in constant time while storing the grid efficiently. This approach is a variation of spatial hashing [16], which has been used in graphics for example for collision detection [78, 45].

To quantize the state space we use a cell size of $\frac{\varepsilon_x}{2} \times \frac{\varepsilon_z}{2} \times \frac{\varepsilon_\theta}{2}$. Note that the pose index a is discrete by definition. For each node in the search tree, its x, z, and θ coordinates are discretized to the closest cell and used to compute the hash key. When a node N passes the cut, we insert N and its parent N' into their own MHT (Figure 3.1a). Then we use their states X and X' to query the MHT of the other search tree (Figure 3.1b). For each query, we also check its neighbors. That is, for a key X, we also use $(a, x \pm \frac{\varepsilon_x}{2}, z \pm \frac{\varepsilon_z}{2}, \theta \pm \frac{\varepsilon_\theta}{2})$ to query for matches. If a match is found between two nodes N_1 and N_2, the total cost of the corresponding path is $g(X_1) + g(X_2) + \lambda\|X_1 - X_2\|$, where λ is a scaling constant for the cost in merging two states.

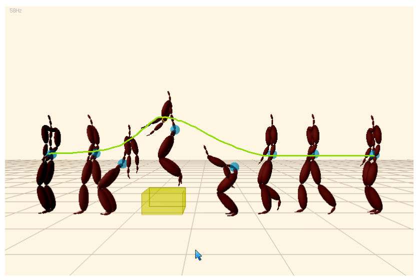
During the entire search we always keep record of the path with minimal cost. Similar to standard A* search, each tree will terminate growing if every node in the queue has greater cost than the best solution so far, or if the queue is empty (in this case, every path surpasses the cut). The search ends when both trees terminate.

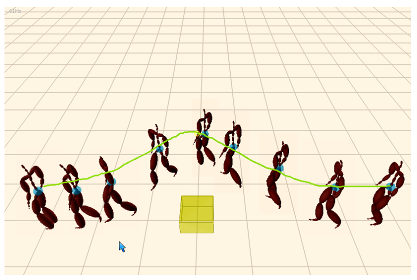
Every time we adjust the cut, we must update the MHTs along with the search queues. To do the update efficiently, we maintain a Fibonacci heap for each search tree respectively to order the nodes according to their distance to the search root in state space. Given a cut, this allows us to extract all nodes beyond the cut efficiently (the amortized running time is $O(m \log n)$, where m is the number of nodes beyond the cut, and n is the number of nodes in the tree). More specifically, we update the search queues and MHTs as follows: First, insert every pair of successive nodes crossing the new cut into the MHT; second, mark all nodes beyond these pairs as obsolete; third, insert all the other frontiers into the search queue if they are not already in it and if their costs are smaller than the current minimal total cost. We illustrate the updated status after cut adjustments in Figure 3.1c and Figure 3.1d.

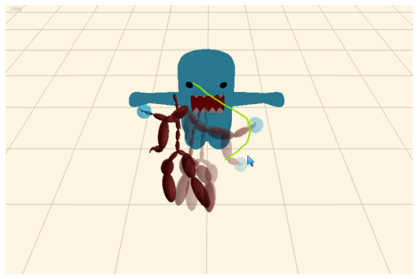
3.3 Intuitive Motion Control with Strokes

The basic idea of our system is to allow the user to specify a motion by sketching a desired trajectory for any controllable body part. Then we generate a motion whose trajectory of the specified body part best resembles the input stroke under the current viewing perspective. Our system is intuitive to use in that similar strokes may represent different motions under different viewing perspectives (Figure 3.2a and b), and different stroke trajectories can predictably generate different motions (Figure 3.2c and d) even with the same start and end conditions. These features are infeasible with gesture-based sketching systems [60, 79].

Searching a sequence of motion based on user sketches as described above is an under-constrained problem. There may be more than one, or even


(a) Jump


(b) Detour


(c) Slap

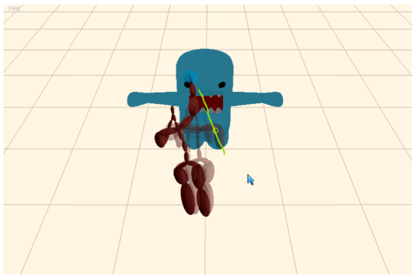

(d) Punch

Figure 3.2: Our system allows users to intuitively generate motions using a simple stroke interface. Similar strokes under different situations can lead to different motions.

many, motions that result in similar trajectories. Also, the user-input strokes may jiggle, and stray away from a natural trajectory. Our system must be able to handle these inputs robustly. To achieve this goal, we synthesize motion by searching a path on the motion graph that minimizes the difference in trajectories along with the energy required to perform the motion. We design a cost function for the search to represent the user intention while being robust to noisy input. Bidirectional A* search as described in Section 3.2 allows us to compute a near-optimal motion sequence in seconds on a moderate-sized motion graph.

3.3.1 Input Stroke Analysis

The input strokes convey the user's intentions for a motion in several ways. First, the shape of the stroke represents the desired motion trajectory under the current viewing perspective. Second, the user can draw strokes at

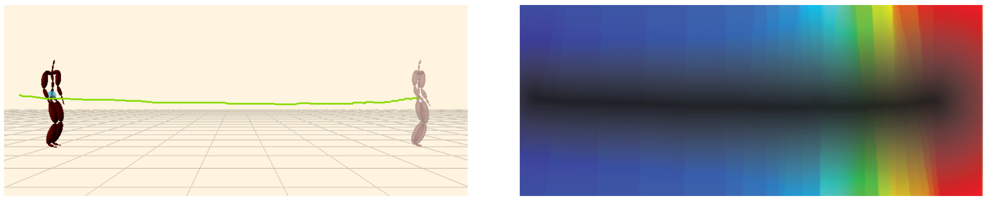

(a) Input stroke (b) Distance function

Figure 3.3: Visualization of the distance function and chronological flow of the input stroke: (a) A user input stroke and (b) its corresponding distance function (brightness) and chronological flow (hue).

different speeds to control the speed of the final motion. At last, the intersection of the stroke and the objects in the environment represents the character's contact with the environment. We next describe different cost functions that capture these constraints.

The **distance cost** measures the deviation of the synthesized motion from the trajectory specified by the input stroke. We use the technique proposed by Hoff et al. [21] to generate a *distance function* to the input stroke. The distance is measured in the image plane where the user draws the strokes. We sample the input stroke uniformly in time, and for each pixel c on the image plane we store its distance to the closest sample point. We denote the distance function by $D(c) : \mathbf{R}^2 \rightarrow \mathbf{R}$, which enables constant time look-up of distances. Along with the distance, we also use $T(c) : \mathbf{R}^2 \rightarrow \mathbf{R}$ to keep track of the identity of the closest sample point for each pixel on the image plane. $T(c)$ provides information about the sketching speed along with the chronological flow of the stroke. Visualization of $D(c)$ and $T(c)$ is shown in Figure 3.3.

We compute the distance cost for each new node in the search tree that we explore. To obtain a robust cost that is also efficient to compute, we evaluate the distance to the stroke only at a few key frames [3]. Let us denote the state of a parent node in the search tree by X' and the state of one of its children by X. To measure the distance error of X with respect to the stroke, we project the controlled body part to the image plane at each key frame. We denote the 3D positions at the key frames by $(p_1, p_2, \ldots, p_n)$ and their projected positions on image the plane by $(c_1, c_2, \ldots, c_n)$. Our distance cost is then

$$\Phi(X', X) = \sum_{i=1}^{n-1} \left(\|p_i - p_{i+1}\| \times \frac{D(c_i) + D(c_{i+1})}{2} \right), \tag{3.1}$$

which penalizes large deviation of the projected trajectory from the input stroke, weighted by the actual distance traveled in 3D. Since the stroke is defined on the 2D plane, there might be more than one sequence of motions fulfilling the constraint. We use $\|p_i - p_{i+1}\|$ to give preference to those having shorter paths in the 3D environment, which are usually more intuitive.

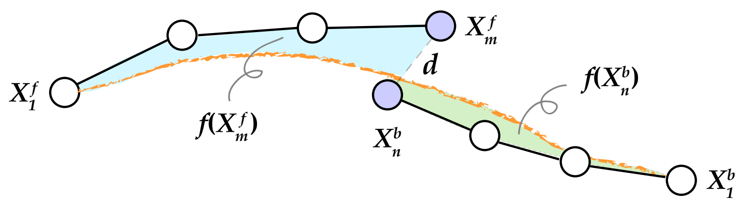
The **speed cost** attempts to adjust the speed of the synthesized motion to the speed of the user stroke. It also ensures that the character follows the stroke in a chronological order. In the preprocessing stage we calculate and store the moving speed along each link in the motion graph. The speed of a motion is simply the distance between the character root at the beginning and the end of the motion divided by the temporal length of the motion. At run time, we compute the speed cost of the transition from state X' to X as

$$\Psi(X', X) = \frac{\max(|T(c) - T(c')|, s)}{\min(|T(c) - T(c')|, s)} - 1, \tag{3.2}$$

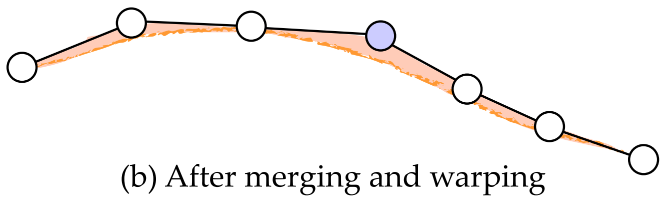
where s is the pre-computed speed associated with the link in the motion graph, and c and c' represent the projected position of the character's controlled body part on the image plane in state X and X' respectively. Note that we normalize the walking speed to a reference sketching speed to calibrate this cost. To ensure correct chronological order, $T(c)$ should be greater than $T(c')$ if they belong to the forward search tree and vice versa in the backward search tree.

The cost function also considers the smoothness of the transition between the poses a' and a, and the physical energy required to perform the in between motion. We adopt the point cloud metric [31] with weights on different joints [84] to compute the transition cost. The energy cost is the sum of the squared torques via inverse dynamics [68]. There may be more than one sequence of motions fulfilling the input stroke, but the one with the least energy usually looks more natural. Hence, the **complete cost** function is defined as

$$f(X') = f(X) + \gamma_\Phi \Phi(X, X') + \gamma_\Psi \Psi(X, X') + \gamma_\Delta \Delta(a, a'), \tag{3.3}$$



(a) Before merging



(b) After merging and warping

Figure 3.4: Merging the two partial solutions with motion warping. (a) A possible solution of bidirectional search. The two mergeable nodes are colored in purple. (b) We warp the whole sequence while fixing the start and end states so that the final pose can fulfill the constraint more precisely.

where $\Delta(a, a')$ denotes the sum of transition and energy cost, and γ_Φ, γ_Ψ, γ_Δ are scaling constants.

Finally, we provide **environmental constraints** that allow users to gain more control over the character motion. The users can specify contacts with the environment by pausing a few seconds when drawing. We then use ray tracing to find the intersection of the user stroke with objects in the environment. From the intersections we obtain contact information, such as the position and normal of the contact point, which serve as hard constraints in the search. This allows users to guide the character to interact with the environment, e.g. pick up, kick, or sit on an object.

3.3.2 Bidirectional A* Search

In this section, we provide more details about applying bidirectional A* search with the stroke interface. Our objective is to find two mergeable state sequences with minimal cost, $(X^f_1, X^f_2, \ldots, X^f_m)$ and $(X^b_n, X^b_{n-1}, \ldots, X^b_1)$, from forward and backward searches respectively. X^f_m and X^b_n are mergeable states, as illustrated in Figure 3.4a. We extend the state definition from Equation 2.4 as

$X = (a, x, z, \theta, t)$, where t indicates the chronological order of the current state along the user stroke. We find the value of t by projecting the controlled body part (pelvis for example) of the character onto the image plane and looking up the identity of the closest sample point on the input stroke using the T function.

The bidirectional search begins from the start state X_1^f and the end state X_1^b respectively. X_1^f is simply the character's current state. On the other hand, there may exist more than one candidate for the end state X_1^b. In this case, we create a virtual root node for the backward search tree and set all the candidates as its children. The candidates are derived from the user input. We first determine the end pose by intersecting the tail of the input stroke with objects in the environment. If there is an intersection, we use the contact information to extract all the possible final poses. Otherwise, we adopt all the resting poses without environmental contacts. Secondly, we estimate the end position by shooting a line from the current viewpoint to the tail of the stroke and matching it with the controlled body part in the poses. Finally, we allow the user to adjust the final orientation freely.

We define the cut by dividing state space. More specifically, we split the space along the t axis only, since it provides enough indication for splitting the search space. The initial cut is the hyperplane $(t_1^f + t_1^b)/2$. Note that t_1^f is the identity of the first sample point on the stroke (the minimal value of $T(\cdot)$), while t_1^b is that of the last (the maximal value of $T(\cdot)$). During the dynamical cut adjustment, we update the value of the cut as the median t value in the unexplored state space.

The mergeable states X_m^f and X_n^b in the result sequences may not match exactly because the motion space is discrete and the environment is quantized. If we simply concatenate the two sequences, the final state of the merged sequence will deviate from the user-specified state. Although the deviation is within the error tolerance, the result motion will be unnatural in some cases. For example in the case of grasping or kicking the character will act to the air. Therefore, we include a warping step [87, 18] so that the final states match more precisely. We illustrate the solution before and after warping in Figure 3.4a and Figure 3.4b respectively. To avoid foot sliding due to warping we apply analytical IK at run time to plant the feet [80, 40]. Warping should be reduced to

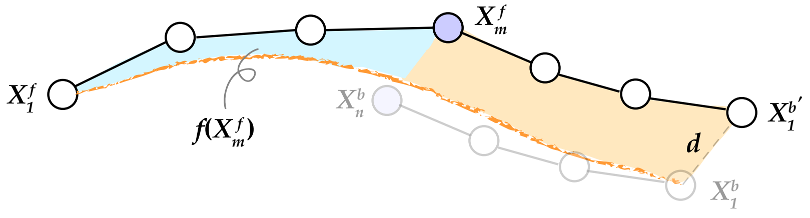


Figure 3.5: Direct merging result of the two partial paths in Figure 3.4a found with bidirectional search.

a minimum since it often degrades the motion quality. An advantage of near-optimal solutions is that they require less warping than motions with higher costs, which deviate further from user constraints. In Section 3.3.3, we show that we can control the optimality of the final solution by adjusting the size of quantization, ε.

The heuristic function in our search is used to estimate the cost of getting to the goal. Our heuristic function is a product of the shortest distance from the current position to the goal, multiplied by the minimum cost function value required to travel one unit distance. We run a number of tests to empirically compute this minimum value from the motion graph data, as suggested in Safonova and Hodgins's work [68].

3.3.3 Optimality Analysis

In this section, we clarify that, due to discretization of the motion space and quantization of the environment, the motions found by unidirectional and bidirectional A* search may not be exactly the same, but the difference is bounded.

Because of discretization, most solutions will not meet the goal exactly, but it is important that the motion should fulfill the end constraint precisely. Thus, to evaluate the cost of a solution, we warp its tail to the goal position. This warping step will change the cost evaluated during the search framework. The cost difference between the original and warped solutions, however, is bounded above by a linear function, $\sigma(\varepsilon, l)$, where l is the length of solution, and ε is a threshold used to discard solutions with large deviation. The reason why the bound depends on ε and l can be visualized intuitively as the difference of the shaded area in Figure 3.5 and Figure 3.4b.

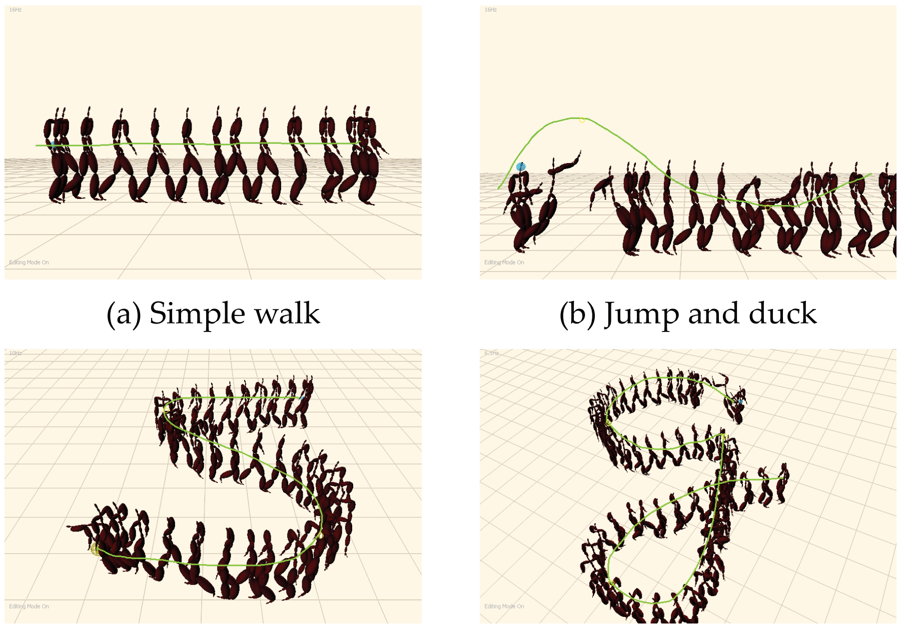
Now to compare a unidirectional solution with a bidirectional solution, we can split the unidirectional solution into two pieces and shift the second half so that the end goal is reached exactly. The cost difference of the split is also bounded by σ. Let C_u and C_b denote the cost of the optimal solution in unidirectional A* before and after splitting respectively, then $C_u - \sigma \leq C_b \leq C_u + \sigma$. The optimal solution in bidirectional A*, C'_b, is the optimal one over all paths with two segments divided by the cut, then $C'_b \leq C_u + \sigma$. Thus, the cost of the bidirectional solution C'_b is at most σ plus the cost of the optimal unidirectional A* solution C_u.

To conclude, the difference between the warped solution from A* search and that from bidirectional A* search is bounded by $O(\sigma)$. We can adjust the tolerable threshold ε to control the difference. When ε is negligible, the difference is unnoticeable.

3.4 Results

To produce our results, we construct a well-connected motion graph [93] from input motions. During the construction, we interpolated the motions in a close to physically correct way [67]. The resulting graph is larger than a standard motion graph since it includes many interpolated poses that do not belong to the original data to achieve better quality in the synthesized motion. We compress the graph by retaining only the nodes where contact changes happen[1], and the links among them [68]. We first use the technique from Lee and his colleagues [38] to identify contacts and then modify them manually. There may be more than one path connecting each pair of contact change nodes, and we use Dijkstra's shortest path algorithm to compute and retain the optimal one. The culling step does not affect the functionality of the graph as long as users are not allowed to control the details of the motion during a period of time when the contacts are not changing [68].

[1]A node is defined as a contact change node if its directly preceding node from the same input motion sequence has different contacts with the environment, e.g. changing from left foot stance to right foot stance.



(a) Simple walk (b) Jump and duck

(c) Walk and pick (d) Long walk

Figure 3.6: Motion sequences obtained from our algorithm (without the final warping step). The character is controlled at its pelvis (a), right hand (b), right hand (c), and pelvis (d) respectively. In (d), the character jumps and lifts his leg up high once because under this viewing perspective the input stroke only provides constraint on the moving path, and these actions happen to fulfill the trajectory better.

3.4.1 Performance

We generated a variety of examples on a Quad Core 2.4 GHz Intel processor to show the effectiveness of our approach. We construct the motion graph from a varied set of motion capture data, including walking with various turns, jumping, ducking, stepping over, cartwheeling, sitting, stepping onto, kicking, slapping, punching, picking, and pitching. The total length of original data is about 4 minutes long. The graph has 11436 nodes (including interpolated and original nodes) and 15471 links. We compress the graph into 84 nodes and 827 links, by keeping only the nodes where contact changes happen and links among them.

We set ε_x and ε_z to be about half of the length of a walking step, while setting ε_θ to 5.7 degrees. We compute the distance function and chronological flow for each input stroke by rendering distance meshes on the GPU. The computation takes less than 0.01 seconds, but the read back latency from GPU to CPU is about 0.1 seconds.

We compare the search performance of standard A*, bidirectional A*, and bidirectional A* with dynamic cut adjustment. In all the test cases presented in this section, we use the same graph, cost function, and heuristic function. The synthesized results and performance comparisons are shown in Figure 3.6 and Figure 3.7 respectively. Since the synthesized results of the three approaches are indistinguishable, we only show those of bidirectional A* with cut adjustment in Figure 3.6.

In the first test case ("simple walk") the user draws a straight line in a side view to guide the character to walk from left to right, as shown in Figure 3.6a. This case is simple and the two search trees are almost balanced, so adjusting the cut provides no further speed up. The second test case ("jump and duck") is harder because the constraint is abstract and strays away from the actual trajectory. A larger search space is required to better fulfill the constraints. Figure 3.7 shows that both versions of bidirectional A* outperform standard A*. Bidirectional A* without dynamic cut adjustment, however, does not fully utilize the advantage of search space splitting. Since the link of the jumping motion is longer than average, the forward search tree terminates much earlier, and dynamic cut adjustment can improve performance by moving the cut towards the other end. In the next test ("walk and pick"), shown in Figure 3.6c, the character is asked to detour in an S route to pick up a ball. Both bidirectional versions outperform standard A* search by a large margin. Our approach is more than seventy times faster than unidirectional search. The main reason is that this case is more favorable to backward search, so bidirectional A* with cut adjustment gains an advantage by moving the cut toward the start state several times. In the last experiment ("long walk") the character is asked to walk a long way before making a 180-degree turn in the end, as shown in Figure 3.6d. This example favors forward search a bit more than backward search. Without cut adjustment, the forward search waits while backward search expands ten times more nodes, so the performance is even slightly worse than A*. With our cut adjustment the cut is moved backward twice to balance the search, and we achieve interactive performance even for

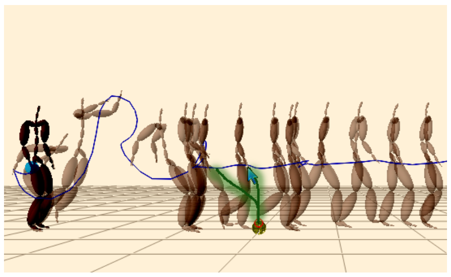
	A*	Bidirectional A*	Bidirectional A*+
Simple walk, 6 seconds			
time	0.031 s	0.016 s	0.016 s
#expanded nodes	3,175	78 + 65	78 + 65
speedup	1.00x	1.93x	**1.93x**
Jump & duck, 9 seconds			
time	4.781 s	3.25 s	1.781 s
#expanded nodes	879,771	1,506 + 413,356	68,219 + 210,757
speedup	1.00x	1.47x	**2.68x**
Walk & pick, 19 seconds			
time	14.188 s	0.375 s	0.187 s
#expanded nodes	2,841,462	37,238 + 9,918	10,122 + 9,918
speedup	1.00x	37.83x	**75.87x**
Long walk, 30 seconds			
time	3.25 s	3.344 s	0.875 s
#expanded nodes	567,542	68,995 + 507,839	69,658 + 67,223
speedup	1.00x	0.97x	**3.25x**

Figure 3.7: Performance comparison between standard A* search (*A**), bidirectional A* search without cut adjustment (*bidirectional A**), and our proposed bidirectional A* search with cut adjustment (*bidirectional A*+*). With each example we indicate the time it takes to execute the motion. We report the computation times to find a near-optimal motion, the number of nodes expanded in the forward and backward search trees, and the speedup.

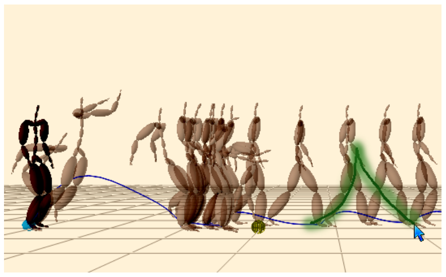
this motion that takes 30 seconds to execute.

3.4.2 Control

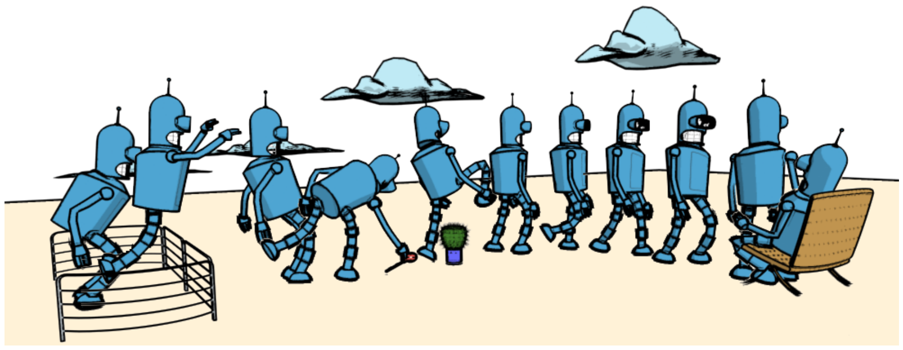
We present a sketching system for intuitive motion control and editing, with which even a novice user can specify a motion with simple strokes. The user can adjust the viewing perspective and select a body part on the character for sketching. Once a sketch is done, our system performs the bidirectional search to provide immediate visualization feedback. If the user is not fully satisfied with the result, the motion can be further refined with additional



Edit hand trajectory to grasp object



Edit foot trajectory to step over



Final motion

Figure 3.8: An editing session with our sketch interface. The user first edits the trajectory of the character's hand to pick up an object. The original trajectory is marked in blue, and the user edit in green. Our bidirectional search algorithm provides immediate feedback to visualize the edited motion. The user next selects the character's foot and edits its trajectory to step over an obstacle. Our system synthesizes the final motion, shown in the bottom, satisfying all the constraints in a near-optimal fashion.

strokes. An editing session is demonstrated in Figure 3.8. We also show in Figure 3.9 that our synthesized motion can well match the input stroke.

3.4.3 Scalability

Our bidirectional strategy can be applied on many graph-like structures to improve the performance of motion synthesis, e.g. state machines, motion graphs, or even interpolated motion graphs [68]. Also, the bidirectional strategy can be applied on other A* variants (truncated A*, inflated A*, or anytime A*), which would promise further performance improvement at the cost of reduced quality of the solution.

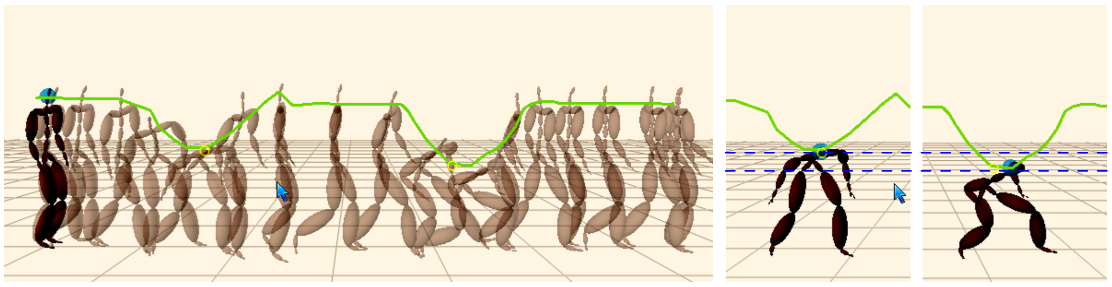


Figure 3.9: The user specifies a motion by sketching the trajectory of the character's head. Note the depthes of the two concavities in the strokes are different, and our system is able to match them with two different ducking motion.

In this section, we show how the performance of different search algorithms scales with the size of the graph. We ran experiments on an interpolated motion graph created from a well-connected motion graph, which includes walking with various turns and has 45 abstract nodes and 230 abstract links (after compression). The interpolated graph has 1493 abstract nodes and 24882 abstract links.

First, we compare unidirectional and bidirectional search (with dynamic cut adjustment) using a motion graph and an interpolated motion graph on the same input. The synthesized motions and performance results are shown in Figure 3.10 and Figure 3.11 respectively. With the interpolated motion graph, the synthesized motion can fulfill the constraint more precisely, but the running time increases accordingly. With depth reduction, however, our bidirectional search suffers less from the increased graph size and achieves a speedup of more than 10. We also implemented inflated A* by multiplying the estimated cost with an error tolerance $\delta > 1$, so that the cost of the solution is bounded from above by δ times the cost of an optimal solution. From Figure 3.11, we can see that even with $\delta = 10$, unidirectional search still requires 5 seconds, which is 3.7 times slower than our bidirectional search with no inflation ($\delta = 1$, 1.36 seconds). This shows that our algorithm is able to find a better solution in a considerably shorter amount of time (the quality of the synthesized motions can be compared in Figure 3.10).

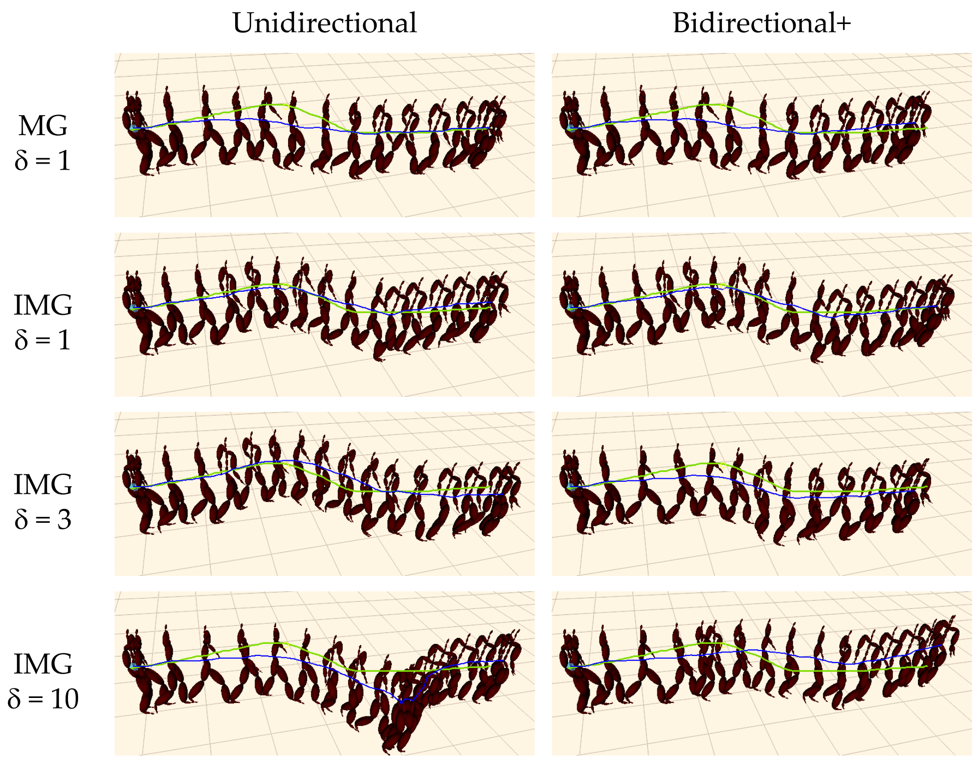


Figure 3.10: Synthesized results from unidirectional and bidirectional search. The result motions are about 7 second long. The green line indicates the input stroke, while the blue line represents the trajectory of the synthesized motion.

Secondly, we make the same comparison as above but with a longer and more complicated input. The synthesized results and performance comparisons are shown in Figure 3.12 and Figure 3.13 respectively. In this case, without inflation, both unidirectional and bidirectional search cannot find a solution within two minutes, and more than ten million search nodes are expanded. Setting $\delta = 2$, we are able to find a solution with bidirectional inflated A*, but fail again with standard inflated A*. With standard inflated A*, we need to set a much larger tolerance $\delta = 10$ to find a solution in a comparable amount of time. Since the cost of the solution is only guaranteed to be bound within 10 times the optimal cost, its quality is reduced significantly as shown in Figure 3.12.

Although we demonstrated the bidirectional strategy only on basic A* and inflated A*, we can apply it to other variants too. We would expect sim-

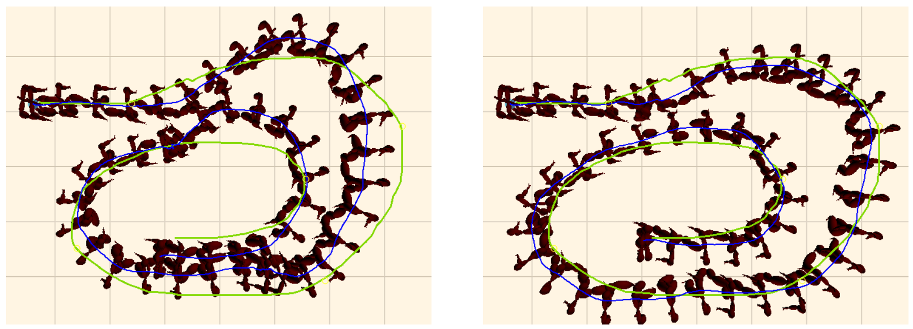
	Unidirectional A*	Bidirectional A*+
Motion Graph, δ = 1		
time	0.297 s	0.078 s
#expanded nodes	47,821	4,786 + 6,310
speedup	1.0x	**3.80x**
Interpolated Motion Graph, δ = 1		
time	14.171 s	1.36 s
#expanded nodes	2,450,768	78,278 + 83,434
speedup	1.0x	**10.41x**
Interpolated Motion Graph, δ = 3		
time	10.547 s	0.078 s
#expanded nodes	1,235,655	1,447 + 3,119
speedup	1.0x	**135.21x**
Interpolated Motion Graph, δ = 10		
time	5.094 s	0.063 s
#expanded nodes	625,766	1,554 + 1,274
speedup	1.0x	**80.85x**

Figure 3.11: Performance comparison between unidirectional and bidirectional search. We applied both search algorithms on a motion graph and an interpolated motion graph with the same input. We also made comparisons with both standard A* ($\delta = 1$) and inflated A* ($\delta > 1$).

ilar results for example for anytime A*, which is a variant of inflated A* that dynamically adjusts the optimality bound δ at run time. We can also adopt the approach of pre-expanding A* search trees [36] to further enhance the search performance.

3.5 Conclusions

We present an algorithm to improve the search efficiency for near-optimal motion synthesis using motion graphs, and demonstrated its application to interactive motion synthesis using an intuitive sketching interface. The main idea of our algorithm is to use a bidirectional search strategy. The benefit of this ap-



Unidirectional, δ = 10 Bidirectional+, δ = 2

Figure 3.12: Synthesized results from unidirectional and bidirectional inflated A*. Note the deviation from the synthesized motion (blue line) to the input (green line) in the right image due to large inflation. Bidirectional search produces a higher quality result ($\delta = 2$) in a shorter amount of time (6.6 sec.) than unidirectional search ($\delta = 10, 10.18$ sec).

search type	time	#expanded nodes
Unidirectional A*, $\delta = 2$	$>$ 2 minutes	$>$ 10 million
Bidirectional A*+, $\delta = 2$	6.60 seconds	354,641 + 459,431
Unidirectional A*, $\delta = 10$	10.18 seconds	2,385,957

Figure 3.13: Performance comparison between unidirectional and bidirectional search with interpolated motion graph using a longer and more complicated input.

proach is that it can reduce the maximum search depth by almost a factor of two. We demonstrate that this leads to significant performance improvements. To fully exploit the potential of bidirectional search, we propose to dynamically adjust a cut that separates the two search trees, and we use efficient data structures to limit the overhead required to merge them. We showed that in some cases, the bidirectional search outperforms unidirectional search by an order of magnitude.

We see the following limitations and opportunities for extensions that we plan to address in future work: Although our approach is more efficient than unidirectional search, the length of motions that we can generate at interactive rates is still limited. This could be addressed with a hierarchical approach. Another alternative is to use A* variants. Both approaches are orthogonal to the bidirectional search strategy and could be plugged into our framework easily. A common issue in motion synthesis governed by a cost function is to adjust the parameters of the cost to obtain intuitive results. This often requires experimentation. It would be useful to have a more robust and automatic way to determine appropriate parameters.

Moreover, although in this work we focus on the problem of motion synthesis, the bidirectional framework has the potential of being applied on other problems that can be solved with conventional search algorithms. The main issue, however, is to find a mapping from the search space or state space to the Euclidean space, since we define a cut and perform the cut adjustment directly in the Euclidean space. Once the mapping is obtained, our bidirectional framework can be applied to speed up the search performance.

This chapter is based on "Bidirectional Search for Interactive Motion Synthesis", Wan-Yen Lo and Matthias Zwicker, *Computer Graphics Forum (Proceedings of Eurographics EG'10)*, 2010.

Chapter 4
Reinforcement Learning

"To judge what one must do to obtain a good or avoid an evil, it is necessary to consider not only the good and the evil in itself, but also the probability that it happens or does not happen; and to view geometrically the proportion that all of these things have together."

Port-Royal Logic
ANTOINE ARNAULD

The problem of interactive motion control by assembling motion fragments is similar to *sequential decision problems* in AI. In a sequential decision problem, an agent needs to make a sequence of decisions to maximize the total reward, while in an interactive application of motion control, the program needs to compose a sequence of motion fragments that best fulfils user control and environment constrains. In this chapter, we describe how the optimal behavior can be learned in a pre-process, so that optimal or near-optimal decisions can be made instantaneously in real-time. Such guarantee on both real-time performance and (near) optimality is hard to achieve with the search strategies described in Chapter 2 and Chapter 3. We first explain in Section 4.1 how to formulate the sequential decision problems using Markov decision processes and how to solve the problems to produce optimal behavior. We next introduces reinforcement learning in Section 4.2, which does not require a complete model of the environment, but instead lets the agent interact with the environment and uses the observed rewards to learn an optimal policy. Hence, reinforcement learning allows us to specify only the high level goals of the problem, e.g. move to the destination or win the game, without explicitly describing the possible outcomes of each action. Finally, in Section 4.3 we provide details of building a motion controller with reinforcement learning and review relevant work in character animation.

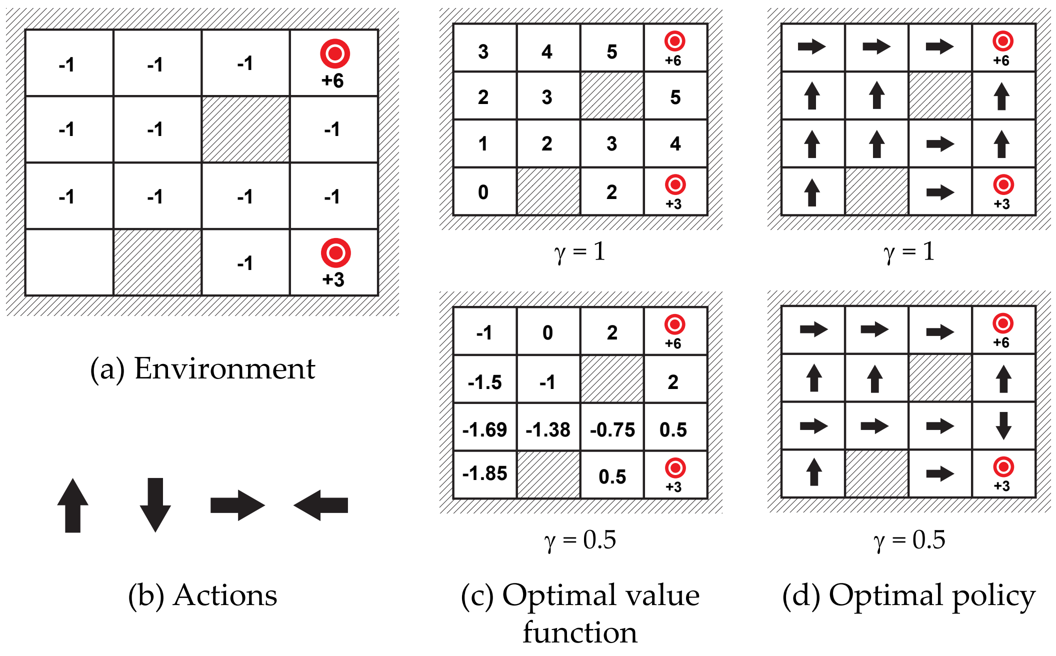
4.1 Markov Decision Process

A *Markov decision process* (MDP) is used to specify sequential decision problems, and is defined as $(\mathcal{S}, \mathcal{A}, T, R)$, where

- $\mathcal{S}$ is the **state space**.
- $\mathcal{A}$ is the **action space**.
- T is the **transition model**, where $T(s, a, s')$ denotes the probability of reaching state s' if action a is taken in state s. The transitions are *Markovian*, meaning the probability of reaching s' from s depends only on s and not on the history of earlier states. If the probability is either 0 or 1 in the transition model, the MDP is *deterministic*, that is, taking action a in state s always leads to the same state s'. On the contrary, if any state-action pair may lead to more than one possible state, the MDP is *stochastic*.
- R is the **reward function**, and $R(s, a, s')$ returns the reward of taking action a in state s and moving to state s'. The reward can be positive or negative, but must be bounded. The definition of the reward function can also be simplified as $R(s)$ to only depend on the current state s. This simplification does not change the problem in any fundamental way.

Let us take the navigation problem as an example to show how it can be modeled with MDP. Suppose the agent is situated in the environment shown in Figure 4.1a, and in each time step, the agent can move in one of the four directions shown in Figure 4.1b. The state space consists of all possible locations in the environment, and the action space consists four movements. To simplify the problem, we assume each action always achieves the intended effect, except that colliding with a wall results in no movement. In addition, the agent stops moving after reaching one of the two target states. Therefore, $T(s, a, s')$ returns one only when s is not a target state and one of the following conditions holds:

1. $s' \neq s$ and s' is in the direction a of s.



(a) Environment (b) Actions (c) Optimal value function (d) Optimal policy

Figure 4.1: A simple navigation problem. The state of an agent is its position in the environment (a), and the agent can move in four different directions (b). The two target states have reward +6 and +3 respectively, and all other states have a reward of -1. A collision with a wall results in no movement. Two different discount factors are used to learn the optimal value functions (c) and optimal policies (d). A small discount factor results in a near-sighted behavior, with which the agent does not plan far about the future and always move toward the closer target even though the reward is smaller.

2. $s' = s$ and there is a wall in the direction a of s.

The MDP is thus deterministic. The reward function is used to specify the agent's behavior. In this example, we would like the agent to go to the top-right corner g_1 or the bottom-right corner g_2 but with stronger preference on the top-right corner. The rewards are then set as $R(g_1) > R(g_2) > 0$. Also, $R(s) = -1$ for all other states, giving the agent an incentive to reach the targets quickly.

A solution to the problem modeled with MDP is called a *policy*, denoted by π. A policy is a deterministic mapping from the states to the actions, and given a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$, the agent will always take action $\pi(s)$ in state s. The total number of all possible policies is thus $|\mathcal{A}|^{|\mathcal{S}|}$, assuming the state

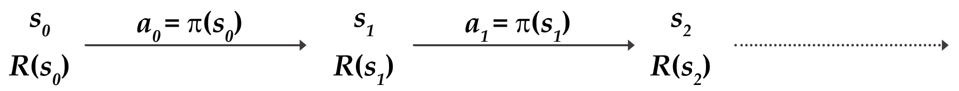


Figure 4.2: At each discrete time t, the agent is in state s_t, and by taking action $a_t = \pi(s_t)$, the agent reaches state s_{t+1} with reward $R(s_{t+1})$.

and action space are discrete. In a discrete-time system, the agent takes action $\pi(s_t)$ at time t and reaches state s_{t+1} receiving reward $R(s_{t+1})$ [1], as illustrated in Figure 4.2. The quality of a policy π is measured with the accumulated rewards in the long term,

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots = \sum_{t=0}^{\infty} \gamma^t R(s_t),$$

where $\gamma \in [0, 1)$ is a *discount factor*, which accounts for future uncertainty and gives more weight to the near future than the distant future. This factor can be used to specify how far into future the agent should plan for. If $\gamma = 0$, only immediate reward matters, and the agent will act in a *greedy* fashion. If $\gamma \ll 1$, the agent will be near-sighted, otherwise if $1 - \gamma \ll 1$, the agent will be far-sighted. Figure 4.1d compares the effects of two different discount factors: when the distant future is as important as the near future, the agent will risk short-term benefits to achieve bigger objectives in the long run; as the distant future is weighted less, the agent becomes more opportunistic, and will aim for closer rewards.

If the MDP is stochastic, executing a policy from the same initial state will lead to different state sequences. Hence, to evaluate the quality of a policy, we need to take into account the stochastic nature of the environment by computing the *expected* accumulated long-term rewards in the long term. The *value function* $V^\pi : \mathcal{S} \to \mathbb{R}$ is thus defined as,

$$V^\pi(s) = E^\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \middle| s_0 = s \right] \tag{4.1}$$

$$= R(s) + \gamma \sum_{s'} T\left(s, \pi(s), s'\right) V^\pi(s'), \tag{4.2}$$

[1]The complete definition of the reward function, $R(s_t, a_t, s_{t+1})$, depends also on the action and outcome. Here we use the simplified definition, so that the reward only depends on the current state.

which returns the expected long-term reward from an initial state s following the policy π. Equation 4.2 is called the *Bellman equation* [6], stating the recursion relation of the value function. It can be proved that there is always at least one optimal policy π^* that yields the optimal value V^* for every state [77], that is, $V^*(s) \geq V^{\pi}(s)$ for all policy π and state s. Although there may exist more than one optimal policies, there is only one optimal value function V^*, which is the unique solution to the *Bellman optimality equation*,

$$V^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s')V^*(s'), \tag{4.3}$$

named after Richard Bellman [6].

A similar notation to the value function is the *action-value function* $Q^{\pi} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, which returns the expected long-term reward from an initial state s taking action a and then following the policy π,

$$\begin{aligned} Q^{\pi}(s,a) &= E^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \middle| s_0 = s, a_0 = a\right] && (4.4) \\ &= R(s) + \gamma \sum_{s'} T(s,a,s')V^{\pi}(s'). && (4.5) \end{aligned}$$

The optimal action-value function is denoted by Q^*. An optimal policy can be obtained with the optimal action-value function or with the optimal value function,

$$\begin{aligned} \pi^*(s) &= \operatorname*{argmax}_a Q^*(s,a) && (4.6) \\ &= \operatorname*{argmax}_a \left[R(s) + \gamma \sum_{s'} T(s,a,s')V^*(s')\right] \\ &= \operatorname*{argmax}_a \left[\sum_{s'} T(s,a,s')V^*(s')\right] && (4.7) \end{aligned}$$

Example optimal value functions along with the optimal policies are shown respectively in Figure 4.1c and Figure 4.1d. Next, we describe two algorithms for finding optimal policies.

4.1.1 Policy Iteration

To find the optimal policy, the *policy iteration* algorithm initializes a policy π_0 at random, and then alternates the following two steps to improve the policy until convergence:

- **Policy evaluation**: Given a policy π_i, calculate the value function V^{π_i} for every possible state.
- **Policy improvement**: Use V^{π_i} to calculate the new policy π_{i+1}, which performs better or equal to the old policy π_i.

The algorithm terminates when the policy improvement step yields no more change in the value function.

We first describe how to implement the policy evaluation step. Given a policy π, we can re-arrange Equation 4.2,

$$V^{\pi}(s) - \gamma \sum_{s'} T\left(s, \pi(s), s'\right) V^{\pi}(s') = R(s), \forall s \in \mathcal{S}, \tag{4.8}$$

where $V^{\pi}(s)$ is the only unknown, and by factoring out $V^{\pi}(s)$, we can form a system of $|\mathcal{S}|$ linear equations with $|\mathcal{S}|$ unknowns:

$$\left[I_{|\mathcal{S}|\times|\mathcal{S}|} - \gamma T\right] \mathbf{v} = \mathbf{r}, \tag{4.9}$$

where I is an identity matrix, T is a transition matrix built from the transition model, $\mathbf{v}$ is an unknown vector of the value function, and $\mathbf{r}$ is a vector of the rewards. Hence, the value function can be solved as,

$$\mathbf{v} = \left[I_{|\mathcal{S}|\times|\mathcal{S}|} - \gamma T\right]^{-1} \mathbf{r}. \tag{4.10}$$

The time complexity of this exact solution method is $O(|\mathcal{S}|^3)$, dominated by the matrix inversion. For small state spaces, this is often the most efficient approach to evaluate the value function, but for large state spaces, $O(|\mathcal{S}|^3)$ computing time might be prohibitive. Therefore, in the case of large state spaces, an alternative is preferred, which iteratively performs the following update,

$$V^{\pi}(s) \leftarrow R(s) + \gamma \sum_{s'} T\left(s, \pi(s), s'\right) V^{\pi}(s'). \tag{4.11}$$

This is a simplified version of value iteration (Section 4.1.2) and provides a reasonable approximate of the value function.

Secondly, the policy improvement step is implemented by defining the new policy as,

$$\pi_{i+1}(s) = \operatorname*{argmax}_{a} Q^{\pi_i}(s, a). \tag{4.12}$$

We can then derive the following inequality,

$$V^{\pi_i}(s) = Q^{\pi_i}\left(s, \pi_i(s)\right) \leq \max_a Q^{\pi_i}(s,a) = Q^{\pi_i}\left(s, \pi_{i+1}(s)\right), \tag{4.13}$$

which can be expanded with Equation 4.5 as,

$$V^{\pi_i}(s) \leq Q^{\pi_i}\left(s, \pi_{i+1}(s)\right) = R(s) + \gamma \sum_{s'} T\left(s, \pi_{i+1}(s), s'\right) V^{\pi_i}(s'). \tag{4.14}$$

Hence, it is better to take the first step under π_{i+1} than to always follow π_i. The inequality can be expanded one time step further as,

$$V^{\pi_i}(s) \leq R(s) + \gamma \sum_{s'} T\left(s, \pi_{i+1}(s), s'\right) \left[R(s') + \gamma \sum_{s''} T\left(s', \pi_{i+1}(s'), s''\right) V^{\pi_i}(s'') \right].$$

We can conclude again that it is better to take the first two steps under π_{i+1} than to always follow π_i. Similarly, if we expand the inequality t time steps, we can conclude that it is better to take the first t steps under π_{i+1} than to always follow π_i. Let $t \to \infty$, it is better to always follow π_{i+1} than π_i. As a consequence, π_{i+1} is proved to be better or equally good as π_i.

Since there are only finitely many policies for finite state and action spaces, the policy cannot be infinitely improved. At some point, the policy from the previous iteration π_i must be as good as the new one π_{i+1}. This happens when the equality in Equation 4.13 holds,

$$V^{\pi_i}(s) = \max_a Q^{\pi_i}(s,a),$$

which can be further expanded with Equation 4.5,

$$V^{\pi_i}(s) = R(s) + \gamma \max_a \sum_{s'} T\left(s, a, s'\right) V^{\pi_i}(s').$$

Satisfying the Bellman optimality equation (Equation 4.3), V^{π_i} is thus optimal. We can then conclude that the algorithm must terminate and π_i will converge to the optimal policy π^*.

4.1.2 Value Iteration

In the previous section, we show how policy iteration iteratively improves an initial policy until it converges to an optimal policy. One drawback of policy iteration, however, is that each iteration of the algorithm involves policy evaluation, which itself may also be computed iteratively (see Equation 4.11). In this section, we describe an alternative, *value iteration*, which directly compute the optimal value function with an interactive algorithm, and in the end uses the optimal value function to define the optimal policy with Equation 4.7.

The optimal value function is the solution to the Bellman optimality equation (Equation 4.3). However, unlike the Bellman equation (Equation 4.2), which defines a system of linear equations and can be solved directly with linear algebra techniques (Equation 4.9), the Bellman optimality equation defines a system of nonlinear equations. Value iteration provides an iterative approach to solve the problem. The algorithm starts by initializing a value function V_0 to be zero everywhere and iteratively update the value function until equilibrium. The iterative step is called *Bellman update*,

$$V_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s,a,s') V_i(s'), \forall s \in \mathcal{S}, \tag{4.15}$$

which use the right-hand side of the Bellman optimality equation to update the value function.

If the Bellman update is applied infinitely often, V_i is guaranteed to converge to the solution of the Bellman equation, that is, the optimal value function. Let Δ_i denote the approximation error at the ith iteration,

$$\Delta_i = \max_s |V_i(s) - V^*(s)|, \tag{4.16}$$

then with the Bellman update we have

$$\begin{aligned}
\Delta_{i+1} &= \max_s \Big| V_{i+1}(s) - V^*(s) \Big| \\
&= \max_s \Big| R(s) + \gamma \max_a \sum_{s'} T(s,a,s') V_i(s') - V^*(s) \Big| \\
&= \max_s \Big| R(s) + \gamma \max_a \sum_{s'} T(s,a,s') V_i(s') - R(s) - \gamma \max_a \sum_{s'} T(s,a,s') V^*(s') \Big| \\
&= \gamma \max_s \Big| \max_a \sum_{s'} T(s,a,s') V_i(s') - \max_a \sum_{s'} T(s,a,s') V^*(s') \Big|.
\end{aligned}$$

With the follwing lemma,

$$|\max_a f(a) - \max_a g(a)| \leq \max_a |f(a) - g(a)|, \tag{4.17}$$

we further obtain the inequality

$$\begin{aligned}\Delta_{i+1} &\leq \gamma \max_s \max_a \left| \sum_{s'} T(s,a,s') \left(V_i(s') - V^*(s') \right) \right| \\ &\leq \gamma \max_{s'} \left| V_i(s') - V^*(s') \right| \\ &= \gamma \Delta_i,\end{aligned}$$

which implies

$$\Delta_i \leq \gamma^i \Delta_0. \tag{4.18}$$

Suppose the reward function is bounded by B_r, that is, $\max_s |R(s)| < B_r$, Δ_0 is bounded by

$$\Delta_0 = \max_s |V_0(s) - V^*(s)| = \max_s |V^*(s)| \leq B_r \left(1 + \gamma_+ \gamma^2 + \ldots\right) = \frac{B_r}{1-\gamma}.$$

Finally, we derive the error bound of the approximated value function at the ith iteration,

$$\Delta_i \leq \frac{\gamma^i B_r}{1-\gamma}, \tag{4.19}$$

proving the convergence property of the value iteration algorithm, since $\Delta_i \to 0$ when $i \to \infty$. The convergence rate of the value iteration algorithm depends on the discount factor γ, and more iterations are requires when the value of γ increases. In addition, with large MDP or limited computing resources, a near-optimal policy may be preferred, and we can use Equation 4.19 to compute the minimal number of iterations required to reach an error bound on the sub-optimality.

4.2 Reinforcement Learning

In Section 4.1, we have seen how a sequential decision problem could be formulated with a MDP, and how an optimal policy could be solved with the policy iteration or value iteration algorithm. However, in many complex domains, it is difficult a obtain a prior knowledge of the complete transition model and the reward function. For example, when training an agent to navigate a complex environment with a large set of rules, it is hard to enumerate all the possible outcomes of taking any action in any situation. In this chapter,

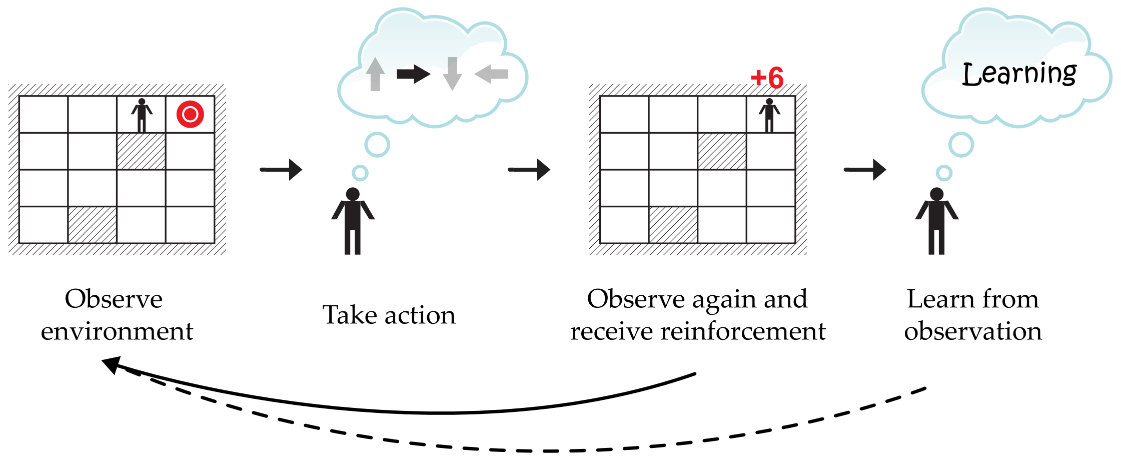


Figure 4.3: Reinforcement learning allows an agent to learn from interacting with the environment. By taking an action, the agent observes the change of its states in the environment and receives back a reinforcement, which could be rewards or penalties. The agent can explore the environment constantly and use the observations all at once to learn an optimal behavior, or the agent can alternate between learning and exploration, so that the learned policy can be used to make better exploration.

we describe how the problems could be solved when the transition model and the reward function are unknown. This learning framework allows the training to be performed at high levels. It works by placing the agent in the environment and observing the agent's interactions with the environment. The agent is never taught how to behave but is only informed when the task is fulfilled or when a rule is broken.

Reinforcement learning (RL) is biologically inspired and use observed rewards or penalties to learn an optimal or near-optimal policy for the environment [27, 77]. Unlike supervised learning, RL requires no explicit teacher to tell the agent what action to take in each circumstance. Instead, the agent receives *reinforcement* signals while interacting with the environment. The reinforcement can be either positive or negative so that the agent learns the good or bad consequence of taking a specific action in a specific situation. Such reinforcements can also be delayed from the actions which are responsible for them, allowing an agent to learn long-term consequences. Similarly, from a biological perspective, when an animal bumps into a wall, it receives a sensation of pain as a negative reinforcement and will eventually learn to keep a distance away from the wall.

The reinforcement learning framework consists of two step: gathering observations from the environment and learning from the observations. We can either gather all the required observations at once for learning a control policy, or iterate between the two steps so that in each iteration the updated policy is used to gather more informative observations, as shown in Figure 4.3. There are two types of learning algorithms: model-based and the model-free approaches. The model-based algorithms use the observations to explicitly build a MDP model and learn an optimal policy with the MDP (using the value iteration or policy iteration algorithms for example). On the contrary, the model-free approaches learn the value function or action-value function directly without building any MDP model. We will first describe how to gather observations from the environment in Section 4.2.1, and introduce a model-based and a model-free learning algorithm in Section 4.2.2 and Section 4.2.3 respectively.

4.2.1 Exploration

To compensate the absence of an explicitly defined transition model and reward function, reinforcement learning algorithms require the agent to explore the environment extensively so that sufficient information can be gathered to learn an optimal policy. Two strategies are commonly used for the agent to explore the environment: *random exploration* and *greedy exploration*. With the first strategy, the agent explore the environments with random actions, and the algorithms will eventually converge if every action is taken an infinite number of times in every state. The performance, however, can be extremely poor. In order to put the gained knowledge in use to make less exploration necessary, the second strategy requires the agent to follow the recommendation of the currently learned policy. By greedily taking actions with the current policy, however, the agent risks getting stuck with the same (suboptimal) action when entering the same state. The agent might just linger around the small rewards discovered, even though bigger rewards exist in the undiscovered region. Hence, although greedy exploration can lead to shorter learning time, the learning algorithms are not ensured to get sufficient information to converge to an optimal solution.

This is a fundamental trade-off in reinforcement learning, between *exploitation* of learned behaviors with high reward and *exploration* in undiscovered region of state-action space. An compromise approach is *ε-greedy exploration*, where the agent chooses a random action with probability ε and follow the currently learned policy with probability $1 - \varepsilon$. Ernst et al. [14] show that with limited observations, learning with the ε-greedy exploration results in a better policy than learning with the random exploration, especially in the problems where the goal is hard to reach by simply taking random actions. The ε-greedy exploration method is also used in our work (Chapter 5 and Chapter 6).

Russell and Norvig [66] suggest another exploration strategy, which gives some weights to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low value. This is implemented on top of the greedy exploration by assigning to relatively unexplored state-action pairs an optimistic value estimate of the best possible reward obtainable in any state. Such value estimate has the effect of making the agent behave initially in an very optimistic way, as if there are rewards scattered all over the place. On the contrary, with pessimistic value estimate, the agent would soon disinclined to go beyond the explored regions. Finally, they show that this exploration strategy leads to a rapid convergence toward optimal performance.

4.2.2 Model-based Learning

The model-based reinforcement learning algorithms work by reconstructing the MDP explicitly and solving the reconstructed MDP with a dynamic programming method (e.g. value iteration, policy iteration). In this section, we introduce a simple model-based learning algorithm.

In order to reconstruct the MDP model from the observations, we can keep track of the outcome and frequency of every transition for estimating the reward function and the transition model. More specifically, after each step of the exploration, we observe a transition, (s, a, s', r), which means by taking action a in state s the agent enter state s' with reward r, and we use the new observation to update the MDP:

$$\begin{aligned}
N(s,a) &\leftarrow N(s,a)+1 \\
N(s,a,s') &\leftarrow N(s,a,s')+1 \\
R(s') &\leftarrow r \\
T(s,a,t) &\leftarrow N(s,a,t)/N(s,a), \forall t \in \mathcal{S}.
\end{aligned}$$

$N(s,a)$ represents the number of times the action a has been taken when the agent is in state s; similarly, $N(s,a,s')$ represents the transition frequency. With these update rules, the estimate of the MDP will converge to the right one eventually if each action is taken an infinite number of times in each state.

Each time the MDP is updated, it can be solved with a dynamic programming method to get a new policy, which can be used by the agent to further explore the environment. Solving for a new policy after each update, however, can lead to high computational overhead, since the MDP needs to be solved completely even if it is modified only slightly. In addition, this model-based approach requires non-linear memory usage to keep track the transition frequencies, which is intractable for large state spaces. Backgammon, for example, has about 10^{50} states. Nevertheless, model-based learning algorithms have an advantage that the reconstructed MDP is useful for task transfer where the reward function or discounted factor changes but system dynamics remains the same, e.g. robot navigation with different goals in the same environment.

4.2.3 Model-free Learning

Contrary to the model-based approaches, model-free learning algorithms compute an optimal policy directly without reconstructing the transition model and the reward function. In this section, we introduce a representative model-free learning algorithm, *Q-learning*, which uses the observations to incrementally update the action-value function. Before describing the Q-learning algorithm, we first explain how to evaluate an existing policy without knowing the corresponding MDP.

Temporal difference. Given a policy π, the *temporal difference* (TD) algorithm allows us to directly compute V^π without explicitly reconstructing the MDP.

The algorithm requires the agent to explore the environment with policy π and uses the observations to incrementally update the values of the observed states. More specifically, for each observed transition (s, a, s', r), the following update is applied,

$$V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\delta(s), \tag{4.20}$$

where α denotes the *learning rate*, and $\delta(s)$ represents the temporal difference between a new value estimate from the observed transition, $r + \gamma V^{\pi}(s')$, and the current value estimate $V^{\pi}(s)$:

$$\delta(s) = r + \gamma V^{\pi}(s') - V^{\pi}(s).$$

The temporal difference indicates whether the current estimate should be increased or decreased in order to move toward the equilibrium of the Bellman equation (Equation 4.2). In general, TD learning is a combination of Monte Carlo ideas and dynamic programming ideas [77]. TD is related to dynamic programming techniques because the value function is updated with the previously learned estimate. TD is related to Monte Carlo sampling because it estimates the value function stochastically by sampling the environment with the fixed policy π.

Assume each state is visited infinitely often under policy π, then the TD learning will converge if the following conditions hold:

- Each state has its own learning rate $\alpha_i(s)$, where i denotes the ith visit to the state s.
- The learning rate α satisfies $\sum_{i=1}^{\infty} \alpha_i(s) = \infty$ and $\sum_{i=1}^{\infty} \alpha_i^2(s) < \infty$ for all state $s \in \mathcal{S}$.

Intuitively speaking, the learning rate α_i should decay sufficiently slowly with respect to i to incorporate a large number of observed transitions, but α_i should also decay sufficiently fast to allow convergence.

Q-learning. Similarly, the Q-learning algorithm [85] learn the optimal action-value function Q^*, by incrementally updating the Q-function with the temporal difference between the new observation and the current estimate. The agent is not required to follow any particular policy but can use any exploration strategy. The following update rule is applied when a new observation is made:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]. \tag{4.21}$$

If each state-action pair is visited infinitely, and a decaying $\alpha_i(s,a)$ is used for each state-action pair, the algorithm converges to the equilibrium of

$$Q(s,a) = R(s) + \gamma \sum_{s'} \left[T(s,a,s') \left(\max_{a'} Q(s',a') \right) \right], \tag{4.22}$$

which can be obtained by applying Equation 4.5 on an optimal policy. If an action-value function satisfies Equation 4.22, it is optimal. Once the optimal action-value function is learned, Equation 4.6 is used to derive an optimal policy.

4.3 Learning Motion Controllers

In this section, we describe a reinforcement learning framework to obtain optimal or near-optimal motion controllers for interactive character animation. Given a corpus of motion fragments, the *motion controller* is responsible for making a sequence of decisions to generate a motion that achieves user goals. The controller makes decisions in real-time, and only decides the next motion fragment when the currently played fragment completes. Making a good decision is important, as some user objectives require planning ahead of time. For example, if the character is required to leap over a long gap, the controller should prepare the character to run several steps ahead. Otherwise, if the decision is made too late, a sudden transition from 'walk' to 'jump' might appear too abrupt, or there might even exist no transition from 'walk' to 'jump' (only from 'run' to 'jump'), and in this case the motion fails to satisfy the constraint. Each decision, however, can only be made within a short amount of time, since time lags are not allowed in interactive applications.

Reinforcement learning is a promising approach to address these issues. With reinforcement learning, the controller can be constructed in a pre-process by exploring and learning from all possible situations. The learned controller can make optimal or near-optimal decisions in real-time, reacting to user input or changes in the environment. Furthermore, reinforcement learning allows formulating only high level goals, such as obstacle avoidance or grasping objects at specific locations. Control policies can be generated automatically to achieve these goals, and there is no need to specify which states are more preferable in which situations.

The rest of this section is organized as follows. Section 4.3.1 describes how the problem of motion planning can be framed in the context of reinforcement learning, and Section 4.3.2 reviews different strategies to learn motion controllers. Section 4.3.3 summarizes the advantage and disadvantage of the reviewed controllers and concludes this chapter.

4.3.1 Problem Formulation

In general, the motion control problem can be modeled as a deterministic MDP with discrete-time dynamics:

- The action space $\mathcal{A}$ is defined as the collection of motion fragments. Some systems split the motions in the database into short clips in a pre-process and allow transitions between any two clips. In this scenario, an element $a \in \mathcal{A}$ denotes a specific motion clip. Alternatively, some systems build a motion graph directly from the motion capture database to encode smooth transitions among the motions (see Chapter 2). In the second scenario, an element $a \in \mathcal{A}$ represents a graph link in the motion graph.

- The state space $\mathcal{S}$ is composed of all possible configurations of the current character motion, the environment, and user input. The parameterization of state space is application specific, but in general a state $s_t \in \mathcal{S}$ is a vector $(a_{t-1}, x_t^1, \ldots, x_t^n)$, where t is a discrete time step, $a_{t-1} \in \mathcal{A}$ denotes the previously played motion fragment, and $x_t^1, \ldots, x_t^n \in \mathbb{R}^n$ are parameters describing the character's current situation in the environment, such as the relative position to the goal, or deviation from desired orientation.

- The transition model is deterministic and defined with the *transition function*, f, which describes how a state is updated when a certain action is executed:
$$f(s_t, a_t) = s_{t+1} = (a_t, x_{t+1}^1, \ldots, x_{t+1}^n). \tag{4.23}$$

$$\begin{tikzcd} s_0 \arrow[r, "a_0=\pi(s_0)", "R(s_0,a_0)"'] & s_1 \arrow[r, "a_1=\pi(s_1)", "R(s_1,a_1)"'] & s_2 \arrow[r, dotted] & {} \end{tikzcd}$$

Figure 4.4: An MDP specifying the problem of motion planning: At each discrete time t, the agent is in state s_t, and by taking action $a_t = \pi(s_t)$, the agent reaches state s_{t+1}, receving reward $R(s_t, a_t)$.

At time t, when action a_t is selected, the corresponding motion is concatenated with the previous motion, a_{t-1}, and the state s_{t+1} is updated with the character's new situation in the environment.

- The reward function R is defined as,

$$R(s_t, a_t) = R_s(s_t) + R_t(a_{t-1}, a_t), \tag{4.24}$$

where R_s denotes the *state reward* and R_t denotes the *transition cost*. The state reward measures how well the character respects user objectives and environmental constraints, while the transition cost is used to ensure smooth motion transition between different motion fragments. The state reward is application specific and usually designed manually, while the transition cost is usually computed in a similar way as what we described in Section 2.2.

The system dynamics is illustrated in Figure 4.4.

Care must be taken when designing the state space: on one hand, the state definition should be descriptive enough so that the transitions are Markovian (the next state can be derived solely from the current state and action); on the other hand, the state definition should be as concise as possible to avoid the Bellman curse of dimensionality. Hence, it usually relies on the designer's comprehensive understanding of the problem to define a proper state space with least possible parameters. The previously played action a_{t-1} is included in the state space for a similar purpose of making the transitions Markovian, and also for incorporating a transition penalty into the reward function.

4.3.2 Continuous State Space

When the state and action spaces are discrete and small enough, the value function V^* (or action-value function Q^*) can be tabulated during learning, and the derivation of the control policy from the value function (or action-value function) is straightforward. However, when dealing with motion planning, the state space is usually continuous and large, so the motion controller cannot be learned directly with the conventional methods introduced in Section 4.1 and Section 4.2. In this section, we review some previous work in computer animation that learn motion controllers with reinforcement learning, and discuss how they handle the continuous state space.

Among all techniques, *discretization* of the state space may be the most straightforward way to simplify the problem. Lee and Lee [39] use reinforcement learning to create a controllable and responsive avatar for boxing, and a state is defined to be the opponent's relative position in 3D space. They discretize the state space to form a grid of locations, and apply value iteration (Section 4.1.2) to learn the value function V^* on the grid. In run time, whenever the avatar is provided with more than one available action, the motion controller selects the one that makes transition to the state with the highest value (Equation 4.7). Since a state s may not coincide with a grid point, they approximate its value, $V^*(s)$, by linearly interpolating values at adjacent grid points. Similarly, Lee et al. [44] choose state samples by taking the Cartesian product of the database motion states and a uniform grid sampling of the task parameters. They apply value iteration to learn the value function and also use interpolation to estimate the values at non-sampled points.

McCann and Pollard [50] use reinforcement learning to create a motion controller that can generate high quality motion but also rapidly adapt to changes in player input. In their work, a state is defined as $s_t = (a_{t-1}, c_t)$, where $a_{t-1} \in \mathcal{A}$ refers to the previouly played motion fragment, and c_t denotes the current control signal from a player's input device. The control signals originally exist in a continuous space, and they deal with this problem by discretizing the signals into several bins. Each control signal is then treated as the center of its corresponding bin. Since the control signals generally will not come in a deterministic order, they model the problem with a stochastic MDP, and estimate the transition model by collecting example input streams.

Hence, $T(s_t, a_t, s_{t+1}) = P(c_{t+1}|c_t)$ returns the probability that the next control signal will be in bin c_{t+1}, given the previous control signal is in bin c_t. This is also a rough model of player behavior. The reward function is defined as $R(s_t, a_t) = R_t(a_{t-1}, a_t) + R_s(c_t, a_t)$, where $R_t(a_{t-1}, a_t)$ indicates smoothness and realism of the transition between two fragments, and $R_s(c_t, a_t)$ indicates how closely fragment a_t matches the control signal c_t. In the end, they apply value iteration to learn the value function V^*, and derive the optimal policy from the value function. Similarly, Lee et al. [42] and Levine et at. [46] approximate the value function by binning the state space and by applying the value iteration algorithm.

Ikemoto et al. [22] present a RL framework for controlling autonomous agents in a hostile world. Their state representation encapsulates information about the obstacles and enemies around the agent. They discretize the continuous state space by placing bins locally in the nearby space around the agent. If the center of a bin falls within an object, the bin is marked as occupied. In addition, they represent the value function in a parametric form. A general parametric value function can be written as:

$$V(s) = \theta_1 b_1(s) + \theta_2 b_2(s) + \ldots + \theta_n b_n(s), \tag{4.25}$$

where $b_1, \ldots, b_n$ are features (or basis functions) and $\theta_1, \ldots, \theta_n$ are parameters for weighting the features. In their work, the features in the parametric value function include the distance to the goal and kernel functions that isolate the effect of each bin surrounding the agent. They sample a few states in the state space and learn a set of parameters, $(\theta_1^s, \ldots, \theta_n^s)$, for each sampled state s. To do this, they generate 3000 random parameter sets for each sampled state, run simulation in the environment using different sets of parameters, and keep the parameters that generate the highest reward for the sampled state. Given a new state, its control parameters can thus be interpolated using the neighboring sampled states.

Treuille et al. [81] use the parametric representation of the value function in a more general way to avoid any discretization of the state space. They define the basis function, $(b_1, \ldots, b_n)$, as polynomials or Gaussians, and adapt a linear programming approach for solving a global set of parameters, $(\theta_1, \ldots, \theta_n)$. The algorithm starts with a set of state samples $\bar{\mathcal{S}} \subset \mathcal{S}$, and an empty set of transition $\mathcal{L} \subseteq \bar{\mathcal{S}} \times \mathcal{A} \times \mathcal{S}$, and alternates between the following two steps until convergence:

- For each state in $\bar{\mathcal{S}}$, use the current value function to build a transition and insert it into $\mathcal{L}$.
- Update the value function by solving the linear program:

$$\max_{\theta_1,\ldots,\theta_n} \sum_{s_t \in \bar{\mathcal{S}}} V(s_t)$$
$$\text{s.t.} V(s_t) \leq R(s_t, a_t) + \gamma V(s_{t+1}), \forall (s_t, a_t, s_{t+1}) \in \mathcal{L}.$$

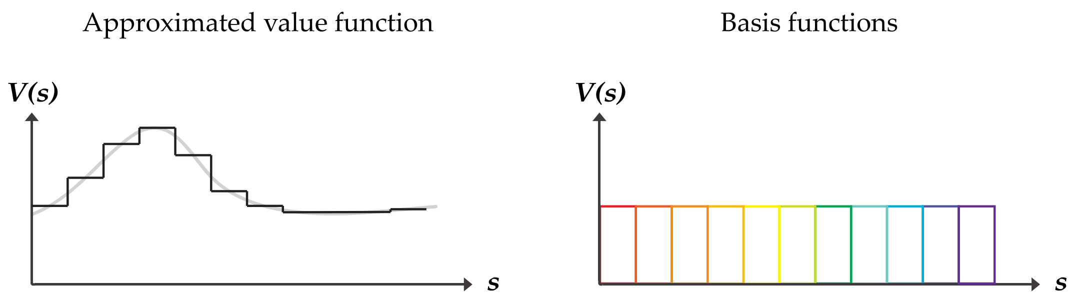
The second step essentially inflates the value function as much as possible subject to the bounds given by the Bellman equation. With a continuous value function representation, their controller can produce motions that fluidly respond to user control and environment constraints in real-time. This approach, however, relies on the assumption that the desired value function can be approximated using only a few basis functions. The learning algorithm becomes very inefficient for larger number of basis functions, because it requires the solution of a linear programming problem in each iteration. In addition, prior knowledge of the optimal value function is required to select a good set of basis functions to fit the manifold, otherwise the algorithm may not converge. Wampler et al. [82] extend this approach by applying truncated PCA to reduce the number of basis functions, and they define the basis functions as the bases of a third-degree B-spline. The predefined set of basis functions, however, is not able to approximate any arbitrarily-shaped value function, and expert knowledge of the domain is required to choose the appropriate basis functions.

4.3.3 Summary

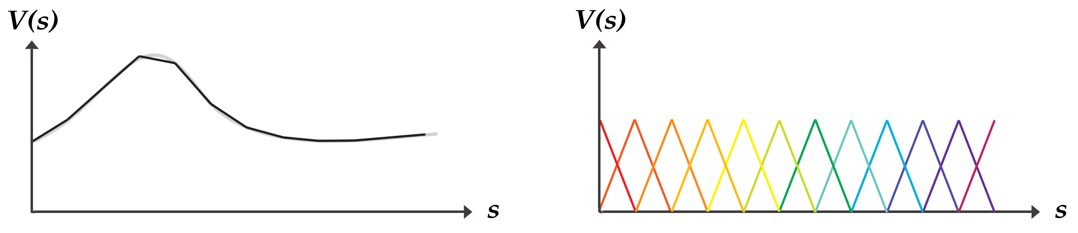
To sum up, each of the methods discussed in Section 4.3.2 can be perceived as fitting a *function approximator* to the value function, whose general form is shown in Equation 4.25. A function approximator allows the agent to generalize from states it has visited to states it has not visited [66]. In the case of discretizing the state space into a fixed number of bins and storing one value for each bin [50, 42, 46], a *piecewise constant* function approximator is used to fit the value function, where $(b_1, \ldots, b_n)$ are box functions, n is the number of bins, and $(\theta_1, \ldots, \theta_n)$ are the values stored in the bins, as shown in Figure 4.5a. In a similar case where the state space is discretized but the values are interpolated among the sampled states [39, 44], a *piecewise linear* function approximator is used instead, where $(b_1, \ldots, b_n)$ are linear interpolating functions, as shown in Figure 4.5b. Piecewise constant and piecewise linear function approximators are simple to implement, but suffer from the facts that the grid resolution strongly influences the quality of the solution, which needs to be tuned manually. However, for simple problems with small state spaces, these might still be good choices.

More complex basis functions can be used to reduce the size of the function approximator [22, 81, 82], as illustrated in Figure 4.5c. This can provide a smooth representation of the value function and achieve considerable compression for large state spaces. The main difficulty, however, is to select a priori shape of the parametric approximation architecture that may lead to some good results. If the basis functions are chosen arbitrarily, the learning may not converge and may even diverge to infinity. Setting up the basis functions thus requires trial and error, because for a non-trivial task the value function is usually hard to guess. In Chapter 5, we will introduce an adaptive function approximator, which is more flexible and robust for learning motion controllers. An example of an adaptive function approximator is shown in Figure 4.5d. After we published our work, Lee et al. [43] adopted a similar idea to adaptively learn the value function with an octree-based representation.

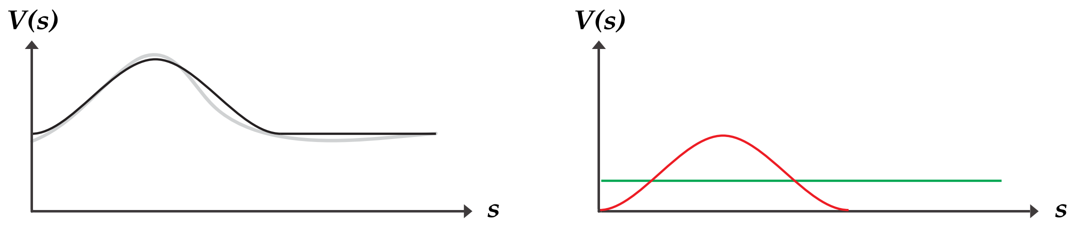
In this chapter, we always assume the environment is *fully observable*. With this assumption, the agent always knows which state it is in. The real world, however, is *partially observable*. Although learning with partially observable MDPs (POMDPs) is important, we opt to leave out introduction to POMDPs in this chapter, since to our knowledge, it has not been used in character animation to model the environment. One type of the learning methods, *policy search*, is also not covered in this chapter for a similar reason. These might be interesting topics to explore in future research of learning motion controllers.



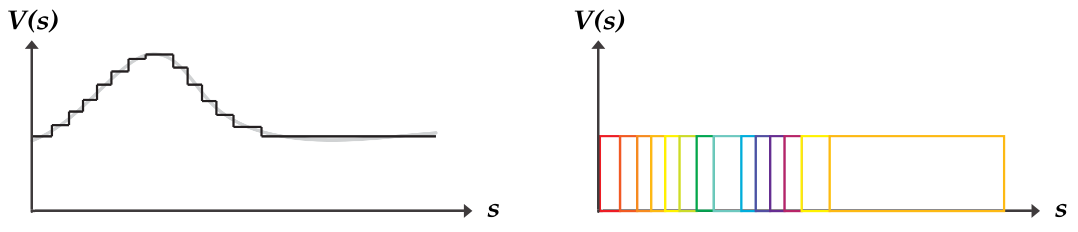
(a) Piecewise constant function approximator



(b) Piecewise linear function approximator



(c) Parametric function approximator



(d) Adaptive function approximator

Figure 4.5: Comparisons of different function approximators. The approximated value functions are shown in the left column, while the corresponding basis functions are shown in the right column. In the example of parametric function approximator (c), a Gaussian and a line are used to approximate the value function.

Chapter 5

Real-time Planning with Parametric Motion

"We balance probabilities and choose the most likely. It is the scientific use of the imagination."

Sherlock Holmes

In this chapter, we present a novel approach to learn motion controllers for real-time character animation based on motion capture data. We employ a *tree-based fitted iteration* algorithm for reinforcement learning, which enables us to generate motions that require planning. This approach is more flexible and more robust than previous strategies. We also extend the learning framework to include parameterized motions and interpolation. This enables us to control the character more precisely with a small amount of motion data. Finally, we present results of our algorithm for three different types of controllers.

5.1 Contributions

In this chapter we describe a reinforcement learning technique for interactive control of human characters. Our approach can generate motions that require planning, and it allows for precise control using parametric blending of several motions. Our algorithm includes two main contributions:

- We use a tree-based fitted iteration algorithm to learn control policies. This approach is more flexible and more robust than previous methods to construct motion controllers using reinforcement learning.
- We extend the reinforcement learning framework to include parameterized motions and interpolation. This allows us to control characters more

precisely without requiring an excessive amount of input data.

Several methods have been proposed in computer animation to utilize reinforcement learning to obtain policies for choosing actions that will increase long term expected rewards [39, 22, 81, 50]. However, as discussed in Section 4.3, having a good approximation of the value function is the key to obtaining a good policy. In this work, we adopt regression trees [14] to adaptively approximate the value function on a continuous state space. Compared to previous works, our approach converges faster without any assumption about the shape of the value function. We can support value functions with any possible shapes.

Moreover, previous works suffer from the limitation that the space of available motions is discrete. This makes it harder to achieve precise control such as walking in an exact direction or stepping on an exact point. Parametric synthesis allows interpolating motions from a parametric space, which is an abstract space defined by kinematic or physical attributes of motions. By parameterizing all motion samples in the space, and by blending among multiple motions, motion interpolation can create novel motions that have specific kinematic or physical attributes [86, 65, 29, 30, 55, 67]. In order to provide accurate control, we present a way to learn *parametric motion controllers*, which can compute the blending parameters in a near-optimal fashion for real-time motion control.

The rest of this chapter is organized as follows: We describe our tree-based fitted iteration algorithm for learning motion controllers in Section 5.2. We describe how to include parameterized motions into a reinforcement learning framework in Section 5.3. Finally, we present results of several motion controllers in Section 5.4 and conclusions and future work in Section 5.5.

5.2 Learning Motion Controllers

In this section we describe a reinforcement learning framework to obtain motion controllers for interactive character animation. Using a database of atomic motion clips, our goal is to generate natural character motion as a sequence of clips. At each time step, the motion controller decides which motion clip best follows the user input and respects constraints imposed by the environment. This decision must be made quickly, since time lags are not allowed in interactive environments. The controller should also be able to achieve user objectives that require planning ahead of time. In addition, both user input and the environment should be represented using continuous parameters to allow for precise control.

The problem formulation is given in Section 4.3, and we use a tree-based fitted iteration algorithm to solve the problem. The algorithm works by observing the agent's interactions with the environment and using the observations to iteratively update the value function. The algorithm is a kernel-based reinforcement learning approach [59], which reformulates reinforcement learning as a sequence of parametric or non-parametric regression problems. In our work, we use regression trees as an approximation architecture to adaptively approximate the value function on a continuous state space [14]. We call this framework tree-based fitted iteration algorithm, for it iteratively fits the tree structures to the value function. In the rest of this section, we first describe a general kernel-based approach to learn the optimal value function in Section 5.2.1, and then introduce our tree-based algorithm for learning motion controllers in Section 5.2.2.

5.2.1 Kernel-based Reinforcement Learning

In the case of discrete state spaces, temporal difference class of methods (e.g. Q-learning algorithm introduced in Section 4.2.3) can be used to learn the optimal action-value function from an agent's experience with the environment. When dealing with continuous or very large discrete state spaces, however, the action-value function cannot be represented anymore in a tabular form with one entry for each state-action pair. Although discretization provides a straightforward way for resolving this issue, it suffers from the facts that the resolution might strongly influences the quality of the solution, so tuning the resolution is a tedious manual work. Ormoneit and Sen [59] presents kernel-based reinforcement learning method to overcome important shortcomings of temporal difference learning in continuous state domains. They reformulate the action-value function determination problem as a sequence of kernel-based regression problems, and allows to fit any (parametric or non-parametric) approximation architecture to the Q-function.

Similarly, we apply the kernel-based approach to approximate the optimal value function. The algorithm starts with an initial value function V that equals zero everywhere in the state space, and iteratively extend the optimization horizon by alternating the following steps until convergence:

1. Generate a set of state samples s, compute their value v based on the current value function, and add each pair (s, v) to a training set $\mathcal{T}$,

   $$\mathcal{T} \leftarrow \mathcal{T} \cup \{(s, v) | s \in \mathcal{S}, v \in \mathbb{R}\}.$$

   The value v of a state s represents the long-term state reward and is given by

   $$v = \max_{a \in \mathcal{A}} \left[R(s, a) + \gamma V(s') \right], \qquad (5.1)$$

   where $s' = f(s, a)$, as defined in Equation 4.23.

2. Update V by fitting a given approximation architecture to the training set $\mathcal{T}$, so that for any tuple (s, v) in $\mathcal{T}$, $V(s)$ is as close to v as possible.

3. Update all tuples in $\mathcal{T}$ using the new value function V.

The algorithm converges when the equilibrium provided in Equation 5.1 is reached and returns the optimal value function. However, for some function approximators there is no guarantee that the algorithm actually converges. Instead, we can define a priori a maximum number of iterations by computing the error bound on the sub-optimality, assuming the function approximator well represents the value function [14]:

$$\|V_N - V^*\|_\infty \leq 2 \frac{\gamma^N B_r}{(1-\gamma)^2}, \qquad (5.2)$$

where V_N is the value function learned after N iterations and B_r is the bound of the reward function. Hence, given a desired level of accuracy, we can fix the maximum number of iterations by using the righthand side of Equation 5.2 to compute the minimum value of N. Note that if the approximation architecture used in Step 2 is not able to provide a good approximation of the value function, the sub-optimality bound in Equation 5.2 is not accurate, but it can still be used as a rough estimate for the stopping condition.

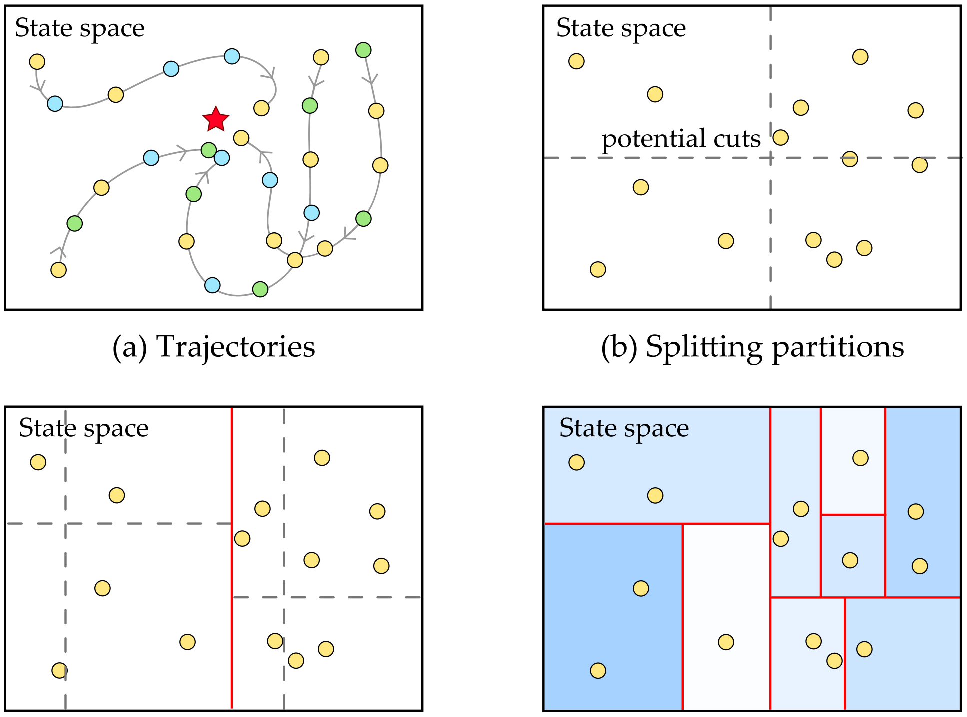
Finding a suitable approximation architecture for Step 2 is crucial for the robustness and efficiency of learning the optimal value function. In Section 4.3.3, we have summarized some function approximators used in previous work of learning motion controllers, and these approximators can also be plugged into the kernel-based RL algorithm. However, Ernest et al. [14] provide a comprehensive comparison among several function approximators, including *k*NN, piecewise constant and piecewise linear grids, and various tree-based methods. Their study of several application cases shows that the *Extra-trees* method [17], which is named for extremely randomized trees and builds an ensemble of regression trees to approximate the value function, performs significantly better than the other methods. Therefore, we adopt Extra-trees as our function approximator in Step 2, and we will present details of our tree-based fitted iteration algorithm in Section 5.2.2.

Generally, a regression tree partitions the training set $\mathcal{T}$ into several regions and determines a constant prediction in each region by averaging the values of the elements from the training set that fall into this region. The Extra-trees algorithm builds the partition in a top-down manner. For each node, it selects a cut position for each dimension at random. It then computes a score for each of the potential cuts to measure the relative variance reduction (as shown in Equation 5.5), and chooses the one that maximizes the score. The algorithm stops splitting a node when the number of samples in this node is less than a pre-defined parameter n_{min}. Each regression tree is built independently using the whole training set to form the final ensemble of trees, which are averaged in the end to approximate the value function.

5.2.2 Tree-based Fitted Iteration Algorithm

Our state space $\mathcal{S}$ is continuous except for the dimension along the motion clips A, which is discrete. Hence, we build an ensemble of regression trees for each clip in the database, but we optimize all the trees at the same time. Here, we present our tree-based fitted iteration algorithm that includes three steps: initialization, iteration, and pruning.

Initialization. We initialize the value function V to zero everywhere on S. We maintain a training set $\mathcal{T}_A$ for each clip A. The training sets are initialized as



(a) Trajectories

(b) Splitting partitions

(c) Splitting recursively

(d) Final regression

Figure 5.1: Illustration of one step in our tree-based fitted iteration. The figure shows an abstract visualization of state space; it should not be interpreted as the 2D position of a character. Please refer to Section 5.4 for the precise definition of state space for different motion controllers. (a) We use the current value function to generate trajectories that approach the goal, depicted by the red star. Each circle represents a sample state. The colors denote different motion clips included in the states. (b) Each sample is added to a training set according to the clip it includes. For each training set we build an ensemble of regression trees. Each tree is built in a top-down manner starting with the whole space as the root node. We generate a potential cut randomly for each dimension and select the cut with highest score. (c) We recursively split the nodes until the number of samples in each node is below or equal to n_{min}. (d) After tree construction, the leaf nodes provide a piecewise constant approximation of the value function, and the final approximation of the value function is obtained by averaging all the regression trees in the ensemble.

empty sets.

Iteration. We illustrate one step of our iteration process in Figure 5.1. In each step, we first add samples to the training sets $\mathcal{T}_A$ by generating a number of trajectories in the system. Each trajectory starts from a randomly chosen initial state and finishes when the task goal is achieved, or when a predefined maximal number of steps is reached. During the trajectories, the action a_t selected at time t is chosen with the current value function V,

$$a_t = \operatorname*{argmax}_a \left[R(s_t, a) + \gamma V\left(f(s_t, a)\right)\right] \tag{5.3}$$

$$= \operatorname*{argmax}_a \left[R_t(a_{t-1}, a) + \gamma V\left(f(s_t, a)\right)\right]. \tag{5.4}$$

For every state s generated in the trajectories, we compute v using Equation 5.1, and then add (s, v) to the the training set of the chosen clip.

After updating the training sets $\mathcal{T}_A$, we use the Extra-trees algorithm to build an ensemble of regression trees for each training set. Each regression tree is built independently with the whole state space treated as a root node, and the nodes are recursively split until the number of samples contained in each node is equal or less than n_{min}. To determine a split at each node, we randomly pick a cut position for each dimension and compute a corresponding score that measures the relative variance reduction,

$$Score = \frac{var(\mathcal{N}) - \frac{\#\mathcal{N}_l}{\#\mathcal{N}} var(\mathcal{N}_l) - \frac{\#\mathcal{N}_r}{\#\mathcal{N}} var(\mathcal{N}_r)}{var(\mathcal{N})}, \tag{5.5}$$

where $\mathcal{N} \subseteq \mathcal{T}_A$ denotes the subset of samples in the node, $\mathcal{N}_l$ and $\mathcal{N}_r$ denote the samples on the two sides of the cut in the node, and var is the empirical variance of the sample values v. We make the cut with highest score. When no more nodes can be split the value function V is updated using the new regression trees, and all existing tuples in the training sets are also updated using Equation 5.1.

The sampling and regression steps are repeated until a stopping condition is reached. We measure the quality of the current value function using the Bellman residual [4], which is defined as the difference between the two sides of the Bellman optimality equation 4.3,

$$V(s) - \max_{a \in \mathcal{A}} \left[R(s, a) + \gamma V(s')\right]. \tag{5.6}$$

Note that V^* is the only function leading to a zero Bellman residual for every possible state. Therefore, the residual measures how close the value function is to the optimal one. In our system, we compute the mean square of the Bellman residual over the training sets. The iteration is stopped if the residual is below a predefined threshold.

Pruning. It is nontrivial for the user to specify n_{min}, the minimal sample size for splitting a node, and the optimal value may vary for different tasks and for different sizes of training sets. Therefore, we use pruning as a post-processing step to automatically determine the maximal number of samples in a leaf. Pruning is carried out by selecting at random two thirds of the elements of $\mathcal{T}$, re-building trees for every possible value of n_{min} with this smaller training set, and determining with which value of n_{min} the square error over the last third samples is minimized. Then, we run the Extra-trees algorithm again on the whole training set $\mathcal{T}$ using this optimal value of n_{min}.

Convergence. Although the Extra-trees algorithm can well extract information from the training sets, it does not guarantee convergence, since it readjusts the approximation architecture, i.e., the tree structure, to the new expected rewards at each iteration. However, and contrary to many parametric approximation schemes, it does not lead to divergence to infinity problems, but just oscillates around some value [14]. To ensure convergence in our system, we freeze the tree structure and stop adding new samples after it begins to oscillate or after the number of iterations exceeds some predefined number. It converges fast with a frozen tree structure.

5.3 Incorporating Parameterized Motion Groups

One of the challenges of character animation based on motion data is that it may require large databases and excessive sampling of the continuous space of motions to allow for precise control of generated motion. Moreover, in our context it would lead to large precomputation and memory requirements to learn and store a value function for each clip in a large database. Here, we present an approach to incorporate parameterized motion groups in our reinforcement learning framework. We effectively reduce the number of actions by clustering similar motions, alleviating the precomputation and storage cost. In addition, each group forms a parameterized subspace of motion. This allows us to obtain precise control over the synthesized motion by continuously interpolating in this space. In the following paragraphs, we first explain how we pre-process our motion data. Then we describe how we cluster motion clips into groups and how we define transition costs between the groups. Finally we demonstrate how to modify the transition function so that the planning controller can work with parameterized groups.

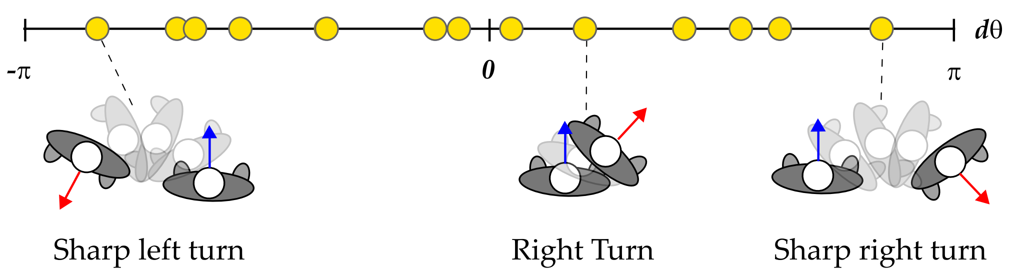
Pre-processing. We manually segment motion data into short motion clips. Our current database includes walking and grasping motions. We define constraint frames for each clip similar to Treuille et al. [81]. The constraint frames are used to temporally align consecutive clips. This allows us to construct a valid animation from any sequence of clips, and to prevent foot-skating. We do not need to construct an explicit graph structure, because unnatural and non-smooth transitions will be avoided by the reinforcement learning approach with the transition reward. To construct a desired controller, we design a state space represented by parameters $(X_1, \ldots, X_n) \in \mathbb{R}^n$. We parameterize each motion clip by determining the change $(dX_1, \ldots, dX_n) \in \mathbb{R}^n$ caused by the clip. Hence, the transition function in Equation 4.23 can be re-written as:

$$f(s, a) = s' = (a, x_1 + dx_1^a, \ldots, x_n + dx_n^a), \tag{5.7}$$

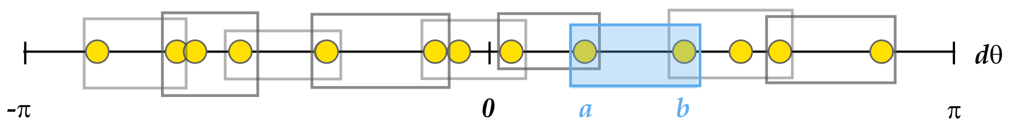
where $dx_1^a, \ldots, dx_n^a$ are the changes of state parameters caused by taking action a. We illustrate this using a navigation controller in Figure 5.2a. For more examples we refer to our results in Section 5.4.

Clustering. In the clustering step, we group similar motions together to share a single value function. Because motion clips are characterized by their instantaneous reward for the purpose of learning, it is reasonable to classify clips with similar reward functions into one group. Our reinforcement learning algorithm is then based on groups of motion clips rather than on individual clips.

To determine clips with similar rewards, remember that the reward is composed of a transition reward and a state reward. Any two motions that are numerically similar [30] will have a similar transition reward. On the other



(a) Parameterization of the clips



(b) Clustering

Figure 5.2: Parameterized groups for the navigation controller, which has only one parameter θ. The value of θ represents the difference between the current and desired walking directions. (a) The motion clips (yellow circles) are parameterized by the change in torso orientation $d\theta$. Blue and red arrows indicate the character's orientation in the first and last frame of the clip, respectively. (b) Parameterized groups. Each group has one corresponding value function.

hand, the state reward measures how well a state fulfills the goal of the controller. According to the transition function in Equation 5.7, if two motions have similar parameters $(dX_1, \ldots, dX_n)$, they will lead to similar states. Therefore they will have similar state rewards. Following these observations, we group together motions that are numerically similar and close in the parametric space. In our current implementation, clustering is performed manually. We illustrate clustering for the navigation controller in Figure 5.2b. Note that a motion clip may belong to several groups. This allows us to make sure that the parameter domain is covered completely by the groups.

Clustering allow us to interpolate in a continuous space of motions spanned by the group members. We use registration curves [29] to perform the interpolation. In other words, each group represents a range of motions instead of a single motion. This allows us to achieve more precise control as shown in Section 5.4.

Transition Cost. We define the transition cost $R_t(a, a')$ between two groups simply as the average of the pairwise transition costs of all clips in the two groups. This is justified because the clips in each group are similar and each pair of clips will have a similar cost.

Parametric Transition Function. We define a modified transition function to incorporate parameterized groups into the reinforcement learning framework. Now an action $a \in \mathcal{A}$ corresponds to the selection of a parameterized group, which represents a continuous range of motions. The long-term reward of taking an action a from the current state is determined by finding an instance on the continuous motion space defined by group a that maximizes the value function.

Assuming there are m members in the group a, we use blending weights $\mathbf{w}_a = (w_1 \ldots w_m)$ to characterize any motion interpolated in the group. We further denote the changes of parameter X_i induced by each memeber in group a as a vector $\mathbf{dx}_i^a$. The parametric change given by a set of blending weights is therefore the vector

$$\left(\mathbf{w}_a \cdot \mathbf{dx}_1^a, \ldots, \mathbf{w}_a \cdot \mathbf{dx}_n^a\right).$$

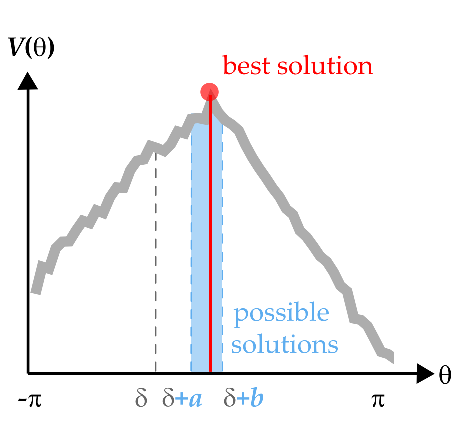
Given a current state $s = (\grave{a}, x_1, \ldots, x_n)$, where $\grave{a}$ denotes the currently played motion (to be distinguished from the next action a), the maximum of the value function for a potential next action a happens at

$$\mathbf{w}_a^* = \underset{\mathbf{w}_a}{\operatorname{argmax}}\, V\left(a, x_1 + \mathbf{w}_a \cdot \mathbf{dx}_1^a, \ldots, x_n + \mathbf{w}_a \cdot \mathbf{dx}_n^a\right). \tag{5.8}$$

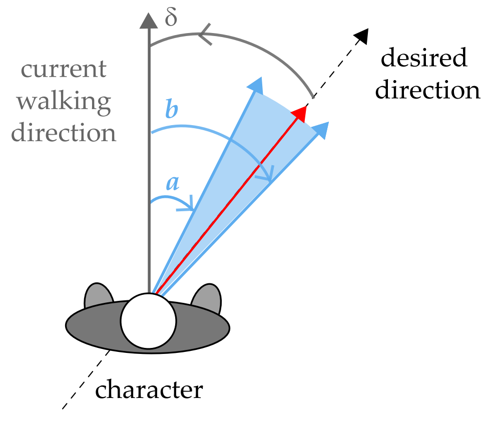
This is also illustrated in Figures 5.3a and 5.3b. In our implementation we solve the above equation by uniformly sampling the space of blending parameters $\mathbf{w}_a$ and picking the one with the highest value. Once the blending weights are determined, the transition function for any state is

$$f(s, a) = s' = \left(a, x_1 + \mathbf{w}_a^* \cdot \mathbf{dx}_1^a, \ldots, x_n + \mathbf{w}_a^* \cdot \mathbf{dx}_n^a\right). \tag{5.9}$$

Parametric Motion Controller. At run time, whenever a motion clip finishes, the controller selects the next group with Equation 5.4. More precisely, this is achieved by scanning every group as a potential next action a, and by using the optimal blending parameters from Equation 5.8 to compute the transition with Equation 5.9.



(a) Value function of one group



(b) Parameter range

Figure 5.3: Finding the optimal blending parameter in a parameterized group. (a) The example value function of the group shaded in Figure 5.2b. Assume the current value of θ is δ, the shaded area represents the possible values of θ in the next time step, with the use of any motion from this group. (b) Visualization of the variables in (a). If the controller picks the best solution in the shaded area in (a), the character will turn exactly toward the desired direction in the next time step.

5.4 Results

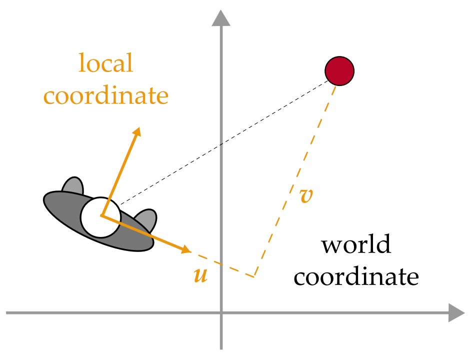
We learned three different controllers to demonstrate our method: navigation, grasping, and guidance.

Navigation. The navigation controller allows a user to navigate a character through an environment by specifying its desired walking direction. A state is given by $s = (\grave{a}, \theta)$, where the parameter $\theta \in [-\pi, \pi)$ measures the angle between the current and the desired walking direction. We define the state reward function as

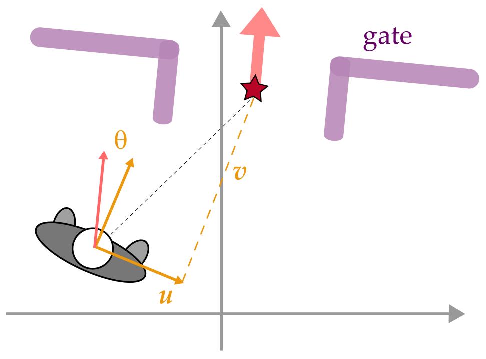
$$R_s(s) = -\alpha|\theta|, \tag{5.10}$$

where α is a scaling parameter.

Grasping. With the grasping controller, the goal of the character is to grasp an object that is at an arbitrary position relative to the character. The object can be



(a) Grasping



(b) Guidance

Figure 5.4: State variables for the grasping (a) and guidance controller (b). The red star and arrow indicate the position and orientation of the gate, and θ is the angle between the the gate and the walking direction of the character.

moved around in real-time, and the character will immediately adjust its path accordingly. Once the character is within reach of the object, the controller picks an optimal way to grasp it according to its relative position.

We define a state for this controller as $s = (\grave{a}, u, v)$, where u and v represent the position of the object projected onto the ground plane in local coordinates of the character, as shown in Figure 5.4a. The state reward function is

$$R_s(s) = \begin{cases} \lambda & \text{if the object is reachable,} \\ 0 & \text{else,} \end{cases} \tag{5.11}$$

where λ is a constant. The motion database for this controller includes walking steps and grasping motions. All clips are parameterized with the change in the object's relative position and clustered as described in Section 5.3. The parameterized grasping motions allow the character to reach the object accurately using motion blending.

Guidance. The goal of this controller is to guide a character to walk through a gate without bumping into walls and doors. We define a state as $s = (\grave{a}, \theta, u, v)$. Here, (u, v) is the position of the center of the gate and θ its orientation, both relative to the character. We illustrate the set-up in Figure 5.4b. Our state

	Time	#groups	$\#\mathcal{T}$	Storage
Navigation	5 s	18	13,752	138KB
Guidance	4 min	8	48,000	125KB
Grasping	20 min	17	178,007	460KB

Figure 5.5: Statistics of our motion controllers, where Time means the learning time.

reward function is

$$R_s(s) = \begin{cases} -\lambda & \text{if close to the obstacles,} \\ -\alpha_1|\theta| - \alpha_2\frac{v}{u^2+v^2} & \text{else,} \end{cases} \tag{5.12}$$

where α_1 and α_2 are scaling parameters, and λ is a very large constant. Intuitively, this reward function penalizes states where the character is close to the walls and doors around the opening of the gate. In addition, it guides the character towards the center of the gate while maintaining the appropriate torso orientation to pass through the gate.

Figure 5.5 shows the statistics of our controllers. At run time, it takes less than 1ms to select the best next clip on a Quad Core 2.4 GHz Intel processor.

5.4.1 Motion Planning

One of the advantages of motion planning is that we can generate animations by just specifying a goal, rather than indicating how a character should fulfill a task. We simply formulate the goal as a reward function. For example for our grasping controller, we only need to specify that the character will obtain a reward if he reaches the object. We do not need to tell the character how to approach the object. In a traditional greedy controller, which picks the best next action without planning into the future, this approach would fail completely. The greedy policy is usually defined as

$$\pi_{greedy}(s) = \operatorname*{argmax}_{a\in\mathcal{A}} \left[R_t(\grave{a}, a) + R_s\left(f(s,a)\right)\right]. \tag{5.13}$$

In the grasping problem, however, the state reward $R_s(s)$ is zero when the object is not reachable. Therefore the character would just walk away as shown in Figure 5.11b. Planning also allows characters to prepare well for obstacles. For

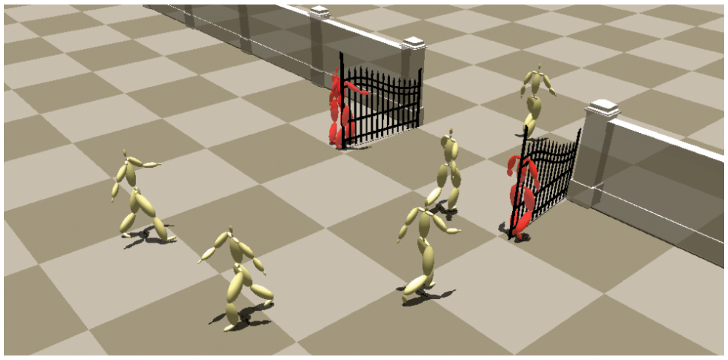
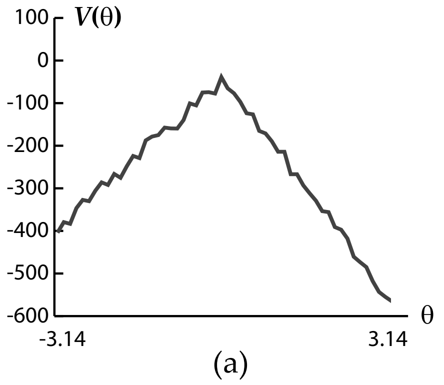


Figure 5.6: The greedy strategy in the guidance controller. In this experiment five out of ten characters bumped into the gate. With our planning controller, however, every character successfully enters it. The starting position and orientation of each character is randomly decided in real-time.

example, if we use the greedy strategy with the guidance controller, characters often run into the gate as illustrated in Figure 5.6. With our approach the characters plan several steps ahead and they successfully avoid the obstacles.

5.4.2 Extra-trees Regression

We compare the navigation controller learned with the work proposed by Treuille et al. to our method. We use twenty motion clips for training, and each clip contains one single walk cycle. Treuille et al. approximate the value function by solving the coefficient vector $\mathbf{r}$ of 2nd-degree polynomials $V \approx r_1 + r_2\theta + r_3\theta^2$. They observed that the quadratic polynomials are only about 11% different from 10th-degree polynomials. Since their method takes much longer to learn with higher order polynomials, they adopted 2nd-degree polynomials for this controller. We use two metrics to asses the performances of the algorithms. The first one measures the quality of the regressions: we randomly sample one thousand states (which may or may not be in $\mathcal{T}$) and average the Bellman residuals in Equation 5.6. The lower the residual, the closer the approximation is to the optimal value function. The second metric measures the quality of the policy: we randomly sample one thousand initial states, trace ten steps from them individually by each policy, and average the long term rewards achieved. The higher the average reward, the better the



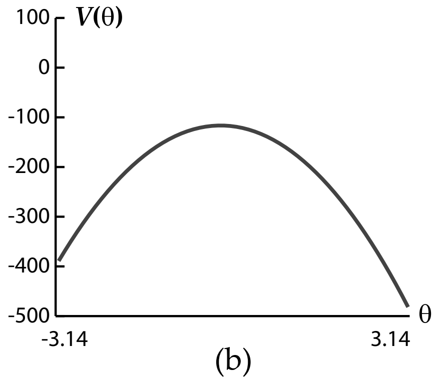


Figure 5.7: Value function of the navigation controller. (a) Tree-based regression. (b) Regression using polynomials.

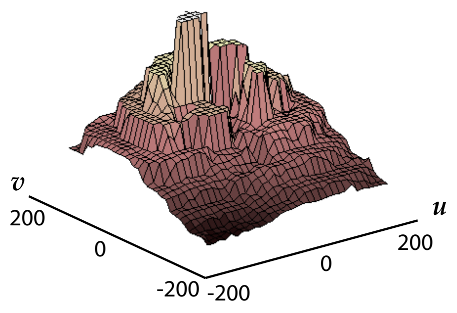
	$\#\mathcal{T}$	Time	Bellman residual	Average reward	Storage
[81]	15,920	7s	6490.31	-328	1k
Ours	15,960	3s	48.45	-231	198k

Figure 5.8: Comparison between previous work and our regression method. With our approach, we are able to obtain a better controller in a shorter amount of time.

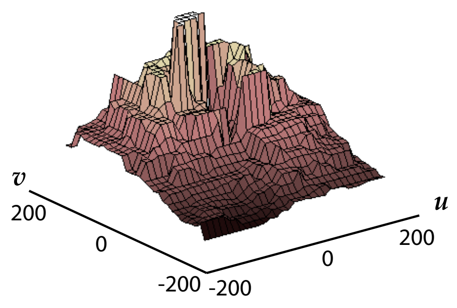
policy performs. We show quantitative results in Figure 5.8, illustrating that our method is scores much better according to these metrics. Our learning time is also much shorter, although we need more memory, because we store tree representations of the value function, while they only need to store coefficient vectors **r**. In Figure 5.7 we visualize the approximated value functions for one clip. This shows that the second order polynomial is not an accurate model for the navigation controller.

We also did the same experiment with the fixed-obstacle-avoidance controller proposed in previous work [81]. Treuille et al. never require more than one hour to learn the controller. Our approach yields a useful controller in less than ten minutes.

For more complex control systems it is hard to approximate the shape of the value function well with a small number of manually designed basis functions. In our experience, the regression algorithm [81] often fails to converge if the basis functions are chosen inappropriately. In contrast, our tree-based algorithm does not make any assumptions about the shape of the value function.



(a)



(b)

Figure 5.9: Value function of the grasping controller. (a) "Ground truth" using about two million samples, three hours learning time. (b) Regression from six thousand samples, less than three minutes learning time.

Figure 5.9a illustrates the complex shape of the value function for the grasping controller. We used dense sampling (about 2 million samples) to obtain a "ground truth" solution in about three hours. Polynomials and Gaussians are ill-suited for approximating value functions such as this one. It is asymmetric, has abrupt changes and even a pit in the middle; but this does not pose a problem for our algorithm. The v axis is asymmetric because it is better to face an object to grasp it, rather than turning one's back to it. Since the reward for successfully grasping the object is a constant, there is a constant plateau near the center. The plateau is slightly off-center because grasping is performed with the right hand. The other plateaus represent the steps needed to walk toward the peak region. For example, if the character's state is in the first plateau near the peak, he needs one more step before being able to grasp. If the character is too close to the object, it is also very difficult to grasp, hence there is a pit in the center of the value function. Figure 5.9b shows a regression computed from six thousand samples in less than three minutes. Here we store about 1700 leaves in the regression tree. This shows that we can obtain a reasonable approximation in a short time.

5.4.3 Parametric Synthesis

The advantage of parametric synthesis is that we gain more precise control. As shown in Figure 5.2c the optimal value may lie between existing mo-

	$\#\mathcal{T}$	Time	Bellman residual	Average reward
non-parametric	15,960	3s	48.45	-231
parametric	13,752	5s	29.63	-161

Figure 5.10: Comparison between parametric and non-parametric planning controllers.

tions. Therefore controllers with non-parameterized motions can only pick sub-optimal clips. Figure 5.10 shows quantitative improvements of parameterized motions for the navigation controller. Figures 5.11c and 5.11d show a comparison with the grasping controller. Without parameterized motions, none of the existing clips initially leads to a good position for grasping the object, so the character takes a detour (Figure 5.11c). With parametric synthesis, a novel motion with optimal value is synthesized and the character turns immediately to grasp the object (Figure 5.11d).

5.4.4 Near-optimal Control

In this experiment, we demonstrate that our controller makes near-optimal decisions in real-time. We modified the reward function of the grasping controller to take into account the effort required for the grasping motion. In general, the effort needed to perform a motion can be approximated by the sum of squared torques computed via inverse dynamics. Figure 5.12b shows the result when the reward is inversely proportional to the effort required for grasping. This motivates the character to step close to the object and pick it in the easiest way. If the reward is positively proportional to the effort required for grasping the character stops at a distance and picks the object in the hardest way, as shown in Figure 5.12c. This demonstrates that the character plans ahead to maximize the long term reward.

5.5 Conclusions

We presented a reinforcement learning framework to obtain motion controllers with parameterized motions. We use a tree-based fitted iteration algorithm to approximate the optimal long-term reward function. This approach is more flexible and more robust than previous methods, and enables us to

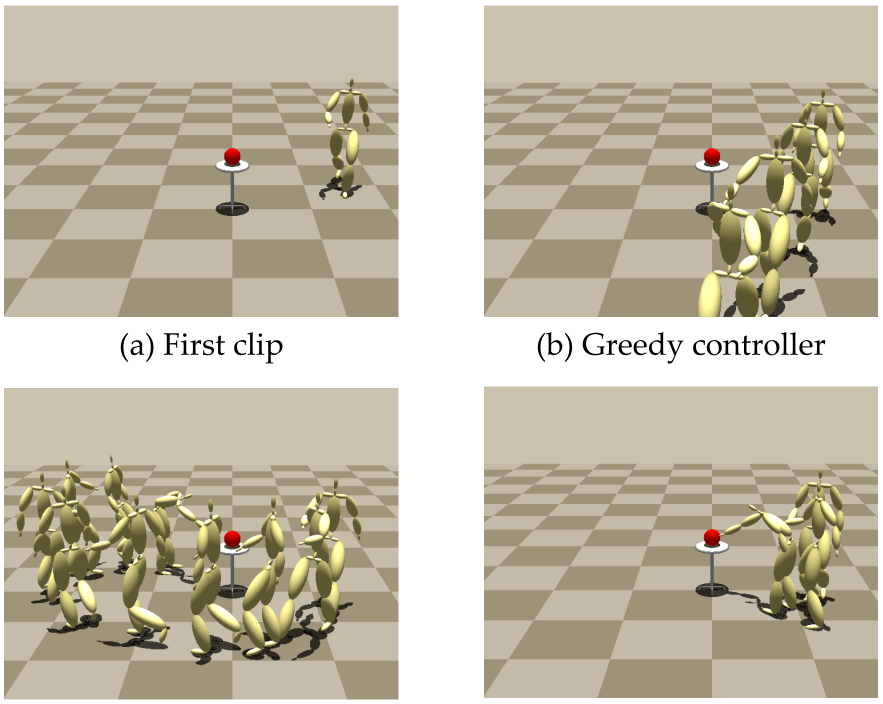
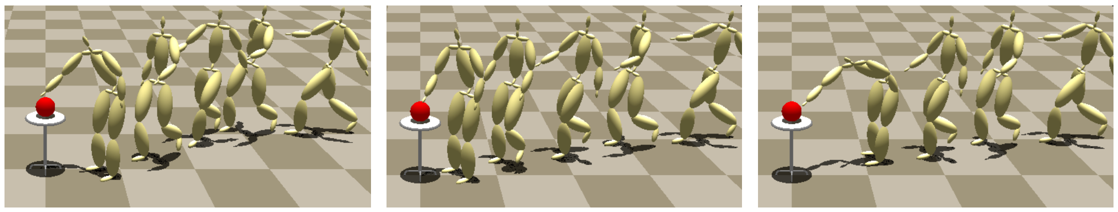

(a) First clip (b) Greedy controller (c) Planning controller (d) Parametric planning controller

Figure 5.11: Comparison of different grasping controllers. (a) Every controller starts with the same first clip. (b) Greedy controller. (c) Planning controller with non-parameterized motions. (d) Planning controller with parameterized motions.

design reward functions in a straightforward way. We also described how to incorporate parameterized motions into the learning framework. This allows us to control characters more precisely with a limited amount of input data. We demonstrate that our approach generates natural animation in real-time for different tasks that require planning.

We believe that the major limitation of our approach is the problem of dimensionality. For more complex environments we need more control parameters to define the state space. Unfortunately, computing time and memory requirements increase superlinearly with the number of dimensions, as more motion data and training samples are required to properly cover the state space. Hence, it is important to sample the high dimensional space effectively. The active learning framework proposed by Cooper et al. [12] could be useful, for it can adaptively determine which motions to add to the system,



(a) Independent of the effort (b) Inversely proportional to the effort (c) Proportional to the effort

Figure 5.12: Real-time near-optimal control. The character is asked to grasp the object (a) in any way, (b) in the easiest way, and (c) in the hardest way.

avoiding capturing and storing nonessential motions. Shum et al. [73] also utilize reinforcement learning for human interactions, and they observe that the subspace of meaningful interactions occupies only a small fraction of the whole state space. They thus propose a way to efficiently collect samples by exploring the subspace where dense interaction occurs. This sampling strategy might also help for high dimensional state spaces. We believe that our approach to include parameterized motions in a reinforcement learning framework is a first step to make this technique more practical. However, future research is required to extend it to more complex environments.

This chapter is based on "Real-Time Planning for Parameterized Human Motion", Wan-Yen Lo and Matthias Zwicker, *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2008.

Chapter 6
Learning with Adaptive Depth Perception

“Learning does not make one learned: there are those who have knowledge and those who have understanding. The first requires memory and the second philosophy."

Alexandre Dumas

Reinforcement learning is appealing to character control due to its ability to make near-optimal decisions in real-time, even taking into account cumulative rewards for actions in the future. Unfortunately, given a desired character behavior, it is often difficult to find a suitable representation of the character's environment. The environment has to be carefully parameterized to limit the dimensionality of the character's state space, as failure to bound the number of state parameters would deteriorate the learning performance. But a too specific state representation also has disadvantages. For example, a controller learned to navigate through circular obstacles, parameterized by their radii, can hardly navigate through sharp obstacles, which are not well described by radii. Currently, controllers are manually parametrized for each task and environment. This process is tedious, and a small change in the state representation requires the learning process to be repeated.

In this chapter, we propose a method which skips this design phase, by letting the character directly “see” the environment using depth perception. Since the large amount of visual information is intractable for learning, we avoid the “curse of dimensionality” by introducing a hierarchical state model and a novel regression algorithm. We build a hierarchical state model by subsampling the input percept, based on the observation that high resolution vision is not always required. When objects are far, blurry vision is enough to make a good decision; when objects are close, clear vision is needed. Also, in general, objects in front of the character require clearer vision than those in other directions. We do not, however, explicitly specify a level of detail for any given situation. Instead, we allow the resolution of visual percepts to be adapted automatically to the scene complexity, and found that the learned controllers confirm this intuition. Finally, we demonstrate that our controllers allow a character to navigate or survive in environments containing arbitrarily shaped obstacles, which is hard to achieve with previous reinforcement learning frameworks.

6.1 Contributions

In this chapter, we propose a method that does not require any ad-hoc parametrization of the environment. Instead of letting the character read a manually interpreted description of the environment, we allow the character to perceive depth directly. We present the following contributions:

- A hierarchical state model for reinforcement learning to replace a single state definition of fixed dimensionality.
- The use of a hierarchical state model in character animation to allow characters to perceive depth. Our approach avoids the need to carefully design ad-hoc parameterizations of environments based on their specific properties.
- An efficient reinforcement learning algorithm that integrates our hierarchical state model. To this end, we present a regression algorithm with automatic resolution adaptation based on scene complexity.

In robotics and artificial intelligence, vision-based sensing has been combined with local path planning [51, 9] and reinforcement learning [52, 23, 24], allowing the agent to navigate toward the goal while avoiding obstacles. Michels et al. [52] define a state by dividing the image into a set of directions. Each direction is a vertical stripe encoding the log distance to the nearest obstacle. However, the number of stripes is pre-defined and does not adapt to scene complexity. On the other hand, Jodogne and Piater [24] argue that most approaches in RL rely on preprocessing the visual input to extract task-relevant information. To avoid task specific coding, they use an automatic image classifier to partition the visual space adaptively while learning. However, they use a fixed map to train the classifier and any change in the map requires the classifier to be retrained.

In computer graphics, we can use synthetic vision to replace real cameras. With modern graphics hardware, accurate visual percepts are obtained efficiently while the problems of depth reconstruction or object recognition from noisy images are avoided. Synthetic vision has already been used in computer graphics for steering virtual humans [56, 32, 61, 58]. However, without pre-planning, computing the solution at run-time leads to a trade-off between optimality and efficiency. In applied perception, Sprague et al. [76] introduce a more elaborate visual model by allowing the virtual character to control the gaze, and use RL algorithms for building mappings from visual percepts to body movements. However, their definitions of state and action space in the RL framework are simple: their obstacle avoidance controller can only take into account one nearest obstacle and three possible turn angles. On the contrary, our perception model does not limit the number of objects in the scene. Our state definition is similar to the one used by Michels et al. [52], but inspired by Jodogne and Piater's work [23], we subdivide the depth percepts adaptively to adjust the level of discretization dynamically and automatically according to the complexity of the scene. Applying our state model to RL, we can achieve near-optimal character control in real-time.

The reinforcement learning framework for learning motion controllers has been introduced in Section 4.3, and this chapter is organized as follows. In Section 6.2, we first explain how virtual environments are parameterized in previous RL frameworks, and then propose our approach based on visual percepts that does not require ad-hoc parameterization of each different type of environment. Learning directly with the visual percepts, however, suffers from the curse of dimensionality. To overcome the problem, we introduce a hierarchical state model in Section 6.3, and present a novel regression algorithm to allow learning motion controllers with adaptive depth perception in Section 6.4. Finally, we show comparisons and results in Section 6.5 and conclude this chapter in Section 6.6.

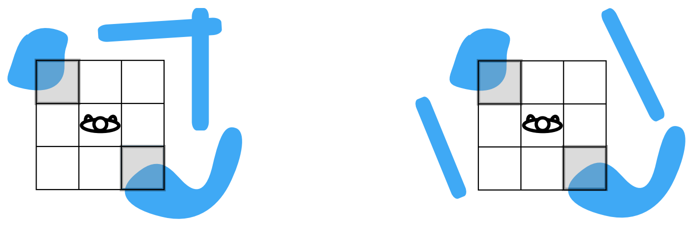
6.2 State Representation

Previous works show that finding a state representation for efficient learning remains challenging. Most state representations are tailored for each specific task, in order to be complete enough to make decisions, but also concise enough to avoid the curse of dimensionality. Lee and Popović [42] demonstrate motion controllers on environments with varying number and shapes of obstacles, but they define a state only with the character's absolute location in the environment, assuming the environment is fixed. For every change in the environment, a new controller must be learned. To support arbitrary configurations, many previous works parameterize each object in the environment explicitly, but in order to simplify the state representation only a fixed small number of objects can be parameterized at a time [81, 49, 43, 46]. Typically, the task parameters (Section 4.1) are defined as

$$\theta = (x_1, y_1, \mathbf{s}_1, \ldots, x_m, y_m, \mathbf{s}_m), \tag{6.1}$$

where m is the number of objects in the environment, and (x_i, y_i) and $\mathbf{s}_i$ are the relative position and shape description of the ith object respectively. The value of m is predefined, yet allowing more objects in the environment makes the learning problem harder. We show in Section 6.5.2 that by increasing m, the learning performance and the quality of the controller deteriorate quickly. Although we can consider only the m closest objects instead of all objects in the environment, there may be more than m objects equally close to the character. Unable to perceive its complete situation, the character may easily make a fatal decision. Moreover, the descriptive power of shape parameter $\mathbf{s}_i$ is usually limited to avoid high dimensionality, which explains the popular use of cylinders in previous work. Finally, a controller learned with one shape description cannot be easily generalized to other shape descriptions.

A natural way to resolve this complexity is to let the character directly "see" the environment, by defining a state with the character's vision. One way to model the character's perception is to use an overhead camera and represent the character's local surroundings with a grid of binary valued cells [22, 51]. The value in each cell denotes whether the location is occupied by an obstacle. However, we found that by rendering the scene from the first person view [32] and by dividing the image into vertical stripes [52], we obtain a more


(a) Overhead view

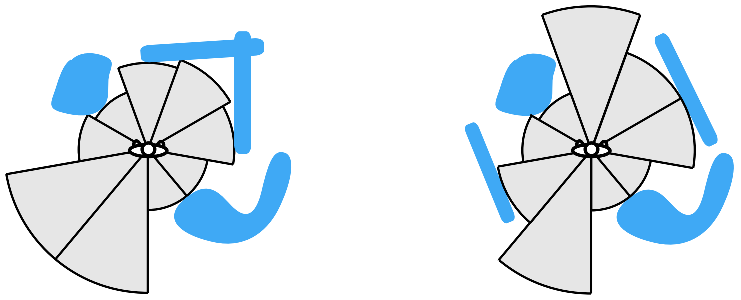

(b) First person view

Figure 6.1: Comparison of perception models using overhead view (a) and first person view (b). The blue regions are real obstacles and the gray regions are their representations in the perception model. In this example, the overhead view classifies the two different environments as the same, while the first person view successfully discriminates them.

descriptive representation with fewer parameters, as shown in Figure 6.1. More specifically, we mount virtual cameras on the character to capture n depth values from the character's panoramic field of view, and store for each stripe i the distance d_i to the closest object. Therefore, we define the task parameters in a state as

$$\theta = (d_1, \ldots, d_n). \tag{6.2}$$

However, it is difficult to use this representation directly in existing RL frameworks, since the value of n needs to be predefined. If the value is too small, the state representation is not descriptive enough for the learning algorithms to converge. If the value is too large, the problem is again cursed by dimensionality. Therefore, we propose a hierarchical state model and a novel regression algorithm to allow the use of depth perception in RL frameworks.

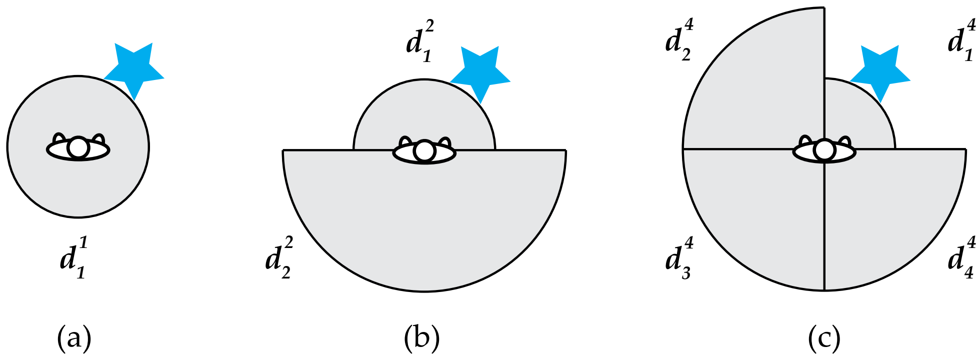


Figure 6.2: Visualization of our hierarchical state model. (a) When $n = 1$, the depth value d_1^1 indicates the distance to the closest object in the scene. (b) When $n = 2$, the two depth values indicate the shortest distances to any object in the front and in the back respectively. (c) With $n = 4$, the character can now tell there are objects in one of the directions, but none in other directions.

6.3 Hierarchical State Model

We propose a novel approach to adjust the dimensionality of the perception automatically and adaptively in the learning process. Our approach is based on the observation that high resolution vision is not always required, but low resolutions vision is sometimes enough for making good decisions. Hence, we build a hierarchy by sub-sampling the input percept, and reformulate the task parameters using a hierarchical representation

$$\theta = (\mathbf{d}^n, \mathbf{d}^{\frac{n}{2}}, \mathbf{d}^{\frac{n}{4}}, \ldots, \mathbf{d}^1), \tag{6.3}$$

where n denotes the finest resolution of the character's perception, and we let n to be a power of 2 to simplify the definition. The vector $\mathbf{d}^i$ is defined as

$$\mathbf{d}^i = (d_1^i, d_2^i, d_3^i, \ldots, d_i^i), i = 1, 2, 4, \ldots, n, \tag{6.4}$$

where i represents the level of sub-sampled vision, and d_j^i corresponds to a non-negative depth value. The perception with the highest resolution is directly obtained from the camera mounted on the character, and for other resolutions the depth values are computed as

$$d_j^i = \min\left(d_{2j}^{2i}, d_{2j-1}^{2i}\right). \tag{6.5}$$

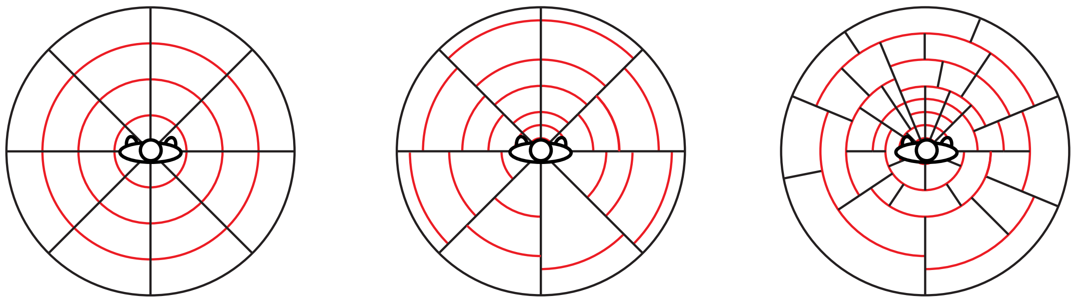
These definitions are visualized in Figure 6.2.

6.4 Adaptive Learning

In this section, we first explain our choice of the learning algorithm and then present a novel regression algorithm to allow learning with the hierarchical state model. Since the depth perception model does not allow the character to see everything in the environment (some objects may be occluded), the transition model T in our MDP is stochastic and unknown. Hence, we approximate the optimal action-value function (Equation 4.4), instead of the optimal value function (Equation 4.1), because Equation 4.6 is more favorable for decision making for its simplicity, especially when performing one-step look-ahead is expensive or the transition model is unknown. We approximate Q^* using the *fitted Q iteration algorithm* [15], which reformulates the Q-function determination problem a sequence of kernel-based regression problems (see Section 5.2.1 for discussion and Algorithm 1 for pseudo codes). The algorithm takes as input an approximation architecture, and a sequence of the character's interactions with the environment (s_t, a_t, r_t, s_{t+1}), where $r_t \in \mathbb{R}$ is the reward of taking action a_t in state s_t. The algorithm iteratively updates the action-value function and uses the updated policy to generate new sequences of interactions.

Algorithm 1 Fitted Q Iteration Algorithm

```
Input: a set of transition tuples $\mathcal{T}$ and an approximation architecture $F$
Output: an approximation of the optimal action-value function $\hat{Q}^*$
1:  $n \leftarrow 0$
2:  $\hat{Q}_n \leftarrow 0$
3:  repeat
4:      $n \leftarrow n + 1$
5:      for all $(s_t, a_t, r_t, s_{t+1})$ in $\mathcal{T}$ do
6:          $x_t = (s_t, a_t)$
7:          $y_t = r_t + \gamma \max_{a \in A} \hat{Q}_{n-1}(s_{t+1}, a)$
8:      end for
9:      Use $F$ to induce $\hat{Q}_n$ from $\{(x_t, y_t) : t = 1, \ldots, |\mathcal{T}|\}$
10:     Generate new transitions with $\hat{Q}_n$ and insert them into $\mathcal{T}$.
11: until $\|\hat{Q}_n - \hat{Q}_{n-1}\|_\infty < \epsilon_q$
12: return $\hat{Q}_n$
```



(a) Uniform grid (b) Adaptive regression (c) Hierarchical adaptive regression

Figure 6.3: Illustration of the depth perception space discretized with (a) uniform grid, (b) adaptive regression, and (c) hierarchical adaptive regression. The resolution of perception is fixed for (a) and (b), shown by the black angular dividers. But with adaptive regression (b), each dimension is discretized adaptively, shown as red radial dividers. With our hierarchical regression algorithm, the resolution can be adaptively adjusted and each dimension can also be adaptively discretized.

We choose Extra-trees (Section 5.2.1 and Section 5.2.2) as the approximation architecture in Algorithm 1. The Extra-trees algorithm works with a fixed hierarchy and builds regression trees to adaptively discretize the state space in each dimension of Equation 6.2. Compared to the use of a uniform grid, as shown in Figure 6.3a, adaptive regression allows some portions of the perception space to be discretized finer than others, as shown in Figure 6.3b. This provides better approximation of the action-value function, because if the object is far from the character, a displacement does not affect the long term expected reward as much as if the object is close. Therefore, the state space can be well represented with coarse nodes when the distance values are large, and with fine nodes when the values are small, as shown in Figure 6.4.

The Extra-trees algorithm allows us to discretize each dimension in Equation 6.2 adaptively but it does not allow us to adaptively adjust the number of dimensions n. In order to adaptively discretize the perception space in both radial and angular directions, as shown in Figure 6.3c, we present a regression algorithm that works with the hierarchical state model introduced in Section 6.3. Our Hierarchical Extra-trees algorithm maintains the original

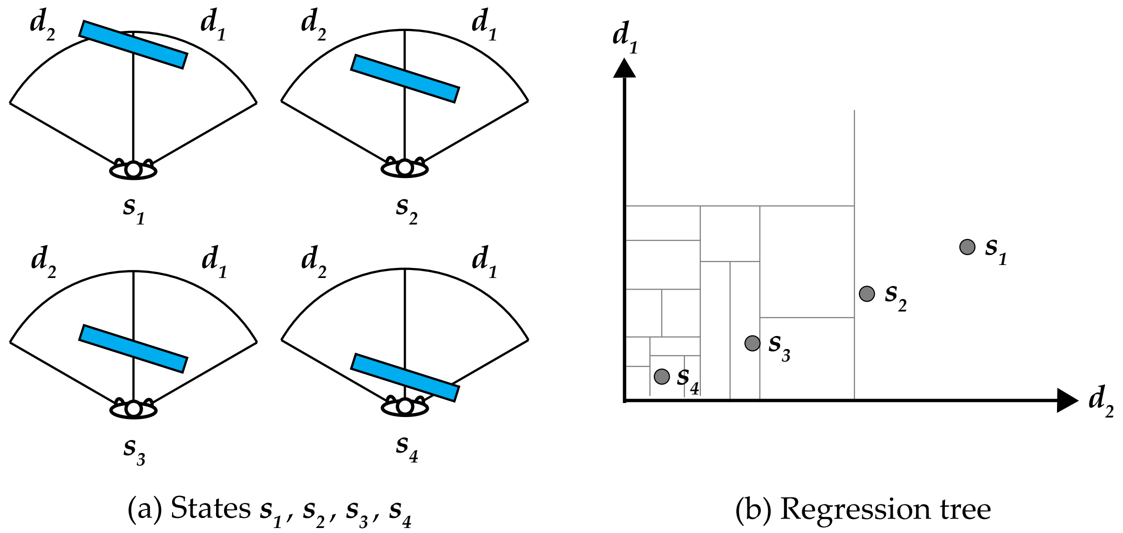


Figure 6.4: Example of evaluating $Q(s, a)$ for different input states shown in (a), using the regression tree (b). The state space consists of two dimensions d_1 and d_2. Taking the same action for states s_1 and s_2 results in similar values, so they are located in the same leaf node of the regression tree. But for states s_3 and s_4, the values are different and they are therefore located in different leaf nodes.

framework of iteratively selecting a node split that leads to highest relative variance reduction, while allowing the set of dimensions be adjusted in a hierarchical way. In Algorithm 2, Line 7–16 is our modification, while the rest is from the original Extra-trees algorithm.

We start this recursive regression algorithm using the lowest resolution of depth perception, so the input $\mathcal{D}$ equals to $\{(1, 1)\}$, that is, only the value of d_1^1 is considered in θ. In the first few splits, blurry vision is sufficient for discriminating between hostile and preferable states. However, after the character has learned enough using the blurry vision, further splitting along any particular input dimension yields no satisfying variance reduction (Line 7, where ϵ_s is a pre-defined threshold). Since the current set of dimensions cannot discriminate more complex situations, our adaptive strategy is to increase the resolution of the perception, by *refining* the input set of dimensions $\mathcal{D}$. For each dimension in use, we check if we can get better variance reduction by replacing it with its two finer-level dimensions (Line 8–9), and we replace the one

Algorithm 2 Hierarchical Extra-Trees Algorithm

Input: $\mathcal{T}' = \{(x_t = (s_t, a_t), y_t)\}$ and $\mathcal{D} = \{(i, j)\}$

```
1:  procedure SplitNode($\mathcal{T}', \mathcal{D}$)
2:      for all $(i, j)$ in $\mathcal{D}$ do
3:          Find a random split value $c_{i,j}$
4:          Compute the relative variance reduction $r_{i,j}$
5:      end for
6:      $(a, b) \leftarrow \text{argmin}_{i,j}\, r_{i,j}$
7:      if $r_{a,b} < \epsilon_s$ then
8:          $\mathcal{D}' \leftarrow \{(2i, 2j), (2i, 2j-1), \forall (i, j) \in \mathcal{D}\}$
9:          Repeat 1-6 using $\mathcal{D}'$ to get $(a', b')$
10:         if $r_{a',b'} > r_{a,b}$ then
11:             $(p, q) \leftarrow (\frac{a'}{2}, \lfloor \frac{b'}{2} \rfloor)$
12:             $\mathcal{D} \leftarrow \mathcal{D} \setminus \{(p, q)\}$
13:             $\mathcal{D} \leftarrow \mathcal{D} \cup \{(2p, 2q), (2p, 2q+1)\}$
14:             $(a, b) \leftarrow (a', b')$
15:         end if
16:     end if
17:     $\mathcal{T}'_L \leftarrow \{(x_t, y_t) | s_t.d^a_b < c_{a,b}\}$
18:     $\mathcal{T}'_R \leftarrow \{(x_t, y_t) | s_t.d^a_b \geq c_{a,b}\}$
19:     SplitNode($\mathcal{T}'_L, \mathcal{D}$)
20:     SplitNode($\mathcal{T}'_R, \mathcal{D}$)
21: end procedure
```

with the best improvement (Line 10–15). The subtrees from this node will use this new set of dimensions (Line 19–20) until it is not descriptive enough and refined again. Hence, each node in the regression tree might use a different set of dimensions for splitting the state space, as illustrated in Figure 6.5. The adaptive refinement stops when the finest resolution level n is reached.

6.5 Results

We collect a few minutes of motion capture data of a person walking around, and organize the data by building a well-connected motion graph [93]

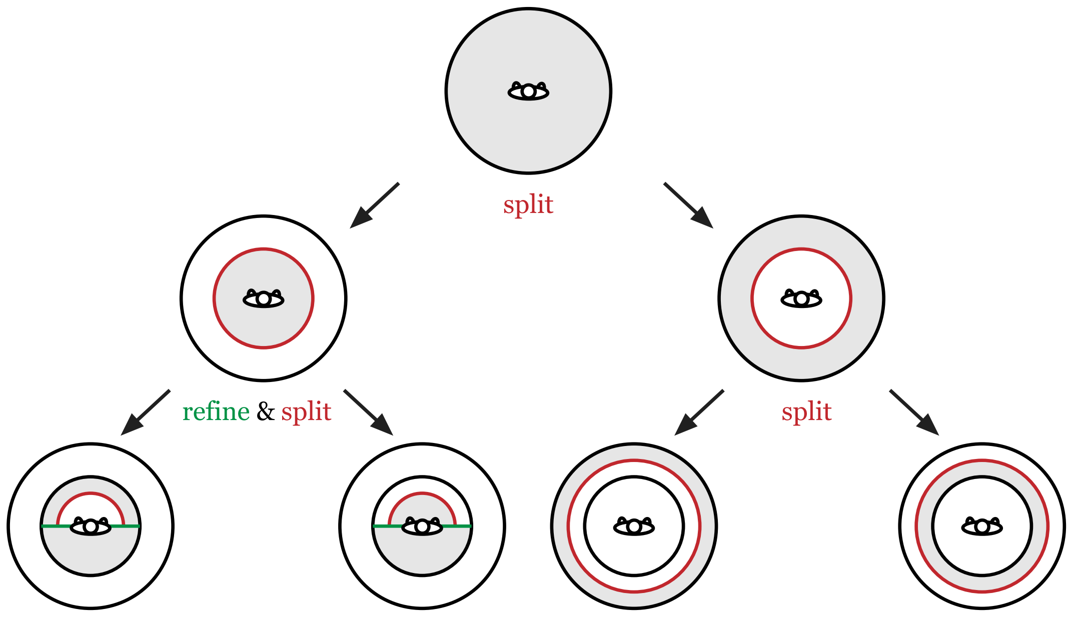


Figure 6.5: Illustration of building a regression tree with our hierarchical regression algorithm. "Split" (shown in red) is what the original Extra-trees algorithm does, "refine" (shown in green) means refinement of our hierarchical state model. The initial state consists of one-dimensional perception with lowest resolution. First we find a cut along the dimension and split the root node into a left and right subtree. The right subtree represents those states where even the closest object is distant from the character. When all the objects are far, blurred vision is enough, so no state refinement is required. If the character is in a state represented by the left subtree, there is at least one close object, and the character requires clearer vision to judge the situation. Hence, only after the resolution of the perception is increased can we find a cut to sufficiently reduce the variance.

to ensure smooth transitions between motion segments. The graph contains 2173 nodes and 2276 links. We further collapse the links with only one successor and one predecessor and merge those nodes on the links. The final graph contains 25 nodes and 128 links. Each graph link corresponds to an action in our MDP.

To simulate the character's vision, we place four cameras on the body, each with 90 degrees field of view, covering the panoramic vision. We use object false-coloring for rendering different object types, e.g., goals and obstacles. Instead of introducing extra dimensions in the state definition to store the types, we use a set of dimensions for each object type separately.

The remaining section is organized as follows. We first explain the experiment setup in Section 6.5.1. We demonstrate the effectiveness of using the depth perception model in 6.5.2. We further show in Section 6.5.3 that by adjusting the perception adaptively, the learning efficiency is greatly improved. We analyze the optimality and generalizability of the result controllers in Section 6.5.4 and Section 6.5.5 respectively. In the end, in Section 6.5.6, we use a game-like scenario to highlight the advantage of applying reinforcement learning in real-time applications.

6.5.1 Experiment Setup

In Section 6.5.2–6.5.5, we make comparisons by learning *navigation controllers*. The objective of a navigation controller is to guide the character to a goal without colliding with any obstacle in the scene. Our reward function is defined as: $+100$ for stopping in a goal region, -200 for collision, -1 for each second elapsed, and the transition cost of concatenating two motion fragments. When acquiring the character's perception, we render the goal objects in a separate pass so the character can always see the goal, but the obstacles will occlude each other. All the comparisons are made on a PC with a Xeon 2.50GHz dual 4-core CPU and 8GB memory, using the same learning framework:

- We rebuild regression trees using the (hierarchical) Extra-trees algorithm in each of the first 50 iterations of the fitted Q iteration algorithm. After 50 iterations, we freeze the tree structure, stop generating new transition samples, and let the algorithm run until convergence. The convergence parameter ϵ_q in Algorithm 1 is set to be 0.0001.

- The transition samples (s_t, a_t, r_t, s_{t+1}) are obtained by generating trajectories in different environments with varying number and position of objects. Each trajectory starts from a random initial position and finishes when a goal is reached, or when it reaches 100 time steps. In each iteration of fitted Q iteration algorithm, we simulate N trajectories with *ϵ-greedy exploration*, where the agent chooses a random action with probability ϵ and follows the currently learned policy with probability 1 - ϵ.

In our setup, $\epsilon = 0.1$.

- We build 10 regression trees to approximate an action-value function $Q(s,a)$, that is, having 128 actions in our MDP, we produce 1280 regression trees for a motion controller. Although more regression trees can provide a better, smoother approximation of Q, the learning time and memory requirement increase linearly. Empirically we found 10 trees provide good approximations within a reasonable amount of time.

To assess the quality of a motion controller, we run simulations with the controller to compute the average expected return. We start each simulation by placing the character randomly in a random environment, and run the simulation by using the controller to navigate the character. The expected return of a simulation is computed as $\sum_{t=0}^{1000} \gamma^t R(s_t, a_t)$. In the end, we average the expected returns from all simulations to obtain the *score* of a controller. The higher the score is, the better the controller performs. To make fair comparisons, we fix a set of 10,000 random environments and initial states, so the average scores of different controllers are directly comparable. Finally, since the learning algorithm is randomized, we repeat the learning process three times for each experiment and average the scores and running time.

6.5.2 Depth Perception

We first compare the conventional state representation using explicit parameterization of obstacles with our novel approach based on depth perception. The environment contains a goal object and m obstacles. All objects are cylinders with variable sizes. In this comparison, the task parameters θ^e (explicit parameterization) and θ^d (depth perception) are defined respectively as

$$\theta^e = (x_1, y_1, a_1, \ldots, x_{m+1}, y_{m+1}, a_{m+1})$$
$$\theta^d = (d_1^g, d_2^g, d_3^g, d_4^g, d_1^o, d_2^o, d_3^o, d_4^o),$$

where (x_i, y_i) and $a_i = \tan^{-1}(r_i/\sqrt{x_i^2 + y_i^2})$ denote the relative position and viewing angle of the ith object whose radius is r_i; $(d_1^g, d_2^g, d_3^g, d_4^g)$ denote the character's depth perception of the goal in four directions; $(d_1^o, d_2^o, d_3^o, d_4^o)$ are

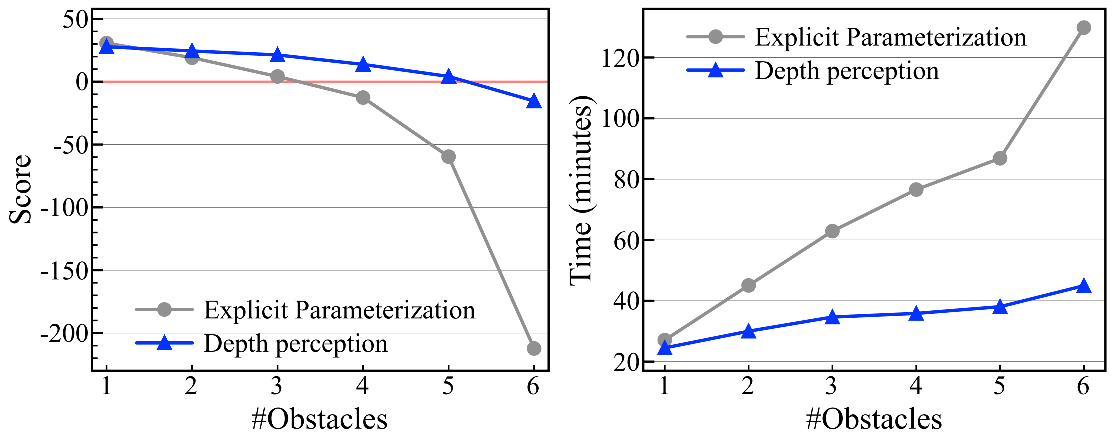


Figure 6.6: Performance comparison between a controller learned with explicit parameterization of the environment and that learned with depth perception.

defined similarly for the obstacles. In this comparison, $N = 200$ trajectories are generated in each iteration of the learning algorithm to expand the training set.

We compare the two representations with increasing number of obstacles m, and the results are presented in Figure 6.6. We can see that when there is only one obstacle in the scene, the results are comparable both in score and time. However, when there are more obstacles in the scene, θ^e has more dimensions than θ^d, and the performance of the explicit parameterization declines significantly with respect to the number of objects in the environment. Moreover, with explicit parameterization, after m increases to four, the score drops below zero, meaning that in the simulations the character often takes unnecessary detours and collides with obstacles. On the contrary, the depth perception model is more robust when the number of objects increases. The results show that even a perception model with only low resolution ($n = 4$ in Equation 6.2) can lead to a better controller in a shorter amount of time.

6.5.3 Adaptive Depth Perception

Next, we analyze how the learning performance scales with increasing resolution of depth perception. First, the experiment is performed on sparse environments containing a random choice of 1 to 4 obstacles. All objects are cylinder, whose radii are randomly decided. We compare resolution

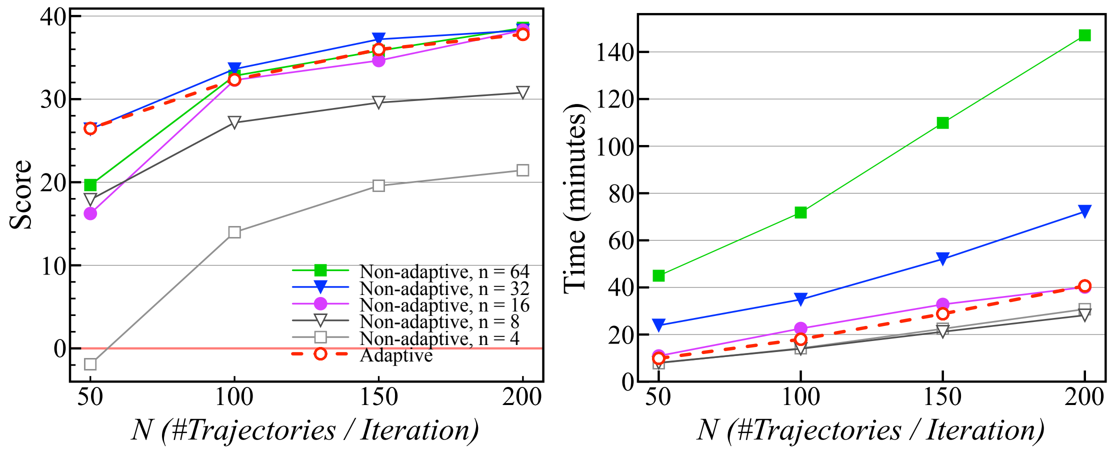
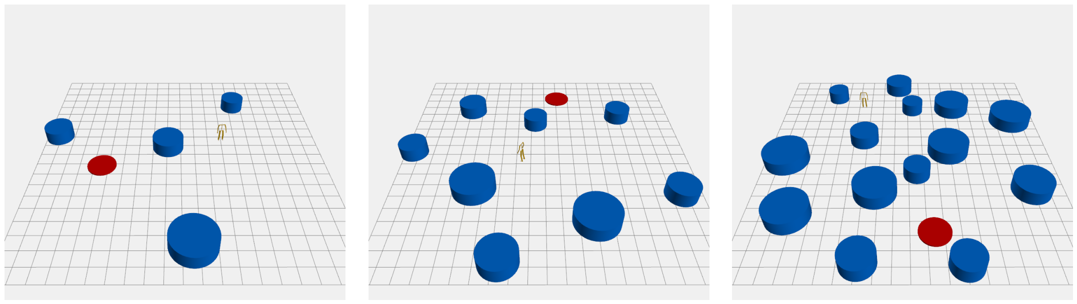
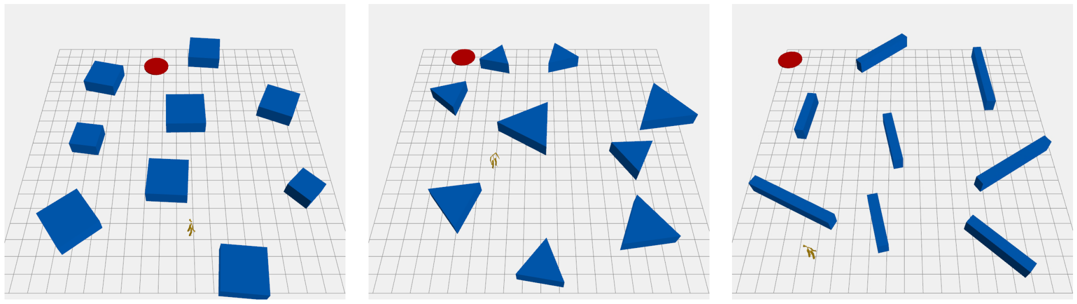


Figure 6.7: We compare depth perception of different resolutions with the proposed adaptive approach. Non-adaptive depth perception, with n fixed to 4, 8, 16, 32, and 64, is plotted in solid lines, while adaptive depth perception is plotted in dotted lines. We experiment with three types of environments with varying density of obstacles, and this figure shows the results of sparse environments containing a random choice of 1 to 4 obstacles. Results of the other densities are shown in Figure 6.9 and Figure 6.10 respectively. Three experiments lead to coherent results. For larger n, computing time increases drastically, while the score does not always improve due to limited number of training samples. Predefining a good value of n is thus difficult, since the best value varies for different scene complexity and available resources. On the contrary, learning with adaptive depth perception has the better scores from high dimensionality and retains shorter computing time from low dimensionality.

of $n = 4, 8, 16, 32$, and 64 in Equation 6.2 (when $n = 4$, the state representation is the same as θ^d in Section 6.5.2), and plot the scores and learning time with respect to N, the number of trajectories generated in each iteration. The plots are shown in Figure 6.7. The plots show that the first two increases of the resolution greatly improves the learning results, but the increase from $n = 16$ to $n = 32$ only moderately improves the scores with the expense of a noticeable increase in learning time. The resolution of $n = 64$, while consuming much more time for learning, does not improve the scores at all. To sum up, with limited samples, the scores cannot be infinitely improved with the increase of resolution, because as the dimensionality increases, the volume of the state space grows so fast that the available samples become sparse. It is thus very difficult to predefine an ideal value of n. In this example, when the environ-



(a) Different sparsity



(b) Different shapes

Figure 6.8: Environments of varied density (a) and containing obstacles of varied shapes (b).

ment is sparse, the ideal fixed resolution is $n = 16$, achieving high scores with short learning time.

However, our adaptive depth perception model outperforms any fixed resolution: compared to $n = 16$, higher scores can be obtained in a shorter amount of time. The result is plotted with dotted red lines in Figure 6.7. More specifically, the multi-scale state representation used in the experiments are defined as

$$\theta^a = (\mathbf{d}_g^n, \mathbf{d}_g^{\frac{n}{2}}, \dots, \mathbf{d}_g^4, \mathbf{d}_o^n, \mathbf{d}_o^{\frac{n}{2}}, \dots, \mathbf{d}_o^4), \tag{6.6}$$

where $\mathbf{d}_g^i$ and $\mathbf{d}_o^i$ are defined as in Equation 6.4 with the subscript g and o denote the perception of the goal and obstacles respectively. The value ϵ_s controls the degree of adaptivity in Algorithm 2, and is set to 0.5.

We further extend the experiments by increasing the density of the environment. The next two experiments are performed on environments containing 6 to 9 and 11 to 14 obstacles respectively, and the densities are visualized in

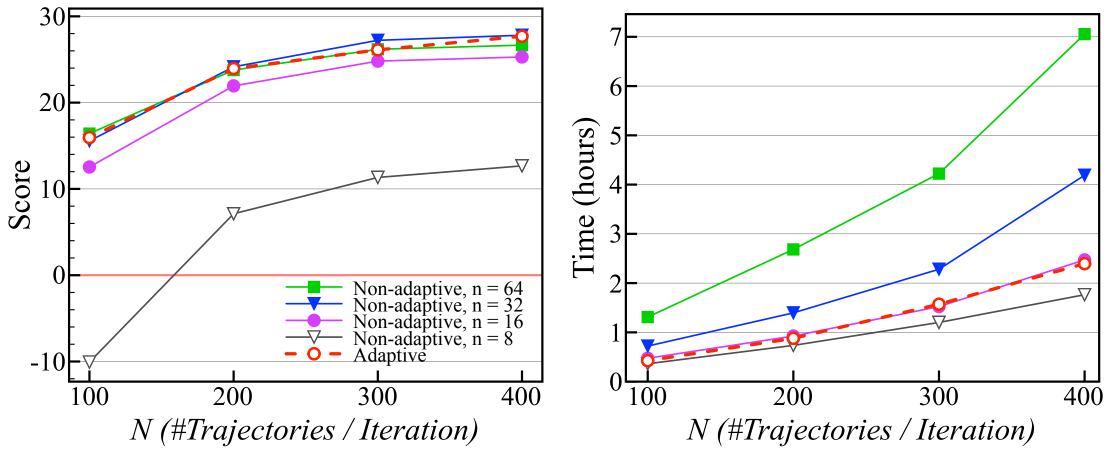


Figure 6.9: We compare depth perception of different resolutions with the proposed adaptive approach. This figure shows the result of denser environments containing a random choice of 6 to 9 obstacles.

Figure 6.8a. When there are 6 to 9 obstacles in the environment, from the plots in Figure 6.9, a similar conclusion can be drawn: 1. Both $n = 16$ and $n = 32$ lead to good scores, but the former is more efficient. 2. Learning with $n = 64$ is much slower, while the scores are not improved. 3. Our adaptive model has the best performance, having the good scores of higher resolutions and the short learning time of the lower resolutions. However, as the environment becomes even more cluttered, higher resolution of perception is demanded to distinguish different scenarios. Figure 6.10 shows that when there are 11 to 14 obstacles in the environment, $n = 16$ is no longer comparable to $n = 32$, and the latter now becomes the ideal fixed resolution. Our adaptive model still obtains the properties of better scores from high resolution and faster learning from low resolution.

6.5.4 Optimality

In order to evaluate the optimality of our controllers, we apply the A* search algorithm to compute the optimal score for the same sets of random environments. We define the heuristic function as the product of the straight line distance from the current position to the goal and the minimal cost required to travel one unit distance. In Figure 6.11, we plot in red dotted lines the

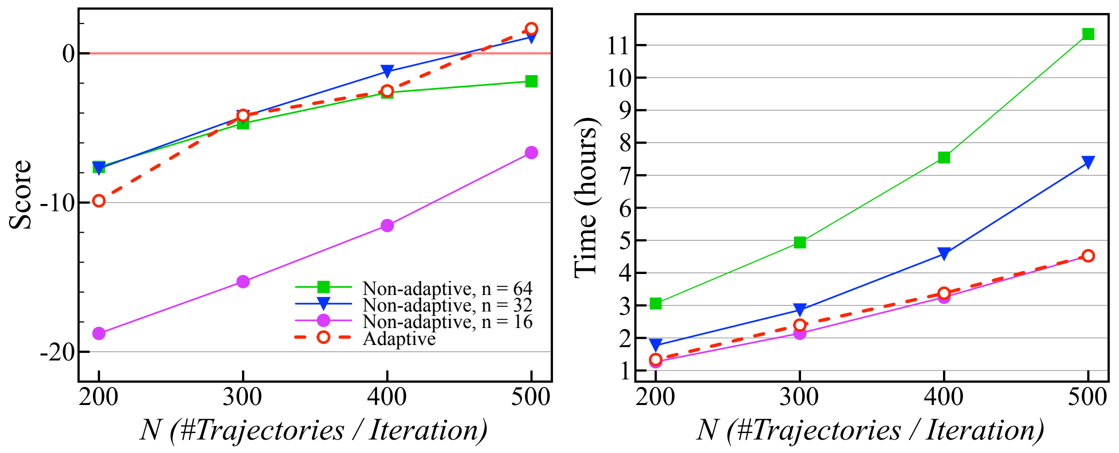


Figure 6.10: We compare depth perception of different resolutions with the proposed adaptive approach. This figure shows the result of cluttered environments containing a random choice of 11 to 14 obstacles.

difference between the scores of our controllers (learned with adaptive depth perception) and the optimal scores from A* search. The differences decrease as the number of sample trajectories N increases, but when learning with more cluttered environments more samples are required to achieve the same optimality. However, since it takes less than 2 milliseconds for the controllers to make a decision in run-time, we can also allow the controller to look ahead one step into the future for making better decisions. Expanding the policy in Equation 4.6, we can obtain the one-step look-ahead policy,

$$\pi(s) = \underset{a}{\operatorname{argmax}} \left(R(s,a) + \max_{a'} \hat{Q}^*(s',a') \right), \tag{6.7}$$

where s' is the consequent state of taking action a from the currents state s, and the depth values in s' are rendered by assuming the environment is temporarily static and simulating the action a. The one-step look-ahead evaluation is commonly used in animation literature [81, 49, 43, 42]. Making a decision with the look-ahead policy takes about 15 milliseconds on average, and the scores are shown in solid black lines in Figure 6.11a. When the environments are sparsely occupied with obstacles, the scores become very close to the optimal ones; when the environments are cluttered, by looking ahead one step, the scores can be greatly improved, but more samples are still required for the controllers to converge toward the optimal behaviors. Finally, the simulation time

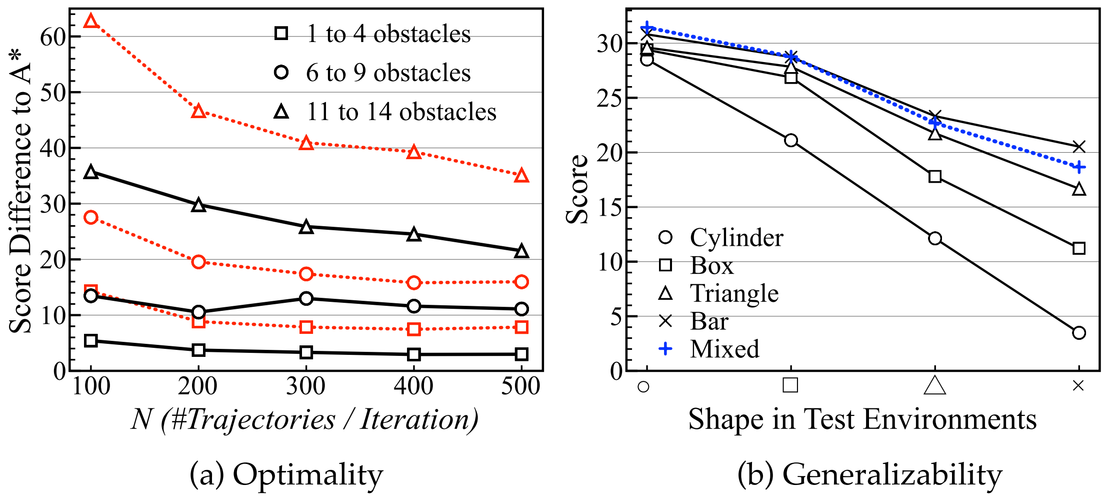


Figure 6.11: Quantification of the optimality and generalizability of the resulting controllers. (a) We analyze the optimality of our controllers by comparing the results to those generated with the A* search algorithm, and plot the difference in the scores. The red dotted lines and the black solid lines denote the controllers without and with one-step look-ahead respectively. (b) We quantify the generalizability of the controllers trained for different specific shapes by evaluating them in environments containing other shapes.

for test sets of 10,000 random environments are reported in Figure 6.12. The A* search produces optimal trajectories, but requires long computation time.

Method	1–4 obstacles	6–9 obstacles	11–14 obstacles
No look-ahead	3.6 min	4.8 min	6.4 min
Look-ahead	19.0 min	26.1 min	38.7 min
A*	732.8 min	1,399.0 min	2,583.4 min

Figure 6.12: Comparison of running-time for simulating 10,000 trajectories in randomly generated environments.

6.5.5 Generalizability

An advantage of using depth perception is that the learned controllers can be directly applied to environments containing arbitrarily-shaped objects, and no preprocessing, such as re-parameterization of the environment, is required. To quantify this property, we generate several controllers, each trained for obstacles of a specific shape, and evaluate the controllers in environments containing obstacles of other shapes. In this experiment, we start with cylinders that are used throughout the previous sections, and then change the shapes into boxes, triangles and bars, as shown in Figure 6.8b. To maintain the density of the environment, the volume of each obstacle is preserved when changed into other shapes. The evaluation results are shown in Figure 6.11b. In general, bars and triangles produce more difficult scenes, because they tend to create wider blocks and narrower passages, while cylinders produce the simplest environments. Although the controller trained for cylinders can still respond to other shapes with positive expected returns, the performance varies a lot with respect to different shapes. On the contrary, the controller trained for bars has the best scores in all shapes. Simpler shapes tend to lead to less challenging environments, such that controllers are exposed to only few samples to learn difficult situations. This explains why the controller trained with bars even outperforms those that are trained with the same shapes used in the test set. This also explains why other controllers have much worse performance in the environment of bars.

Finally, we mix all four shapes to learn a controller, whose scores are plotted in dotted blue line in Figure 6.11b. It has good scores in all tests, even though when compared with the controller trained for bars, it has worse performance in the environments containing only bars. This is due to the fact that the use of all shapes make difficult situations appear less often in the sample set, while the test set of only bars involve more difficult situations. We conclude that using depth perception, the learned controllers can respond to novel shapes, but with limited capability, as the controllers can only infer the situations from experience, i.e., input set of transition tuples of the learning algorithm. In the end, we demonstrate how the controllers trained for bars cope with environments containing arbitrarily-shaped objects in Figure 6.13.

6.5.6 Survival Game

In the end, we use a game-like scenario to highlight the advantage of reinforcement learning over path search techniques in real-time applications. We build a closed environment where all the obstacles move with constant speed in the direction of the character and the only goal of the character is to survive by not colliding with any obstacles. Since the goal is not a concrete

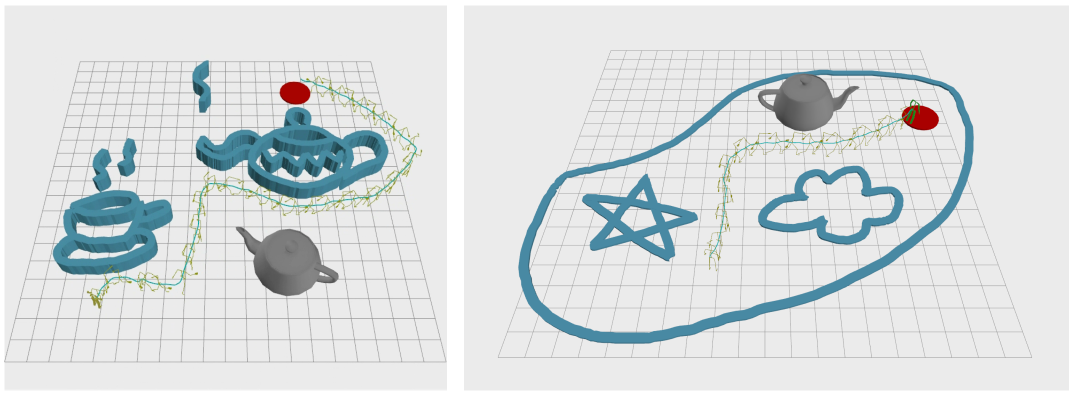


Figure 6.13: We apply the controllers trained for bars in the environments containing arbitrarily-shaped objects. We load in a teapot mesh and ask a user to draw arbitrary obstacles. No collision detection nor re-parametrization of the environment is required, but the controllers can navigate the characters through the obstacles to the goal.

state or position, it is difficult to define a heuristic for path search algorithms like A*. In addition, applying search algorithms in real-time requires collision detection to be performed for each expansion of the search trees, which introduces considerable computation overhead. On the contrary, the scenario can be easily defined in a RL framework: in the learning stage, the character is given a deadly penalty for colliding with obstacles or walls, and a small reward for every surviving moment. This is all that is needed for the character to learn a surviving strategy automatically by interacting with the environment. At run-time, no collision detection nor re-parameterization of the environment is required, but the controller can make decisions instantaneously according to the character's visual perception. Examples of the learned strategy are shown in Figure 6.14. In addition, using the resulting controller, the character can avoid obstacles of arbitrary shape that are drawn interactively by a user.

6.6 Conclusions

We have proposed a method that facilitates the application of reinforcement learning to character control. Traditional state models are manually made up with explicit descriptions of the environment. On the contrary, our state

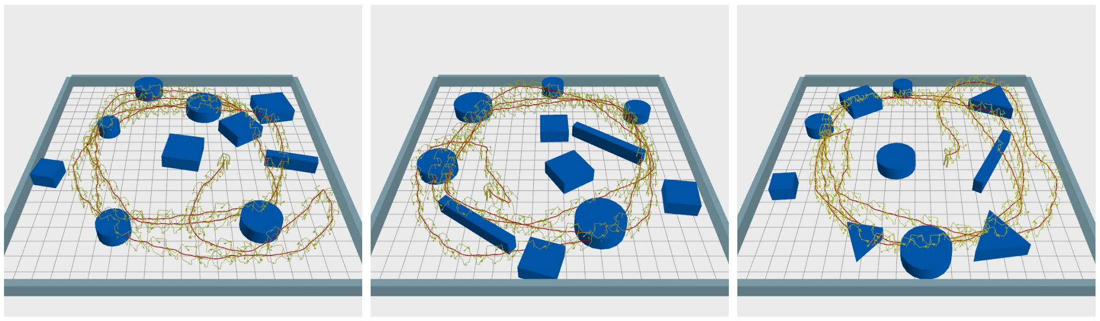


Figure 6.14: Visualization of the learned strategy in the survival game, where all the obstacles track the character and the only goal is to survive by not colliding with any obstacles or walls. The character learns that it is important to quickly move outward through the obstacles, and then circle around the obstacles while avoiding being trapped in a corner. By successfully doing so, the character can continually survive.

model lets the character perceive the environment directly. In order to use a generic high-dimensional sensor like depth vision, it is critical to adapt the dimensionality of the state space to the scene complexity; without adaptation, it is challenging to avoid being cursed by dimensionality. We make such adaptation possible by proposing a hierarchical state model and a novel regression algorithm. Our hierarchical state model replaces a single state definition of fixed dimensionality. Our regression algorithm works with the hierarchical state model to adapt the dimensionality to the scene complexity. We demonstrate that our approach based on adaptive depth perception learns a better controller in a shorter amount of time. The learned controller can be directly applied to environments containing objects of novel shapes, and there is no need for re-parameterizations.

We see numerous directions to further improve our work. First of all, the character only perceives and generalizes the current situation but does not memorize its history. If the environment consists of complex dead ends like in a maze, the character may get trapped and may keep making the same mistakes. Incorporating short-term or long-term memory into the system can resolve this problem. Also, maintaining global information like a scene map could help the character to make better decisions [56, 32, 5]. Another alternative is to model the problem with partially observable MDP (POMDP).

Currently, our motion graph only includes walking motions, because we only capture one-dimensional depth vision from the camera mounted on the character. In order to let the character perform actions like collision-free jumping or ducking, however, two-dimensional depth vision is required, which increases the potential number of dimensions quadratically. Although we adjust the dimensionality of state space adaptively, it will be necessary to draw significantly more samples during learning and require more memory to differentiate complex situations like the width and depth of gaps to jump over. However, we would like to see our work as a first step to explore further research of using adaptive state models in reinforcement learning.

This chapter is based on "Learning Motion Controllers with Adaptive Depth Perception", Wan-Yen Lo, Claude Knaus, and Matthias Zwicker, currently under review.

Chapter 7

Conclusions

“We are at the very beginning of time for the human race. It is not unreasonable that we grapple with problems. But there are tens of thousands of years in the future. Our responsibility is to do what we can, learn what we can, improve the solutions, and pass them on."

What Do You Care What Other People Think?
RICHARD FEYNMAN

In this dissertation, we seek to bridge the gap between real-time motion control and (near) optimal motion synthesis. Both are critical for making a lifelike avatar: the former allows instant responses to user input and the latter ensures realism of the motion. We investigate two most popular data-drive approaches, which make use of motion capture data by piecing together short motion fragments, and point of the inevitable of trading optimality for efficiency. We propose novel algorithms to improve the efficiency for near-optimal motion synthesis, making the data-driven approaches more applicable for practical applications.

We first investigate search algorithms, since the motion capture data is generally organized with a graph structure and the problem of the motion synthesis is cast as a search problem. Among all the search algorithms, A* search is the most favorable, for it is optimal and optimally efficient. However, the search complexity is still exponential to the graph size and the length of the solution, so it is challenging to apply search algorithms for optimal motion synthesis when interactive performance is demanded. To make search algorithms more applicable for interactive applications, we develop a novel method to allow two searches performed simultaneously from both ends of the solution. By cutting the search depth in half, our bidirectional search strategy can lead to a significant performance improvement. This is orthogonal to many search algorithms, so our approach can be applied in existing frameworks to enhance the performance.

We then explore reinforcement learning, which avoids the trade-off between optimality and efficiency by means of pre-planning. Reinforcement learning algorithms allow us to learn in a pre-process the optimal value function, which returns the expected future rewards of every possible state. A motion controller's ability to make optimal decisions then relies on how well the value function is approximated. We introduce a tree-based regression algorithm to adaptively approximate the value function, which is more efficient and robust than previous strategies. We also extend the existing frameworks to include parameterized motions and interpolation for precise motion control. Our approach generate natural animation in real-time while avoiding excessive sampling of the continuous space of motions.

Previous reinforcement learning frameworks require the state space to be carefully designed to limit the dimensionality. This process is tedious, and a small change in the state representation requires the learning process to be repeated. We propose to skip this design phase by letting the character directly "see" the surroundings. We make this possible by introducing a hierarchical state model and a novel regression algorithm to avoid the notorious curse of dimensionality. We equip the character with 1D panoramic depth vision, and our approach allow the resolution of visual percepts to be adapted automatically to the scene complexity. We demonstrate that our controllers allow a character to navigate or survive in environments containing arbitrarily shaped obstacles, which is difficult to achieve with previous reinforcement learning frameworks.

7.1 Future Work

In this dissertation, we use and improve several machine learning and artificial intelligence techniques. Although we mainly focus on motion planning, we see the opportunities of applying our algorithms in other areas for future work:

Bidirectional Search. For problems that are solvable with conventional search algorithms, our bidirectional search framework may be used to improve the search efficiency if a mapping can be found between the state space and the Euclidean space. The mapping is essential as we define a cut and perform dynamic cut adjustment in the Euclidean space to balance two searches from opposite directions.

Hierarchical State Model. Our hierarchical state model can be used to replace a single state definition of fixed dimensionality, and our regression algorithm can automatically select a suitable resolution according to the complexity of the problem. It is critical, however, to define a meaningful hierarchy from the original state representation. The agent should be able to distinguish between favorable and hostile states with low resolution, and to improve decisions with an increase in resolution. Given an ill-defined hierarchy, the regression algorithm will use the finest resolution all the time, introducing additional overheads while not improving the learning efficiency.

We further list a few opportunities for future work in computer animation.

Hierarchical Search. We have shown that by cutting the search depth in half, the search performance can be dramatically improved. However, with limited computing resources, the length of motions that can be generated at interactive rates is still limited. This could be addressed with a hierarchical search algorithm, or by splitting the search space into more than two parts. The main challenge is to design a mechanism to merge several partial solutions efficiently without sacrificing the optimality.

More General RL Framework. In the last decade, many advances have been made to improve the applicability of reinforcement learning in character control: continuous state space, continuous action space, and automatically learned reward functions from given examples. We also made contributions by introducing a more general state model. We would like to see the action space be generalized with a similar concept, so that a bigger, continuous action space can be used to enable fluid and precise control. Finally, it is challenging to allow both state and action space to be continuous. To avoid the curse of dimensionality, a more adaptive algorithm is inevitable.

Bibliography

[1] Okan Arikan and D. A. Forsyth. Interactive motion generation from examples. *ACM Trans. Graph.*, 21:483–490, July 2002.

[2] Okan Arikan, David A. Forsyth, and James F. O'Brien. Motion synthesis from annotations. *ACM Trans. Graph.*, 22:402–408, July 2003.

[3] Jackie Assa, Yaron Caspi, and Daniel Cohen-Or. Action synopsis: pose selection and illustration. *ACM Trans. Graph.*, 24:667–676, July 2005.

[4] Leemon C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *ICML*, pages 30–37, 1995.

[5] Bram Bakker. Reinforcement learning with long short-term memory. In *NIPS*, pages 1475–1482, 2001.

[6] Richard Ernest Bellman. *Dynamic Programming*. Courier Dover Publications, 1957.

[7] Jinxiang Chai and Jessica K. Hodgins. Performance animation from low-dimensional control signals. *ACM Trans. Graph.*, 24:686–696, July 2005.

[8] Jinxiang Chai and Jessica K. Hodgins. Constraint-based motion optimization using a statistical dynamic model. *ACM Trans. Graph.*, 26, July 2007.

[9] Joel E. Chestnutt, Yutaka Takaoka, Keisuke Suga, Koichi Nishiwaki, James Kuffner, and Satoshi Kagami. Biped navigation in rough environments using on-board sensing. In *IROS*, pages 3543–3548, 2009.

[10] Min Gyu Choi, Jehee Lee, and Sung Yong Shin. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Trans. Graph.*, 22:182–203, April 2003.

[11] Myung Geol Choi, Manmyung Kim, Kyunglyul Hyun, and Jehee Lee. Deformable motion: Squeezing into cluttered environments. *Comput. Graph. Forum*, 30(2):445–453, 2011.

[12] Seth Cooper, Aaron Hertzmann, and Zoran Popović. Active learning for real-time motion controllers. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[13] Marco da Silva, Yeuhi Abe, and Jovan Popović. Interactive simulation of stylized human locomotion. *ACM Trans. Graph.*, 27:82:1–82:10, August 2008.

[14] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[15] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *J. Mach. Learn. Res.*, 6:503–556, December 2005.

[16] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[17] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

[18] Michael Gleicher. Motion path editing. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, pages 195–202, New York, NY, USA, 2001. ACM.

[19] Keith Grochow, Steven L. Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *ACM Trans. Graph.*, 23:522–531, August 2004.

[20] Rachel Heck and Michael Gleicher. Parametric motion graphs. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 129–136, New York, NY, USA, 2007. ACM.

[21] Kenneth E. Hoff, III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 277–286, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[22] Leslie Ikemoto, Okan Arikan, and David Forsyth. Learning to move autonomously in a hostile world. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[23] Sébastien Jodogne and Justus H. Piater. Interactive learning of mappings from visual percepts to actions. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 393–400, New York, NY, USA, 2005. ACM.

[24] Sébastien Jodogne and Justus H. Piater. Closed-loop learning of visual control policies. *J. Artif. Int. Res.*, 28:349–391, March 2007.

[25] Michael Patrick Johnson, Andrew Wilson, Bruce Blumberg, Christopher Kline, and Aaron Bobick. Sympathetic interfaces: using a plush toy to direct synthetic characters. In *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, CHI '99, pages 152–158, New York, NY, USA, 1999. ACM.

[26] James J. Kuffner Jr. and Steven M. LaValle. Rrt-connect: An efficient approach to single-query path planning. In *ICRA*, pages 995–1001, 2000.

[27] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *J. Artif. Intell. Res. (JAIR)*, 4:237–285, 1996.

[28] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[29] Lucas Kovar and Michael Gleicher. Flexible automatic motion blending with registration curves. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 214–224, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[30] Lucas Kovar and Michael Gleicher. Automated extraction and parameterization of motions in large data sets. *ACM Trans. Graph.*, 23:559–568, August 2004.

[31] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. *ACM Trans. Graph.*, 21:473–482, July 2002.

[32] James J. Kuffner and Jean-Claude Latombe. Fast synthetic vision, memory, and learning models for virtual humans. In *CA*, pages 118–127, 1999.

[33] James B.H. Kwa. Bs*: an admissible bidirectional staged heuristic search algorithm. *Artif. Intell.*, 38(1):95–109, 1989.

[34] Manfred Lau, Ziv Bar-Joseph, and James Kuffner. Modeling spatial and temporal variation in motion data. *ACM Trans. Graph.*, 28:171:1–171:10, December 2009.

[35] Manfred Lau and James J. Kuffner. Behavior planning for character animation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pages 271–280, New York, NY, USA, 2005. ACM.

[36] Manfred Lau and James J. Kuffner. Precomputed search trees: planning for interactive goal-driven animation. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '06, pages 299–308, Aire-la-Ville, Switzerland, 2006. Eurographics Association.

[37] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006.

[38] Jehee Lee, Jinxiang Chai, Paul S. A. Reitsma, Jessica K. Hodgins, and Nancy S. Pollard. Interactive control of avatars animated with human motion data. *ACM Trans. Graph.*, 21:491–500, July 2002.

[39] Jehee Lee and Kang Hoon Lee. Precomputing avatar behavior from human motion data. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '04, pages 79–87, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[40] Jehee Lee and Sung Yong Shin. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 39–48, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[41] Kang Hoon Lee, Myung Geol Choi, and Jehee Lee. Motion patches: building blocks for virtual environments annotated with motion data. *ACM Trans. Graph.*, 25:898–906, July 2006.

[42] Seong Jae Lee and Zoran Popović. Learning behavior styles with inverse reinforcement learning. *ACM Trans. Graph.*, 29:122:1–122:7, July 2010.

[43] Yongjoon Lee, Seong Jae Lee, and Zoran Popović. Compact character controllers. *ACM Trans. Graph.*, 28:169:1–169:8, December 2009.

[44] Yongjoon Lee, Kevin Wampler, Gilbert Bernstein, Jovan Popović, and Zoran Popović. Motion fields for interactive character locomotion. *ACM Trans. Graph.*, 29:138:1–138:8, December 2010.

[45] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Trans. Graph.*, 25:579–588, July 2006.

[46] Sergey Levine, Yongjoon Lee, Vladlen Koltun, and Zoran Popović. Space-time planning with parameterized locomotion controllers. *ACM Trans. Graph.*, 30:23:1–23:11, May 2011.

[47] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. Ara*: Anytime a* with provable bounds on sub-optimality. In *NIPS*, 2003.

[48] C. Karen Liu, Aaron Hertzmann, and Zoran Popović. Learning physics-based motion style with nonlinear inverse optimization. *ACM Trans. Graph.*, 24:1071–1081, July 2005.

[49] Wan-Yen Lo and Matthias Zwicker. Real-time planning for parameterized human motion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 29–38. Eurographics Association, 2008.

[50] James McCann and Nancy Pollard. Responsive characters from motion fragments. *ACM Trans. Graph.*, 26, July 2007.

[51] Philipp Michel, Joel Chestnutt, James Kuffner, and Takeo Kanade. Vision-guided humanoid footstep planning for dynamic environments. In *in Proc. of the IEEE-RAS/RSJ Int. Conf. on Humanoid Robots (HumanoidsŠ05)*, pages 13–18, 2005.

[52] Jeff Michels, Ashutosh Saxena, and Andrew Y. Ng. High speed obstacle avoidance using monocular vision and reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pages 593–600, New York, NY, USA, 2005. ACM.

[53] Jianyuan Min, Yen-Lin Chen, and Jinxiang Chai. Interactive generation of human animation with deformable motion models. *ACM Trans. Graph.*, 29:9:1–9:12, December 2009.

[54] Uldarico Muico, Yongjoon Lee, Jovan Popović, and Zoran Popović. Contact-aware nonlinear control of dynamic characters. *ACM Trans. Graph.*, 28:81:1–81:9, July 2009.

[55] Tomohiko Mukai and Shigeru Kuriyama. Geostatistical motion interpolation. *ACM Trans. Graph.*, 24:1062–1070, July 2005.

[56] Hansrudi Noser, Olivier Renault, Daniel Thalmann, and Nadia Magnenat-Thalmann. Navigation for digital actors based on synthetic vision, memory, and learning. *Computers & Graphics*, 19(1):7–19, 1995.

[57] Naoki Numaguchi, Atsushi Nakazawa, Takaaki Shiratori, and Jessica K. Hodgins. A puppet interface for retrieval of motion capture data. In *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '11, pages 157–166, New York, NY, USA, 2011. ACM.

[58] Jan Ondřej, Julien Pettré, Anne-Hélène Olivier, and Stéphane Donikian. A synthetic-vision based steering approach for crowd simulation. *ACM Trans. Graph.*, 29:123:1–123:9, July 2010.

[59] Dirk Ormoneit and Śaunak Sen. Kernel-based reinforcement learning. *Mach. Learn.*, 49:161–178, November 2002.

[60] Masaki Oshita. Motion control with strokes. *Journal of Visualization and Computer Animation*, 16(3-4):237–244, 2005.

[61] Christopher Peters and Carol O'Sullivan. Bottom-up visual attention for virtual human animation. In *CASA*, pages 111–117, 2003.

[62] Ira Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.

[63] Katherine Pullen and Christoph Bregler. Motion capture assisted animation: texturing and synthesis. *ACM Trans. Graph.*, 21:501–508, July 2002.

[64] Liu Ren, Gregory Shakhnarovich, Jessica K. Hodgins, Hanspeter Pfister, and Paul Viola. Learning silhouette features for control of human motion. *ACM Trans. Graph.*, 24:1303–1331, October 2005.

[65] Charles Rose, Michael F. Cohen, and Bobby Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications*, 18(5):32–40, 1998.

[66] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[67] Alla Safonova and Jessica K. Hodgins. Analyzing the physical correctness of interpolated human motion. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '05, pages 171–180, New York, NY, USA, 2005. ACM.

[68] Alla Safonova and Jessica K. Hodgins. Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph.*, 26, July 2007.

[69] Arno Schödl, Richard Szeliski, David H. Salesin, and Irfan Essa. Video textures. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 489–498, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[70] Ari Shapiro, Marcelo Kallmann, and Petros Faloutsos. Interactive motion correction and object manipulation. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 137–144, New York, NY, USA, 2007. ACM.

[71] Hyun Joon Shin and Hyun Seok Oh. Fat graphs: constructing an interactive character with continuous controls. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '06, pages 291–298, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[72] Takaaki Shiratori and Jessica K. Hodgins. Accelerometer-based user interfaces for the control of a physically simulated character. *ACM Trans. Graph.*, 27:123:1–123:9, December 2008.

[73] Hubert P. H. Shum, Taku Komura, and Shuntaro Yamazaki. Simulating interactions of avatars in high dimensional state space. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 131–138, New York, NY, USA, 2008. ACM.

[74] Ronit Slyper and Jessica K. Hodgins. Action capture with accelerometers. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 193–199, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[75] Kwang Won Sok, Manmyung Kim, and Jehee Lee. Simulating biped behaviors from human motion data. *ACM Trans. Graph.*, 26, July 2007.

[76] Nathan Sprague, Dana Ballard, and Al Robinson. Modeling embodied visual behaviors. *ACM Trans. Appl. Percept.*, 4, July 2007.

[77] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[78] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomeranets, and Markus Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proc. VMV*, pages 47–54, 2003.

[79] Matthew Thorne, David Burke, and Michiel van de Panne. Motion doodles: an interface for sketching character motion. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[80] Deepak Tolani and Norman I. Badler. Real-time inverse kinematics of the human arm. *Presence*, 5(4):393–401, 1996.

[81] Adrien Treuille, Yongjoon Lee, and Zoran Popović. Near-optimal character animation with continuous control. *ACM Trans. Graph.*, 26, July 2007.

[82] Kevin Wampler, Erik Andersen, Evan Herbst, Yongjoon Lee, and Zoran Popović. Character animation in two-player adversarial games. *ACM Trans. Graph.*, 29:26:1–26:13, July 2010.

[83] Jack M. Wang, David J. Fleet, and Aaron Hertzmann. Gaussian process dynamical models for human motion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(2):283–298, 2008.

[84] Jing Wang and Bobby Bodenheimer. Computing the duration of motion transitions: an empirical approach. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '04, pages 335–344, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[85] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992.

[86] Douglas J. Wiley and James K. Hahn. Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Applications*, 17(6):39–45, 1997.

[87] Andrew Witkin and Zoran Popovic. Motion warping. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 105–108, New York, NY, USA, 1995. ACM.

[88] Yuting Ye and C. Karen Liu. Optimal feedback control for character animation using an abstract model. *ACM Trans. Graph.*, 29:74:1–74:9, July 2010.

[89] Yuting Ye and C. Karen Liu. Synthesis of responsive motion using a dynamic model. *Comput. Graph. Forum*, 29(2):555–562, 2010.

[90] KangKang Yin and Dinesh K. Pai. Footsee: an interactive animation system. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 329–338, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[91] Wataru Yoshizaki, Yuta Sugiura, Albert C. Chiou, Sunao Hashimoto, Masahiko Inami, Takeo Igarashi, Yoshiaki Akazawa, Katsuaki Kawachi, Satoshi Kagami, and Masaaki Mochimaru. An actuated physical puppet as an input device for controlling a digital manikin. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 637–646, New York, NY, USA, 2011. ACM.

[92] Liming Zhao, Aline Normoyle, Sanjeev Khanna, and Alla Safonova. Automatic construction of a minimum size motion graph. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pages 27–35, New York, NY, USA, 2009. ACM.

[93] Liming Zhao and Alla Safonova. Achieving good connectivity in motion graphs. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 127–136, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[94] Victor Brian Zordan, Anna Majkowska, Bill Chiu, and Matthew Fast. Dynamic response for motion capture animation. *ACM Trans. Graph.*, 24:697–701, July 2005.