Modeling Behavior and Variation for Crowd Animation

Manfred Chung Man Lau

CMU-CS-09-148
August 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
James Kuffner, Chair
Nancy Pollard
Ziv Bar-Joseph
Ming Lin, University of North Carolina at Chapel Hill

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

Copyright © 2009 Manfred Chung Man Lau

This research was sponsored by the Air Force Research Laboratory under cooperative agreement number F30602-01-2-0569 and the National Science Foundation under grant numbers IIS-02052247, EIA-0196217, ECS-0325383, and REC-9979894. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Abstract

The simulation of crowds of virtual characters is needed for applications such as films, games, and virtual reality environments. These simulations are difficult due to the large number of characters to be simulated and the requirement for synthesizing realistic human-like motion efficiently. This thesis focuses on two problems: how to search through and select motion clips of behaviors so that human-like motion can be generated for multiple characters interactively, and how to model and synthesize variation in motion data.

Given a collection of blendable segmented motion clips derived from motion capture or keyframed animation, this thesis explores novel ways to apply heuristic search algorithms to generate goal-driven navigation motion for virtual human-like characters. Motion clips are organized and interconnected through a behavior graph that encodes the possible actions of a character. A planning approach is used to search over these possible actions to efficiently generate motion. This technique works well for synthesizing animations of multiple characters navigating autonomously in large dynamic environments.

In addition, this thesis introduces a novel planning approach based on precomputation that is more efficient than traditional forward search methods. We present a technique for precomputing large and diverse trees, and describe a backward search method used during runtime to solve planning queries. This new approach allows us to develop an interactive animation system that supports a large number of characters simultaneously.

Finally, this thesis addresses the issue of motion variation. Current state-of-the-art crowd simulations often use a few specific motion clips or repeated cycles of a particular motion to continuously animate multiple characters. The idea of synthesizing the subtle variations in motion data has been largely unexplored, as previous work considers variation to be an additive noise component. This thesis instead uses a data-driven approach and applies learning techniques to this problem. Given a small number of input motions, we model the data with a Dynamic Bayesian Network, and synthesize new spatial and temporal variants that are statistically similar to the inputs.

Acknowledgments

I thank my advisor, James Kuffner, for his support, advice and encouragement during the past six years. Thanks to my committee members, Nancy Pollard, Ziv Bar-Joseph, and Ming Lin for their advice and comments about my work. This thesis would not have been the same without all of your input.

Thanks to the faculty and students whom I have interacted with during my time here: Jernej Barbic, Jinxiang Chai, Joel Chestnutt, James Hays, Jessica Hodgins, Doug James, Jim McCann, Philipp Michel, Paul Reitsma, Liu Ren, Alla Safonova, FunShing Sin, Ronit Slyper, Kwang Won Sok, Mike Stilmann, Chris Twigg and many others. Thanks to the memories and experiences that I have learned from each of you. Special thanks to Stephen Brookes and Jeff Stylos for helping me throughout my speaking skills talks. Your advice has improved every talk I have since given, and will continue to affect each talk I give from now on. I also anonymously thank all the teachers and students who have influenced my life throughout my education.

Finally, I thank my grandmother, my parents and my brother for their support. There are no words that can possibly describe my thanks to them here.

Contents

1 Introduction **1**
- 1.1 Behavior Planning 4
- 1.2 Precomputed Search Trees 5
- 1.3 Modeling Spatial and Temporal Variants in Motion Data 7

2 Related Work **11**
- 2.1 Crowd Animation 11
- 2.2 Motion Planning Methods for Animation 14
- 2.3 Precomputation Methods for Animation 18
- 2.4 Modeling and Synthesizing Variation 23

3 Behavior Planning **29**
- 3.1 Problem Statement and Overview 31
- 3.2 Behavior Graph 32
- 3.3 Environment Abstraction 34
- 3.4 Behavior Planner 35
- 3.5 Motion Generation and Blending 37
- 3.6 Results 37
- 3.7 Discussion 38

4 Precomputed Search Trees **41**
- 4.1 Problem Statement and Overview 42
- 4.2 Precomputation of Tree 44
  - 4.2.1 Exhaustive Tree 44
  - 4.2.2 Pruned Tree 45
  - 4.2.3 Scalable and Diverse Randomized-based Tree 46
- 4.3 Precomputation of Environment and Goal Gridmaps 50
- 4.4 Mapping Obstacles to Environment Gridmap 52
- 4.5 Runtime Backward Search 53
- 4.6 Coarse-Level Planner 58
- 4.7 Evaluation 59
  - 4.7.1 Interactive System 59
  - 4.7.2 Comparison of Tree Precomputation Methods 61

4.7.3 Comparison between Precomputation Approach and Traditional Forward Search . . . 64
4.7.4 Effect of Grid Resolution . . . 66
4.8 Discussion . . . 67

5 Modeling Spatial and Temporal Variants in Motion Data 69
5.1 Problem Statement and Overview . . . 71
5.2 Dynamic Bayesian Network . . . 72
5.3 Structure Learning . . . 73
5.3.1 Structure Search . . . 74
5.3.2 Non-Parametric Regression for Computing Conditional Distribution . . . 78
5.4 Synthesis of New Variants . . . 80
5.5 Constraints . . . 80
5.6 Evaluation . . . 82
5.6.1 Results for Full-body Human Animation . . . 83
5.6.2 Memory and Performance Time . . . 84
5.6.3 User Study . . . 85
5.6.4 Experiments with Adding Noise . . . 86
5.7 Inputs that work well with Our Approach . . . 88
5.8 Discussion . . . 90

6 Putting it All Together 91

7 Discussion and Future Work 95

Bibliography 99

List of Figures

1.1 Films and games are applications that motivate our work. **Left:** A scene from the animated film *Madagascar*. **Right:** A screenshot from *The Sims* computer game. 2

1.2 **Left:** A simple behavior graph of high-level motions used in our *Behavior Planning* approach. **Right:** Planned behaviors for 100 animated characters navigating in a complex environment. 4

1.3 **Left:** An example of a precomputed search tree. This is a frequency plot; each point represents the number of paths that can reach that point from the root of the tree. The root is near the middle of the figure and the tree progresses in a forward direction (or up in the figure). This tree has about 220,000 nodes and requires a storage memory of 10 MB. **Right:** Screenshot of our interactive system. The characters respond to user changes interactively while navigating in large and dynamic environments. 6

1.4 A DBN for the variables $X_1, ..., X_n$. Each node X_i represents one DOF in the motion data. We use the prior network to model the first 2 frames of each input motion clip. The transition network then models subsequent frames given the previous 2 frames. We assume a 2nd-order Markov property because it is the simplest model that works well. 8

3.1 The range of planning configuration spaces for generating the motions of a human-like character. Our *Behavior Planning* approach differs from previous methods in that it lies in between the two extremes in this range. 30

3.2 *Left:* The problem inputs include a description of the environment, a starting position (larger green dot) and orientation (smaller green dot points toward the direction), and a goal position (red dot). *Right:* The output is a motion sequence. 32

3.3 *Left:* A simple graph of behaviors. Each node or behavior contains a set of example motion clips for that behavior. Each edge indicates allowable transitions between behaviors. *Right:* An example graph used for a human character that includes special jumping and crawling behaviors. 33

3.4 A dynamic environment with a falling tree. *Left:* Before it falls, the characters are free to jog normally in the open space. *Center:* As it is falling, the characters can neither jog past nor jump over it. *Right:* After it has fallen, the characters can jump over it. 38

3.5 The characters avoid each other and the dynamic obstacles (spheres). 38

4.1 Overview of the system. 43

4.2 Frequency plot of the precomputed tree. Each point represents the number of paths that can reach that point from the root of the tree. The root is near the middle of each figure and the tree progresses in a forward direction (or up in the figure). The tree covers an area that is approximately a half circle of radius 16 meters, with the character starting at the center of the half circle. The majority of paths end up in an area between 8 and 14 meters away from the start. We used about 1,500 frames of motion at 30 Hz. **Left:** Exhaustive tree of 6 depth levels built from graph with 21 behavior nodes. This tree has over 6 million nodes (over 300 MB). **Right:** The pruned tree has 220,000 nodes (about 10 MB). 45

4.3 **Left:** An *environment gridmap* initialized with *UNOCCUPIED* cells. The intuition for this gridmap is that if cell (x_i, y_i) is occupied by an obstacle, the tree nodes corresponding to this cell and their descendant nodes (the black ones) are *BLOCKED*. **Right:** For each node i, we precompute and store the corresponding values x_i, y_i, $x_{midtime_i}$, and $y_{midtime_i}$. 51

4.4 **Left:** In the *goal gridmap*, each cell contains a sorted list of paths. Each path's total cost is the sum of the cost of the motion states. The sorting is based on this total cost. Since each node in the tree corresponds to a unique path if we trace the node back towards the root of the tree, we can also say that each goal cell contains a sorted list of nodes. We will use this gridmap during runtime; the intuition is that if we know the gridcell that the goal position is in, the paths or nodes in that cell correspond to the potential solutions. **Right:** A straightforward discretization of the *goal gridmap* may not work well. An "overlapped discretization" works well. 52

4.5 **Left:** We align the coordinate spaces between the environment and the tree. We translate and rotate the obstacles and the goal position so that the starting position and orientation (of the character in the environment) match with that of the precomputed tree. **Right:** If the size of the gridcell is d, we can guarantee that the mapping of an obstacle to the environment gridmap is correct if the sampling of points for the obstacle is at most $d/\sqrt{2}$ apart. 53

4.6 The 2 columns correspond to the first 2 iterations of the runtime path finding phase for this example. The top row shows the start (green sphere) in each iteration, and the sub-goal (red sphere) selected from the coarse-level path. The bottom row shows the path returned by the runtime path finding algorithm (light and dark blue) and the partial path chosen (dark blue only). An estimate of the outline of the precomputed tree is shown. The tree is transformed to the global space only in the figure to show how it relates to the other parts of the environment. There is only one precomputed tree, and it is never transformed to the global space in the algorithm. 55

4.7 The process of tracing back the list of sorted nodes P towards the root node in Algorithm 3. **Left:** The 3 cases under which each trace of node p stops. The sub-goal is inside the dashed square (a cell of the goal gridmap). **Right:** Simple example. The blue nodes are the nodes of the precomputed tree. The sub-goal is somewhere in the square-shaped box of red nodes. The other colored nodes correspond to the 3 cases. 56

4.8 The points of the path that are eventually chosen by the coarse-level planner for this environment. 59

4.9 Screenshot of the interactive system. The characters interactively respond to user changes to obstacles and their respective goal locations while navigating in a large environment. 61

4.10 Examples of precomputed trees used in our comparison. All trees have the same number (826) of nodes. Each tree's root is at (0,0), and the paths move in a forward (or up in the figure) direction because the input actions/motions allow the character to move forward and/or slightly turn left/right. Note that many paths overlap because of the tree's structure. 62

5.1 A DBN for the variables $X_1, ..., X_n$. Each node X_i represents one DOF in the motion data. We use the prior network to model the first 2 frames. The transition network then models subsequent frames given the previous 2 frames. We assume a second-order Markov property because it is the simplest model that works well. 73

5.2 When learning the structure for the transition network, we do a cross validation over each motion sequence. We take each sequence as testing data, and use the others as training data. For the testing sequence, we take the first two frames as input and re-synthesize the whole sequence with the given structure. The newly synthesized sequence is then compared to the original data to evaluate the structure. This is what we compute intuitively in the scoring function for the transition network of the DBN. 77

5.3 We "unroll" the DBN from Figure 5.1 to synthesize new variants. We show here the unrolled network for 5 time frames. Note that the first two frames come from the prior network of the DBN and may not contain cycles. Since the DBN represents a joint probability distribution over the possible trajectories of each DOF, we sample from this distribution to generate new variants. It is important to recognize that the synthesized motion does not have a one-to-one correspondence with any one of the input motions. This means that the synthesized motion is not just a copy of one of the input motions plus some slight differences, but the timing of the whole motion itself is different. Furthermore, no new pose is exactly the same as any previous pose. 81

5.4 Results for cheering, walk cycle, and swimming motion. In each column, the top image shows the 4 inputs (overlapped, each with different color) and the bottom image shows the 15 outputs (overlapped, each with different color). These are frames from the animations. 83

5.5 Given the learned structure and just one jumping motion as inputs, we synthesize four new variant motions. We overlap poses from these four new motions at similar time phases of the jump. We can see the variations in the poses at these time phases. The poses for the head vary the least because the head poses also vary the least in the input data. 84
5.6 Plots of four inputs (in blue) and fifteen output variants (in black or green) for cheering motion. Each curve represents one motion clip. Note that these motions are not cyclic. Left Column: Two selected plots of DOF vs. time. Middle Column: Two selected plots of DOF vs. DOF. Right Column: Two selected plots of PCA-dimension vs. PCA-dimension. 85
5.7 Left: Example frame from walk cycle motion clip with strawman noise added to left shoulder/arm. The left shoulder turns in a way that does not synchronizes with the right arm. Right: Example frame from walk cycle motion clip with Perlin noise added to right hip/knee. The right hip/knee pause and move in a way that do not synchronize with the rest of the walk cycle. In both cases, the unnaturalness of the timing of the whole walk cycle can be better seen in the animations. 87
5.8 Two examples of motion sets that do not work with our approach. There are five overlapped walk cycles in each case. Four of them are inputs (in blue) that are similar, and the other one (in magenta) does not fit together with these four. Left: For the one that does not fit, the arms swing higher than the other four. Right: For the one that does not fit, the motion turns slightly to the right. 88
5.9 Left: Plot of frequency versus likelihoods for the training data. Right: We started with a testing set of eight walk cycles, and our method selected these five to be similar to the ones in the training set. 89

6.1 Left: Example frame of resulting animation generated with five original motion clips. Right: Example frame of resulting animation generated with thirty-two variants together with the five original clips. 93

List of Tables

4.1 Comparison of Tree Precomputation Methods. 62
4.2 Comparison between Precomputation Approach and A*-search Methods. Top set of results: from random planning queries. Bottom set: from C-shaped obstacle case. 65
4.3 Effect of grid resolution on runtime cost and success rate. 66

Chapter 1

Introduction

There is a great need to create crowd animations in computer-simulated environments. Films and games are two main applications that often have scenes with a large number of characters. Figure 1.1 shows some example applications that motivate our work. Animated films such as *The Lord of the Rings* and *Madagascar* [99] require the use of crowd animation systems to autonomously generate the motions for hundreds of characters. Games such as *The Sims* require autonomous control of a large number of characters in real-time.

An ideal animation system should be able to simulate a large number of virtual characters that behave like a large group of humans in the real world. We would like each character to act in its unique way and move about with its own purpose. They need to move and interact with the environment and other characters in a seemingly intelligent way. Each character needs to move around and avoid obstacles automatically. They should exhibit complex motions in large and dynamic environments. The synthesis process needs to be efficient enough such that a user can interact with these characters in real-time. In addition, the characters needs to exhibit variety in many ways. Each character must behave in its unique style and move with its own speed. Each motion should look different even if the same action is performed repeatedly. It would be realistic to have variation in the body and facial features of each character, in how the characters are dressed, and in how their hair are styled. We want all these aspects of a character to look natural. Ideally, we would have efficient algorithms to generate these features.

It would be great if we have crowd animation systems that have all of the above properties. However, this has not been fully achieved. While many crowd animation systems can generate motion for characters that can walk on flat ground, it is often difficult to generate more complex motion with them. For example, in a complex environment with dynamic obstacles and multiple characters navigating in it, it is difficult to assure that the characters avoid the obstacles and each other. Current crowds and game systems sometimes accept slight collisions between the

Figure 1.1: Films and games are applications that motivate our work. **Left:** A scene from the animated film *Madagascar*. **Right:** A screenshot from *The Sims* computer game.

characters as a tradeoff for runtime efficency. As the number of characters increase, it becomes computationally more difficult to guarantee that the characters would not collide with each other. In addition, crowd systems often use steering methods that employ a number of local rules to animate a group of characters. While they work well for simple two-dimensional characters, directly applying these methods to produce motion for human-like characters can lead to jittery artifacts. Although more sophisticated methods might be used to handle this issue, these methods can be time-consuming if we were to use them for a large number of characters.

Crowd animations can also look unnatural because they often use a pre-specified set of cyclic motions. For example, a walk cycle is a motion clip consisting of two steps of walking for a human-like character. To simulate a crowd of characters [66], a few of these walk cycles might be used repeatedly for all the characters and all of the synthesized walking cycles. Similary, in crowd animations for films, an animator typically has a library of cyclic motions from which to generate the motion for all the characters [99]. These cyclic motions lead to synthesized animations that look monotonous and unrealistic. In applications such as games, the same motion clips are usually replayed when they are needed. When a character needs to perform a football throw, for example, a specific motion clip is found and then replayed. Since there may not be many clips available for each specific motion, it will become apparent that the generated motions are repetitive.

Given that the existing animation systems still lack many desirable properties, we want to achieve the following goals in our work:

- Generate motions that make the characters human-like and intelligent: we want to synthesize navigation motion for multiple human-like characters avoiding obstacles and each

other in large and dynamic environments.

- Generate these motions efficiently: create a multiple-character navigation framework where the characters can interactively react to user changes to the environment.

- Synthesize variations in these motions: learn a variation model from example motion data, and be able to synthesize variations of the data that retain features of the original examples but are not exact copies of them.

In this thesis, we describe three approaches that we have developed to achieve these goals. The three sections in this chapter briefly summarize each of these approaches. Our main **contributions** are:

- A planning approach that applies heuristic search methods to efficiently generate goal-driven navigation motion for virtual human-like characters. Compared to methods that use large data sets of motion, we show that we can use a small set of segmented motion clips to generate motions for a large number of characters navigating simultaneously in dynamic environments.

- A novel precomputation-based approach to use human motion data to generate navigation motion: we first precompute a search tree of possible motion paths with the data, and then use a backward search method during runtime to solve planning queries. We show that our approach is more than two orders of magnitude faster than traditional forward search methods such as A*-search.

- We present a technique for precomputing large diverse trees, and explore the advantages and disadvantages of our method compared to previous methods for building diverse trees.

- We study the problem of generating variation in motion data. Instead of considering variation as an additive noise component, we take a data-driven approach and apply learning techniques to this problem. We show that we can use Dynamic Bayesian Networks to synthesize an unlimited number of variants automatically. This process does not require manual parameter tuning and is not tedious compared to the major previous approach of adding noise.

- We show that we can use our method to model and synthesize variation for many types of human motion data. Our model takes a small number of input motions, and synthesizes spatial and temporal variants that retain original features of the inputs but are not exact copies of them. Our approach is novel in that there is no previous automated method that can generate such variants for human motion data.

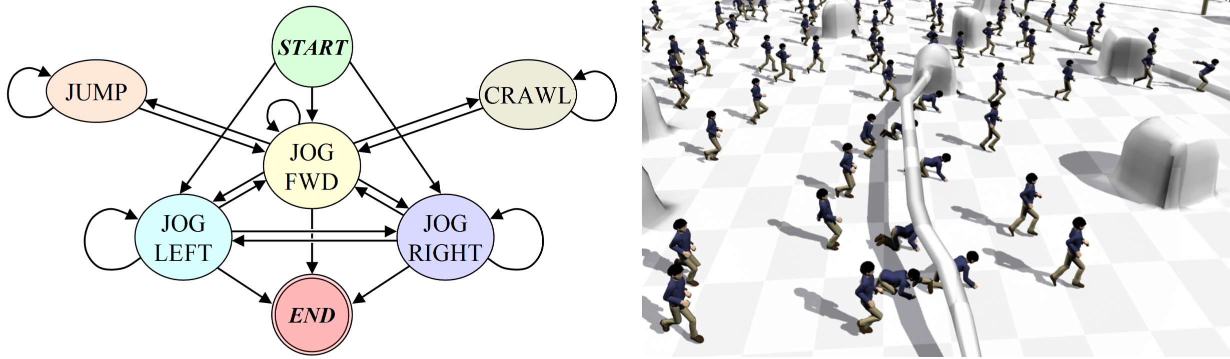


Figure 1.2: **Left:** A simple behavior graph of high-level motions used in our *Behavior Planning* approach. **Right:** Planned behaviors for 100 animated characters navigating in a complex environment.

1.1 Behavior Planning

We call our planning approach *Behavior Planning*. We build behavior-based graphs to represent our motions, and automatically generate sequences of motions of characters navigating in complex and dynamic environments [52]. The inputs are a starting location, a goal location, an environment description, and a behavior graph. The algorithm generates a sequence of motions that allow the character to navigate from the start to the goal.

This approach is well suited for creating long sequences of motions. For example, if a character has to navigate through a terrain with obstacles it has to jump over or duck under, our approach can easily create a sequence of actions that is natural and makes logical sense. We can apply our technique to generate the motions for the characters in crowds and games. This is a difficult problem because it is not trivial to generate natural and collision-free motions for a large number of characters automatically and efficiently.

Our main ideas are to abstract motions as high-level actions, build a behavior graph of these actions, and perform a global search of this graph to find a solution. Figure 1.2 (left) shows an example of a behavior graph. Our approach can generate intuitive sequences of actions such as: walk to the fence, duck under it, jog forward towards the stream and then jump over it. We show results of up to 100 characters navigating in large and complex environments (Figure 1.2 right).

Additional value over previous work: We view this approach as one motion planning method among a spectrum of methods (Figure 3.1). Our approach differs from previous methods in that it has a carefully chosen planning space. **Assumptions/Limitations:** We assume that we are given a set of segmented and blendable motion clips as input. The output sequence is limited

to be a concatenation of these input clips. **Insights:** The abstraction of motions as high-level behaviors and the carefully chosen planning space lead to both the strengths and weaknesses of the approach. The weakness is that we require segmented and blendable motion clips corresponding to the high-level behaviors. The strength is that since the number of these behaviors and input clips are small, the search algorithm is efficient and works well for generating navigation motions. In addition, we empirically found that using an extremely small data set is enough for generating many types of navigation motions.

1.2 Precomputed Search Trees

Runtime efficiency is particularly important for games, virtual reality applications, and interactive simulations. In many interactive systems, the AI that controls the autonomous characters is often limited by the computation time that is available. It may only have a small fraction of a second to decide what the characters should do. This leads to algorithms that generate simple scripted behaviors; more complex behaviors are not possible due to the time constraints. Moreover, the time constraints are worse if there are a large number of characters. This motivates our investigation of the aspects of the planning and motion synthesis processes that can be precomputed in order to achieve a faster runtime.

We argue that it is possible to tradeoff memory for speed: by computing and storing as much information as possible beforehand to allow a faster runtime search. We have demonstrated this concept for human motion data [53]. Our method allows the runtime to be efficient while also keeping the memory requirement to a reasonable amount.

The key idea is to precompute search trees (Figure 1.3 left) of motion clips that can be applied to arbitrary environments. Instead of solving the usual planning problem with one start position and one goal position for each character, we first ignore the obstacles and goal position in the environment. We precompute a tree of all the reachable points of the character given existing motion clips. We then use this tree along with a runtime backward search method to solve planning queries for any configuration of obstacles and goal position.

For distant goal positions, we first use a fast coarse-level planner to generate a rough path of intermediate sub-goals to guide each iteration of the runtime backward search phase. While the use of a coarse-level planner for handling distant goal positions is one possible option, it may not be beneficial in some cases. For example, depending on the discretization of the coarse planner, it is possible that it will return no solution even though a solution exists. We therefore explore methods for precomputing larger trees so that it is not necessary to use the coarse-level planner.

We originally built exhaustive trees of five to six depth levels. We can use these trees to show

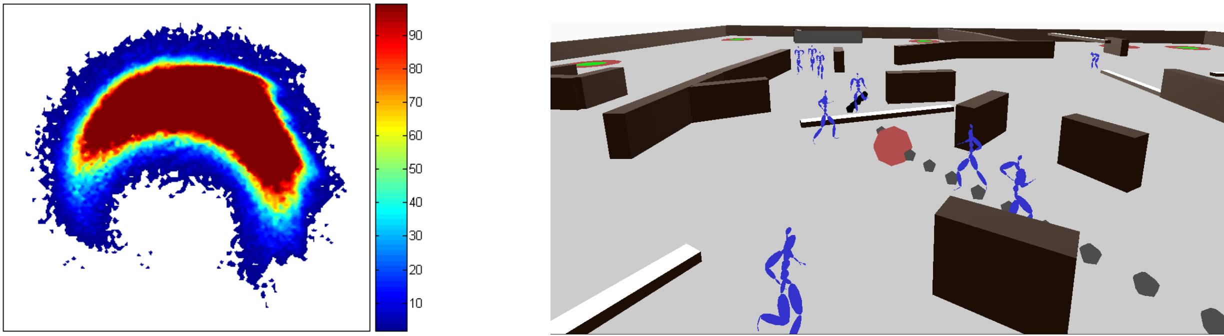


Figure 1.3: **Left:** An example of a precomputed search tree. This is a frequency plot; each point represents the number of paths that can reach that point from the root of the tree. The root is near the middle of the figure and the tree progresses in a forward direction (or up in the figure). This tree has about 220,000 nodes and requires a storage memory of 10 MB. **Right:** Screenshot of our interactive system. The characters respond to user changes interactively while navigating in large and dynamic environments.

that the concept of precomputation works. However, the path lengths in these trees are too small and we cannot use them to solve more practical planning problems that require solutions of up to fifty depth levels.

Since the exhaustive tree requires memory of about 1 GB to store the precomputed information, we then experimented with a simple method to select subsets of paths from the exhaustive tree. We call the selected subset a pruned tree. We found that we can use about 10 MB to store the pruned tree, and we can use it to find solution paths that are similar to the ones from the exhaustive tree.

However, the pruned trees still have small path lengths. This motivates us to build larger and more general trees. In addition, we want to build trees that have diverse paths, which intuitively means that the paths should evenly cover the region that they are in. The purpose of having diverse paths is to allow the tree to handle as many environments (obstacle configurations and goal positions) as possible. The main idea is to use a greedy and randomized-based method to incrementally pick subpaths to add to the tree, starting from an empty tree. We use this method to build trees of up to fifty depth levels. While this tree building method is simple and greedy, we found that we can use it to successfully find solution paths in large environments with many obstacles. We also show the advantages and disadvantages of our large and diverse tree compared to previous methods for building similar kinds of diverse trees.

We demonstrate the efficiency of our technique across a range of examples in an interactive application with multiple autonomous characters navigating in dynamic environments (Figure 1.3 right). Each character re-plans in real-time according to arbitrary user changes to the en-

vironment obstacles or navigation goals. We empirically show that the runtime phase of our technique is more than two orders of magnitude faster than traditional forward search methods; this is under the condition that we are given a set of motion clips as input, and the output is a concatenation of any sequence of these input clips.

Additional value over previous work: We show a *complete* system that demonstrates the concept of precomputation for planning: we show how to precompute diverse search trees; we describe an efficient runtime backward search method for solving planning queries; we use these methods in actual planning scenarios and show runtime results; and we have an interactive system with many characters navigating in complex environments using our approach. **Assumptions/Limitations:** Similar to the behavior planning approach, we assume that we are given a set of blendable and segmented motion clips as inputs. An output sequence is limited to be a concatenation of the input clips. **Insights:** First, we have found that precomputation is certainly a viable approach for motion planning. However, there is a tradeoff between memory, runtime speed, and optimality. These issues should be considered before choosing between precomputation and traditional search methods. Second, we show that precomputing diverse trees with randomized-based approaches is fast and simple, but they work surprisingly well. Our trees can solve more randomly-generated planning queries than previous methods [10, 30] for building diverse trees.

1.3 Modeling Spatial and Temporal Variants in Motion Data

When a person performs the "same" motion more than once, each motion will be performed in a slightly different manner. This is an important part of creating realistic motion that has not been fully explored. For example, typical crowd animation systems [66] utilize a few walking motion clips for every walking cycle and every character of the simulation. This can lead to synthesized motions that look unrealistic due to the *exact repetition* of the original walk cycles. Hence a variation model that can generate even slight differences of the original walk cycles has the potential to greatly improve the naturalness of the output animations. Current games and films [99] also do not typically produce human-like variations in their crowd simulations. In these applications, as soon as even one example of repetition is identified, the whole animation can be immediately deemed un-humanlike. This can make games and films less fun and interesting.

Previous methods consider variation to be an additive noise component, which is not robust for automatically generating animations. While there are previous methods for adding noise to existing motion [8, 75], there is no guarantee that the added noise will match well with the motion. Adding noise is an ad-hoc process that requires manual parameter tuning.

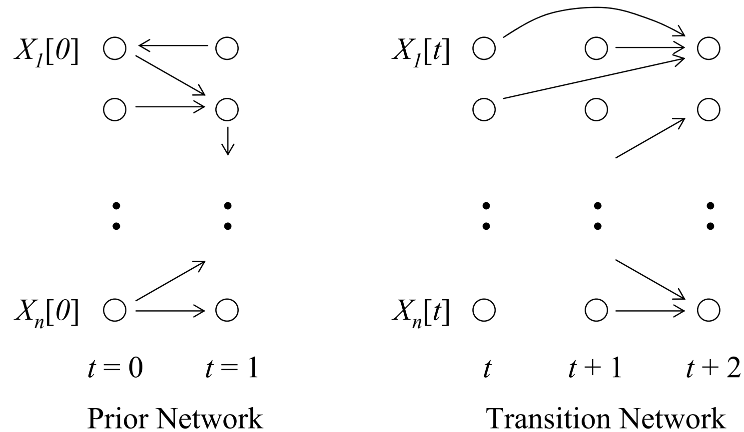


Figure 1.4: A DBN for the variables $X_1, ..., X_n$. Each node X_i represents one DOF in the motion data. We use the prior network to model the first 2 frames of each input motion clip. The transition network then models subsequent frames given the previous 2 frames. We assume a 2nd-order Markov property because it is the simplest model that works well.

We believe that variation should not be just an additive noise component. Instead, we take a data-driven approach to this problem. Given a small number of examples of a particular type of motion (ie. cheering, walk cycle, swimming breast stroke) as input, we learn a model from the input data, and use this model to synthesize spatial and temporal variants of that motion. We claim that the Dynamic Bayesian Network (DBN) [24, 27] model solves this problem well as it provides a formal and robust approach to model the distribution of the data. A Bayesian Network (BN) represents a joint probability distribution of a set of related variables. A DBN extends this distribution for temporal processes: it represents a joint probability distribution of a set of related and time-dependent variables. In the case of motion data (Figure 1.4), the variables are the degrees-of-freedom (DOF) in the data. It is this probability distribution from which we sample to synthesize our new variants. There are three major steps for learning a model and synthesizing new variants. First, we learn the structure of the DBN using the input examples. We use a greedy algorithm based on a variant of the Bayesian Information Criterion score to select a good structure. Second, we use the learned structure and the original data to synthesize new variants. Third and optionally, we can use an inverse kinematics method developed within our DBN framework to satisfy any foot and hand constraints.

The key result of our method is that we can take a few examples of a particular type of motion as input, and produce an unlimited number of spatial and temporal variants as output. The new variants are *statistically and visually similar* to the inputs, but are *not exact copies*. We demonstrate our approach with a variety of full-body human motion data. To evaluate our approach, we perform a user study to show that: (i) our new variants are just as natural as motion

capture data, and (ii) our new variants are less repetitive than "Cycle Animation". In addition, we demonstrate that "just adding noise" to existing motion can create poses and timings that look obviously awkward. We show this with two methods to add noise to motion: (i) a naive/strawman method, and (ii) the Perlin noise function. Finally, it is useful to know what types of inputs work well with our approach. Hence we provide a DBN-based method to select a subset of examples (from a larger set) that would work well with our approach.

Additional value over previous work: The area of synthesizing motion variation is largely unexplored. We provide a data-driven approach to take a few examples of a particular type of motion, and generate spatial and temporal variants of them. **Assumptions/Limitations:** We assume that the input examples come from a particular type of motion (ie. walk cycles, jump forward). Our current approach cannot combine different types. The inputs have to be "similar", but we specifically describe how we can get inputs that work well with our approach. Since we are generating variants of the inputs, the outputs can sometimes be only slightly different visually, althought their numeric values can be quite different. **Insights:** It is possible to automatically generate spatial and temporal variants from a small amount of data. We think of our work as one step towards the problem of motion variation; we believe that the overall problem is still largely unexplored and there is a lot more that can be done.

Chapter 2

Related Work

We first review previous work in the area of crowd animation. We then discuss work that relates to each major section of this thesis: motion planning methods for generating animations of virtual characters, precomputation methods for animations, and techniques for modeling and synthesizing variation in motion data.

2.1 Crowd Animation

There has been much work that focuses on generating crowd animations. We review previous work on steering approaches, particle-based approaches, and AI-based methods.

Steering Approaches. These approaches use a set of simple rules [87] to move each agent. The idea is that a combination of these local rules applied to each agent can lead to emergent behaviors for the group of agents. Since this method requires a nearest-neighbor computation for each agent, the basic approach has a quadratic runtime bottleneck. Reynolds uses spatial hashing to solve this issue, and is therefore able to generate an interactive flock of 280 boids at 60 frames per second [85]. Recently he has extended this method to scale to even larger numbers by incorporating a multi-processor approach [86]. This method can generate up to 15,000 individuals at 60 frames per second.

The basic steering approach pioneered by Reynolds [87] has been extended for many different scenarios. Anderson and his colleagues [3] developed a method to make it easier to control the final motion. Although the emergent behavior is appealing, it is sometimes difficult to control each agent's motion. Their method can be used to allow the agents to meet specific position constraints at specified times, while still retaining their crowd-like features. Lai and his colleagues [51] took a set of simulated motions for a crowd of agents, and then clustered these motions

into different groups to build a graph-based structure of these motions.The idea is that they can then "replay" the original motions in different ways to generate motions of the agents to follow a user-drawn curve, for example. This also allows for a better control of the motions for the group of agents. Pelechano and her colleagues [73, 74] use a combination of psychological and geometrical rules to get emergent behaviors for multiple agents. They are able to generate motions for agents in a line formation and agents pushing each other in a crowded environment.

There exists many other methods to generate motions for a large number of 2D agents, although they differ from the basic steering approach [87]. Blue and his colleagues [7] use a small set of rules to generate the behaviors of pedestrians. They can achieve real-life pedestrian behaviors with their method. Their method uses a coarse discretization of the 2D space: while this allows for a simplified model to study the behaviors that can be achieved, it is unrealistic for motion synthesis for human-like characters. Kamphuis and his colleagues [40] focuses on how to generate motions for individuals to stay together as they move along a path. They can make guarantees of the amount of dispersions of the individuals in the group. The idea is that given a user-specified path or a pre-planned path, they can generate motions for the individual agents to follow the path.

All of the above methods differ from our techniques in that they can work well for generating motions for simple "boid"-like characters in simple environments. Our techniques can generate motions for human-like characters in large and dynamic environments. Steering approaches use local policies or rules that are typically very fast to compute, but they often fail in complicated maze-like environments with local minima. Large environments where indirect paths are needed for the characters to reach a far away position can be troublesome for steering methods. Hence steering methods work well in environments with few or no obstacles. The strength of our global planning method is that it can handle large and complex environments. It is also difficult to adapt these methods appropriately for more complex character skeletons such as human figures. These rule-based methods can generate 2D or 3D paths for "boid"-like characters to follow. If we generate a 2D path for a human-like character to follow, this may cause a character to turn suddenly to avoid something. Such sudden turns can result in jittery motions. Although more sophisticated methods can potentially be used to handle this issue, employing them will increase the runtime of the whole approach significantly, especially if there are a large number of characters. In addition, applying simple rules cannot guarantee collision avoidance between multiple characters. Our planning scheme applied to individual motion clips can handle these issues.

Particle-Based Approaches. These methods model a force-based system affecting many

particles (or agents) instead of considering the motions of each agent separately. These models often assume that the agents are particles of gases or fluids. Realistic crowd behaviors can be generated by modeling the agents this way.

Helbing and his colleagues [33, 34] model pedestrians as point particles. They use a force-based model [33] to "move" each particle. Each particle can be affected by some attractive or repulsive forces based on proximity to other particles and other objects in the environment. They can then compute simulations of pedestrians forming lanes with other pedestrians walking in the same direction. They can also generate two groups of pedestrians moving through a narrow space in opposing directions. They also use a particle-based model of pedestrians [34] to study how they interact in panic situations. They can then use their model to reproduce empirical observations of real-life escape panic scenes. The significance of their work is to study how pedestrians move in different situations. This allows them to propose ways to design structures to alleviate the danger from crowd panic situations. Their work provides an empirical model to study crowds of people. We do not deal with crowd behaviors in the same sense: our characters are considered as individuals and there is no explicit interaction between them.

Treuille and his colleagues [102] generate crowd motions by thinking of crowds of agents as particles in a fluid. They model a potential field in the 2D environment, and use this potential field to move all the characters at the same time. They can generate local collision avoidance between the characters, and be able to achieve some nice effects of crowds such as lane-forming.

In general, the above particle-based approaches model a force-based system to "move" all the particles at the same time. This is in contrast to the agent-based methods for steering approaches. These particles-based approaches work well for generating 2D paths for characters to follow. Similar to steering approaches, they are difficult to extend to human-like characters. However, they can model interactions between particles well. While our planning approaches work well for human-like characters, we do not model any crowd-like interactions between them.

AI-Based Methods. These methods use artificial intelligence (AI) to autonomously generate the motions for each character. Each character may have a certain emotional state; they are given goals and can interact with others just like a real human crowd. These methods can generate more realistic human-like behaviors compared to steering and particle-based approaches.

Funge and his colleagues [25] described an AI paradigm as part of an approach to control the animation of characters give high-level controls. The approach has the same disadvantage as the deliberative AI paradigm: specific behaviors have to be programmed into the system, and it is difficult to extend simple pre-programmed behaviors to truly intelligent characters.

There are commercial systems [1, 2] that use AI techniques to generate motions for large

crowds. *AI-Implant* [1] can be used to simulate crowd and vehicle behaviors for games and training simulations. One of their advertised strengths is the ability to have the characters adapt to changing environments so that game programmers do not have to tediously script every scenario. *Massive* [2] is a computer animation package for crowds. It was originally built for generating the battling scenes in the *Lord of the Rings* films. It can be used to quickly create a large number of individual agents; AI techniques are then used to control each character autonomously.

Shao and Terzopoulos [91] has developed a system of virtual humans behaving automatically in a virtual train station. Their system combines goal-directed behaviors, reactive behaviors, navigation rules, and perception capabilities to generate the motions for each character. However, many of the rules or actions for the characters have to be manually defined, similar to the way a large-scale AI system for crowds are built. Yu and Terzopoulos [112] further incorporates the use of decision networks from the learning community to reason about how the characters should move in an uncertain world.

Our planning method is similar to the above AI-based approaches in that we also have to pre-define the specific behaviors that our characters can perform. Given a small number of short motion clips, the idea is to globally plan for a sequence of them when needed during runtime, so that we do not program or script every scenario as many games do. We show that we can take a small number of motion clips, and apply a variety of planning techniques to generate some nice behaviors for many characters. We do not build an AI-system of the characters: we only focus on the planning and navigation of the characters' motions.

2.2 Motion Planning Methods for Animation

We start by reviewing the simplest approach for planning a character's motion and then progressively discuss more complicated planning techniques. We then discuss work on *motion graphs* and *move trees*.

Planning Approaches. There has been much work that uses a planning approach for synthesizing animations. The simplest approach is to take a 2D top view representation of the environment, and plan for a 2D path that travels from the start position to the goal position while avoiding the obstacles. The character is bounded by a cylinder for the purpose of collision detection. The obstacles are then enlarged by the radius of this cylinder (or circle in the top view), and the character is then represented by a point. The problem then becomes finding a valid path for this point [63, 104].

Kuffner [49] uses this approach to synthesize motions for a character navigating in a maze-

like environment. A proportional derivative controller is then used to allow the character to follow the path. Cycles of walking motions are played back in order to simulate the actual motions of the character. Pettre and his colleagues [77] also use this approach to generate a valid path. However, they use a probabilistic roadmap [41] technique to search for this path. Their path is built to be a composition of Bezier curves. A controller is then used to blend existing data to generate the walking cycles. A warping procedure [13, 109] is applied to the arms for avoiding obstacles, and to allow for more detailed "reaching through a fence" motion.

Esteves and her colleagues [21] also uses this 2D view of the character. Their system, however, is more complex because it allows for multiple characters to perform cooperation tasks. They first plan for paths for their characters and an object being manipulated with a roadmap technique; these paths are then converted into trajectories. A locomotion controller [76] is used to synthesize the motions of the feet, and inverse kinematics (IK) is used to synthesize the motions of the arms. The strength of their work is in the combination of these techniques to generate cooperation tasks for multiple characters.

An important difference between these methods and our *Behavior Planning* method [52] is that we do not first plan a 2D path in the environment and then make the characters follow the path. The actions of the planning algorithm are motion clips that correspond to behaviors like jumping or jogging forward. So we plan for sequences of these behaviors for the character to reach a desired goal position. This has the advantage that the synthesized motions will already be natural by construction. Most crowd and game systems that have their characters follow a 2D path often contain unnatural jittery motions.

Manipulation planning allows these virtual characters to interact with the environment. The key idea is to separate a manipulation task into transit and transfer paths [44]. Transit paths allow the arms to move to and from the object being manipulated. Transfer paths generate the object movements and the corresponding arm motions. The movement in the six DOF space (three for translation and three for rotation) of the object is first planned. An IK algorithm is then used to allow the arms to grasp this object during the transfer path. Yamane and his colleagues [110] also uses this approach to generate animations of manipulation tasks. They use a natural velocity profile to convert paths to motion trajectories. Liu and Badler [62] create reaching motions similarly by first searching for a path for the end-effector (ie. the hand), and then using IK to fit the rest of the body to the positions of the end-effector. They use the depth buffer to perform more efficient collision detection. Our method does not generate any character interactions with objects in the environment. We focus only on navigation methods for the characters to avoid obstacles in large and dynamic environments.

Roadmap-based approaches apply Probabilistic Roadmap (PRM) techniques [41] to anima-

tion. Roadmaps were originally developed in robotics to handle situations that have a large number of DOFs. It is also well suited for problems where multiple queries or solutions are required. For animation, roadmaps are used in a similar way in that we first build a graph of where the characters can move in the environment. However, this graph is often constructed in the space of the environment that the characters move in, as opposed to the configuration space of the objects being moved in robotics. This graph or roadmap is often built in a lower dimensional space based on human knowledge about how the characters can navigate.

Salomon and his colleagues [89] first build a visibility-based PRM [94] that covers the reachable space of a building. They then use this visible-PRM to find a valid path between start and goal locations. The user can also locally steer these characters along the generated path. Choi and his colleagues [16] use a roadmap-based approach to first construct a graph of possible foot placements and transitions in the environment. The graph is searched for a solution path based on the given start and goal foot placements. Motion captured data is then adapted to fit this sequence of foot constraints to synthesize motion. Bayazit and his colleagues [6] build a roadmap of how groups of boids [87] can navigate in the environment. Each boid can use the same roadmap. Each boid can also update the weights of the edges dynamically to generate, for example, a goal-searching behavior for the group. Pettre and his colleagues [78] use a similar approach to build a navigation graph to animate crowds of characters.

As a more general approach to Bayazit et al.'s work [6], we can have roadmaps where the nodes and/or edges are reactive to changes in the environment. The *Elastic Bands* method [Quinlan and Khatib 82] allows paths to deform so that these paths can avoid obstacles dynamically. *Elastic Strips* [Brock and Khatib 12] allows the computation of the paths to be done in the workspace rather than the configuration space, which leads to a faster runtime. *Elastic Roadmaps* [Yang and Brock 111] allows the roadmap itself to adapt. The milestones of the map can move and the map can be continuously updated. In addition, *Reactive Deforming Roadmaps* [Gayle et al. 26] or *Adaptive Elastic Roadmaps* [Sud et al. 96] model the nodes and edges of the roadmap as a physical system. These nodes and edges can be removed and added adaptively as the agents and obstacles move in the environment. In general, roadmap-based approaches are usually used to first build a structure that specifies the reachable space of the environment.

RRT-based approaches apply Rapidly-Exploring Random Tree techniques [54, 55] to animation. RRTs were originally developed in robotics to handle problems with nonholonomic constraints and many DOFs. For animation, they are used to efficiently sample spaces with high DOFs. They work well for generating arm motions, for example, while doing 3D collision avoidance. However, there is no guarantee that the motions are natural.

An RRT algorithm [47] can be used to generate the motions of the manipulated object [110].

In this case, the human motions are synthesized by using IK to fit the character's pose to the object. Kallmann and his colleagues [39] used RRTs to generate reaching motions. They used the idea of RRTs to build and search a roadmap of possible configurations in a pre-defined 22-DOF space. Their technique, however, requires a lot of manual processing to define this space. They also take a straight line in this 22-DOF space to be "smooth"; hence the motions may not be natural. Applying planning methods in such a large space is usually not a good idea as the runtime of these algorithms will increase significantly. In adddition, planning on the individual joints of a human-like skeleton does not guarantee natural motions. Planning on all of the individual joints of a human-like skeleton is the other extreme of the simplest case of planning first in the 2D (top view) environment space. Our behavior planning method avoids both of these extremes to plan for natural sequences of motions efficiently.

There are systems that are explicitly designed to handle human-like figures with many DOFs. Dontcheva and her colleagues [17] presented a layered approach to acting out motions. Each layer specifies the motions of some of the DOFs. For example, the motions of a kangaroo were created with six layers. The first layer specified the large-scale movement of the kangaroo: its motion trajectory. The second layer added details to the motions of the legs. The other layers added more details to the torso, head, arms, and tail. Ching and Badler [15] created motions for a human-like figure by splitting the DOFs into parts, and dealing with each part sequentially given the motions of the previously generated parts. Brock and Kavraki [11] proposed a decomposition-based approach for robot manipulators. They solve for a "tunnel" that is a low dimensional space that includes potential solution paths. They then solve for the final solution by searching in this tunnel. The overall idea is to split up the original problem into simpler subproblems, which results in a reduction of the problem complexity.

Motion Graphs and Move Trees. Our work is closely related to techniques [4, 28, 45, 59, 80] that build graph-like data structures of motions. These approaches facilitate the re-use of large amounts of motion capture data by automating the process of building a graph of motion. Our behavior planning approach [52] is similar in that it also builds graphs of motion data. In our case, we abstract our data into high-level behaviors. This representation offers a number of advantages: it can generate intuitive sequences of motions according to the high-level behaviors and it needs only a small amount of data to generate interesting motions.

Sung and his colleagues [98] use the idea of motion graphs to generate the motions for many characters. The key difference of their work is that they can allow the characters' motions to satisfy position, orientation, and/or timing constraints. This allows for more flexibility rather than just re-playing the existing motion clips as in the motion graph approach.

Similar to crowd systems, interactive applications such as games often use *local policy* methods to generate motions for its characters. Hence these methods have many of the same disadvantages as crowd systems discussed before: they have problems with local minima, and they do not work as well for human-like characters since they can generate only 2D or 3D paths. In our behavior planning method, the representation of motion data as high-level behaviors is reminiscent of *move trees* [68, 69], which are often used in games to represent the motions available for a character. However, our method applies a *global planning* technique. It can output motions that avoid local minima, and generate motions for characters navigating in a large and dynamic environment.

Behavior Planning: Additional value over previous work. The main difference between these previous methods and our *Behavior Planning* method [52] is that we apply a global planning approach in a carefully chosen action space. The actions are high-level behaviors such as jogging forward and jumping: this means that we are planning for sequences of these behaviors. We do not plan in the 2D or top view of the environment first, and we do not plan in the combined joint angle space of all the joints of a human skeleton. Instead our planning space lies somewhere in between these two extremes, and we view our planning approach to be one method among a spectrum of methods. The carefully chosen action space allows our method to be fast and to require only a small memory requirement. Our results show that our method works well for generating navigation motions: (i) we can use a small set of short motion clips to generate a large variety of motions for characters navigating in large environments, and (ii) we can apply a collection of existing planning methods to the basic setup to handle more complicated scenarios such as dynamic obstacles and multiple character motion generation.

2.3 Precomputation Methods for Animation

The idea of precomputation has been used for animation, both for evaluation and efficient generation of motions. We discuss these methods, and identify the differences between them and our Precomputed Search Trees [53] approach. There has been recent work in the robotics community on achieving path diversity among a precomputed set of paths. We then discuss the similarities and differences of these methods compared to ours.

Reitsma and Pollard [83, 84] introduced the idea of embedding a motion graph into a 4-D grid. The 4-D grid then represents the possible ways the character can move in the environment. The embedding works for a specific static environment. This is not a problem for their work because their focus was on the evaluation of the character's moving capabilities given a set of

motions. In contrast, we are concerned with improving the runtime speed of our search technique. Their embedding of motion data and our precomputed trees data structure are similar in that both represent the possible paths that the character can move. Both of them can be used to synthesize motions for a character navigating in the environment. On the other hand, one important difference between their method and ours is that they take the environment into account when they build their data structure. They also build paths for all potential start positions and all goal positions. In our case, we build a tree that represents where the character can go given its current position. This corresponds to having one start position (for the root of the tree) and all the possible goal positions. Most importantly, we do not take the environment into account when we build this tree. This allows us to reuse the tree in different parts of a large environment. We can also deal with dynamic environments as we can map the obstacles to the tree to deal with collision detection when needed. Moreover, we can use the same tree for all the characters. Thus our method is particularly good for efficient pathfinding for many characters in a large and continuously changing environment.

Some of the findings in [83, 84] motivated the design of the motion clips we used. They found that an extremely small dataset was capable of producing good behavior, and that adding duplicate motions provided a relatively small improvement in path quality and coverage. Hence we were able to use a rather small set of motion clips. We also found that we were able to get a large combination of overall motions from a small set of motion clips, given that there is some variety in these clips (a few different types of turning and a few slightly different jog forward clips, for example) and that they are relatively short compared to the size of the environment. Therefore, we do not find scalability to be an issue in our framework: a large amount of data is not needed.

Sukthankar and her colleagues [97] also use the idea of un-rolling a motion graph to cover an environment. They find the paths and costs for the character to reach each point in a 2D grid. Their embedding also takes obstacles into account. This is again different from our work. Our focus is on designing a system that allows for efficient runtime searches, while being able to handle dynamic obstacles and multiple characters.

Lee and Lee [58] have also explored the idea of precomputation. They preprocess sets of motion data to compute a *control policy* for a character's actions. This policy was then used at runtime to efficiently animate boxing characters. There are also existing approaches that use reinforcement learning to compute control policies for generating a character's motions. Ikemoto and her colleagues [37] generate motion for autonomous agents with a controller derived from reinforcement learning methods. Treuille and his colleagues [103] compute near-optimal controllers to interactively control the motions of human-like characters. McCann and Pollard

[65] also compute controllers based on reinforcement learning. They choose the character's next fragment of motion based on the current player input and the previous motion fragment. Our approach is different from these control policy based methods in that we precompute a search tree of all possible future actions given some existing motions. We then build gridmaps over this tree so that we can efficiently index to the relevant portions of the tree during runtime.

Go and his colleagues [29] used the idea of precomputation to create animations of vehicles. Their method allows automatic steering of 3D points by selecting the best trajectory among a set of precomputed ones. Our work is similar in that we precompute potential character trajectories to speedup the runtime search. Their method, however, works well for producing 3D paths for boids, birds, or spacecrafts. The kinds of paths that they produce can also be created from steering approaches [87] for crowds. Since these boid-like characters can move around with great flexibility in the 3D space, they cannot exploit as much the strengths of precomputation. For the case of virtual humans, there is much less flexibility in the ways that they can move. It is common for game and crowd systems to use specific motion clips to animate their characters. Hence our method can exploit the strengths of precomputing these potential paths for efficient runtime search later.

There are many approaches that first computes some kind of map of the environment before the actual motion is generated. Sung and his colleagues [98] use a two-level planning technique to animate their characters: a probabilistic roadmap approach [41] first generates approximate motion paths; these paths are then refined to precisely satisfy certain constraints. Sud and his colleagues [95] first compute a multi-agent navigation graph based on the configurations of the agents and obstacles. The first and second order voronoi diagrams of these configurations are used to compute the navigation graph. This graph can then be used to create a maximum clearance path for each character. van den Berg and his colleagues [106] first compute a global roadmap of the environment to connect the reachable parts of the space. They then use a *reciprocal velocity obstacles* method [105] to generate smooth navigation paths for each character. The work in [106] has the same high-level goals of our precomputed trees work. In their work, they can create the motions of thousands of 2D agents. We generate the motions of up to 150 human-like characters. Their work is well suited for generating paths for 2D agents while our method can be used to generate natural motions for human-like characters.

We now discuss previous work related to the issue of path diversity. The motivation for studying this issue is that while our original system [53] demonstrates the concept of precomputation, it does not build scalable and diverse trees that can be used in more general planning scenarios. This is also true for many previous methods [10, 19, 30, 43], where the number and length of paths built are too small to use for general planning problems.

Our original method [53] builds trees that have a limited depth level. We showed that the concept of precomputation can lead to a faster runtime, but we showed these results either for problems with very small environments or problems requiring a two-level hierarchical approach. Using a two-level approach made it difficult to compare the advantages and disadvantages of the precomputation method against A*-search methods. We were unable to make this comparison fairly in the original system because it is difficult to build large trees effectively. While there are cases when it is beneficial to combine two-level approaches with precomputed search trees, building precomputed trees of at least a reasonable size and comparing them to traditional forward search methods are still important issues.

Green and Kelly [30] and Branicky et al. [10] describe methods to take an existing set of paths in a tree and select a smaller set from it that is as diverse as possible. These methods require at least quadratic time with respect to the number of paths; hence they can only be used for small path sets. Furthermore, they both require the existence of a set of paths from which to select from. Generating the exhaustive set of paths to select from only works for trees with small depth levels, and this is in fact the approach that we take in our experiments. For larger trees, generating an exhaustive set requires exponential time with respect to the depth level, and it is not clear how we can generate a subset of paths from which to begin selecting from. Indeed, our randomized-based tree precomputation method solves this problem: how to generate such a set of paths for trees of large depths while keeping the paths diversified enough that the trees can be used to solve as many planning queries as possible. We show empirical results comparing: our original method [53], our randomized-based method for computing large and diverse trees, and three other methods from previous work [10, 30].

Erickson and LaValle [19] also explore the idea of path diversity. They introduce a survivability criteria that can decrease the likelihood that numerous paths will be obstructed by the same obstacle. Given a larger collection of paths to begin with, they use this criteria to select a smaller subset of paths. Knepper and Mason [43] explore the similar idea of taking a set of paths and selecting one of them for execution. Their focus is on comparing between the static and dynamic planning cases for this idea of selecting one path among many to execute. Both of these recent works [19, 43] consider path sets that are very small. This is an important limitation because their small path sets can work in environments that are small and relatively uncluttered. Given an environment with many obstacles, their path sets will not return a solution even though one exists. While their work focuses on the analysis of the path sets, it is difficult to use them in real planning scenarios of large environments cluttered with many obstacles.

Our method for building large and diverse trees is related to sampling-based planning approaches. In our algorithm, we choose nodes from which to expand from in the same way as

Rapidly-Exploring Random Trees (RRTs) [57]. We choose nodes this way for the same reason as RRTs do: so that the selected nodes will be evenly spaced and not biased towards a particular region. Our algorithm differs from RRTs as we use a metric to locally pick paths that are as evenly spread out as possible; this process does not take the obstacles into account as the tree is being precomputed. Probabilistic Roadmaps (PRMs) [42] are effective for planning in high-dimensional spaces. They first build a roadmap for a given environment and then use it to find solution paths. Our method also has a preprocessing phase, but our precomputed tree can then be used during runtime for any obstacles and any start/goal queries. An extended version of PRMs [60] first builds a tree without taking obstacles into account and later map them back into the environment. The difference in our case is that since we have a set of actions as input, we first build the tree in the action space. This is more general because each path of the tree can later fit anywhere in the environment. Finally, the key difference between our method and RRTs and PRMs is the overall precomputation concept to first *precompute a tree* and then use a *runtime backward search* to find a solution.

Precomputed Search Trees: Additional value over previous work. **Precomputation Concept:** Our approach differs from the standard planning approach and the previous precomputation methods. The standard planning approach takes one starting location and one goal location as input, and generates a solution from the start to the goal. It builds one search tree for each query. The main idea of our approach is to precompute a tree of potential paths. This tree can be used for any configurations of start and goal locations. With respect to the previous precomputation methods, it is common to first compute some kind of map of the environment before motions are generated. The main disadvantage of this is that if the environment changes, the map has to be recomputed every time. The key to our approach is that we first compute a large and diverse tree of motion paths given a set of human motion data. We empirically show that we can re-use the same tree for a large number of obstacle configurations and goal locations. We can also re-use the same tree for different parts of the environment even if it is large. As a result, we found that by doing this precomputation, we can generate the motions for a large number of human-like characters interactively. **Path Diversity:** While there has been much recent work on studying path diversity, the path sets that these works analyze are very small and are difficult for use in actual planning scenarios. We describe a simple but effective randomized-based method to precompute large and diverse trees. We provide empirical results to compare our trees with trees built with previous methods. We also provide empirical results to compare the precomputation concept with traditional forward search methods with actual planning queries. We can make this comparison fairly by using the large and diverse trees that we built. Such a comparison is use-

ful for researchers as the tradeoffs between precomputation and forward search methods must be understood before deciding to use one of these methods. **Complete System:** Finally, our work shows a complete system that demonstrates the concept of precomputation for planning: we show how to precompute large and diverse search trees; we describe an efficient runtime backward search method for solving planning queries; we use these methods in actual planning scenarios and show runtime results; and we have an interactive system with many characters navigating in complex environments using our approach.

2.4 Modeling and Synthesizing Variation

Variability in human motion has been studied for generating diversity in motions. One major previous approach for generating variation in motions is to add noise. Perlin [75] adds noise functions to procedural motions to create more realistic animations of running, standing, and dancing. Bodenheimer and his colleagues [8] adds noise to cyclic running motions. They constructed a noise function based on biomechanical literature to vary the joint angles in the upper body. The noise is added only to the upper body, and is synchronized with the arm swings in the running cycle. Adding noise in such a supervised way requires human knowledge and parameter tuning. Our approach is fundamentally different because the variations that we generate automatically come from the data and is not a separate additive component. Instead we learn a joint probability distribution of the motions from the data, and then use this distribution to generate new motions.

Pullen and Bregler's work [79] to generate motions that are slightly different but similar to the original data is most closely related to our work. They model the correlations between the DOFs in the data with a distribution, and synthesize new motions by sampling from this distribution and smoothing the motions. However, they have to define certain correlations manually. For example, they specify manually that the hip angle affects the knee, and the knee angle affects the ankle. The structure learning in our DBN framework learns these relationships directly from data. Their method only predicts the value of each DOF given the value of one other user-specified DOF. In our case, we can find the probability of each DOF given the values of a subset of DOFs across previous time steps. These subsets of DOFs are learned automatically from data; this process is a key component to the idea behind DBNs. Our method can be thought of as a generalization to Pullen and Bregler's approach [79]. In addition, they used their method to animate a 2-dimensional 5-DOF wallaby figure, and a more complex 3D character in later work [81]. Since they have to manually specify the probability relationships between the DOFs of the character, their approach does not work well for human-like characters with many DOFs. On the

other hand, we demonstrate results of different kinds of motions for a full human figure.

Our method is similar to the "Texturing" method by Pullen and her colleagues [80]. They also used the idea that the joints of a human figure are correlated to predict the values for some DOFs given the values of other DOFs. Our DBN framework also depends on this observation to predict new DOF values. Their objectives are different from ours: (i) their "synthesis" procedure allows them to synthesize the values of some DOFs that did not exist before, and (ii) their "texturing" procedure allows them to add details to some DOFs. Their methods allow animators to more easily combine traditional keyframing methods with motion capture technologies. Our focus, on the other hand, is to generate variations of motions from input data. More specifically, their algorithm has to break the input motions into segments and align them. Our method does not require synchronization or alignment of similar motion clips. In addition, their method takes existing motion curves from different joints, and combine and re-order them to generate new sequences. They do not explicitly change the values of the motion curves. For our case, it is important that we can generate poses that are completely different (in terms of values) than those in the inputs. The exact poses should be different but the overall motion should be a variation of the inputs.

Our approach is similar to previous work in texture synthesis [18, 107]. An important idea for texture synthesis is to use a non-parametric approach and directly sample queries from data. Our method is similar because we also use a non-parametric regression approach to predict values. We first tried to use a parametric approach, but that did not work well. This might be due to the small amount of input data that we have. We also have temporal relationships in addition to spatial ones in our DBN framework. Other texture synthesis methods [5] model the input texture with statistical methods. New and larger textures are generated to be statistically similar to the inputs. We also learn a model of the input motions to synthesize new motions. In addition, Moradoff and his colleagues [70] re-order short segments of motion clips to generate new sequences of motions. Their method is similar to motion graph techniques for re-ordering existing motion sequences to generate new sequences. However, the individual frames of motion are not new. In our case, we generate variants where individual frames are different from those of the inputs.

Sidenbladh and her colleagues developed models of cyclic human motion in their vision system for tracking human figures [72, 92]. Their model can potentially be used to generate variations of the input cycles, but it was designed for (and therefore more appropriate for) a tracking system. Their model allows them to compute the probability of a pose. This gives the prior probability that works well for constraining the search in a bayesian framework for tracking. Their model is thus not designed to generate new poses. In contrast, our generative model is good

for synthesizing new motions. We learn a "distribution" of the motions, and sample from this distribution to get new motions. Furthermore, their algorithm needs to detect, segment, and align the cycles of motion. This process loses the temporal information in the data. Our model takes the temporal relationships into account, without having to align the input motion cycles.

Similar to the non-parametric-based texture synthesis methods [18, 107], Sidenbladh's system also used a non-parametric sampling-based approach [93]. Our model also uses this approach since we use previous frames of motions to predict future ones. More specifically, we use the correlations between joint angles and correlations in motions over time to predict future DOF values. The important idea here is that the prediction of each future DOF value is based on some subset of previous values, and this has a conditional probability distribution. Each DOF value is independent of other values given the particular subset used to predict that DOF. These conditional distributions allow us to build a joint distribution over the motion trajectories for all DOFs, and this joint distribution allows us to sample new variations of motions. Sidenbladh's framework is designed for tracking, and their synthesis technique simply adds gaussian noise to introduce variety into new motions. This process has the same disadvantages of methods (discussed earlier) that add noise to existing motions.

Machine learning has been used for learning models of human motion data. Li and his colleagues [61] generated new motions that are statistically similar to their input data. However, they used 20 minutes of dancing motion as training data. If a large amount of data is available, it is possible to just randomly replay or re-organize certain motion clips without being able to detect repetition in the motion. One of the strengths of our work is that our approach can handle a small amount of original data. Grochow and his colleagues [31] developed a method to learn a probabilistic model of input poses. They use their model to generate new poses in an optimization framework for doing inverse kinematics (IK): the idea is to find a pose that satisfies certain IK constraints and maximizes the probability of the pose at the same time. Our IK framework was inspired by their work. Our idea is to find solutions that satisfy three constraints: the solution should be close to the value predicted by the DBN, the smoothness of adjacent poses, and the IK constraints for the foot/hand. Yu and Terzopoulos [112] use a Bayesian Network (BN) to model the probabilistic nature of motions. In their model, each of the nodes or random variables is a high-level motion such as walking. The model then specifies how likely a character will transition to other motions from walking. The whole motion includes a small number of different motions and the possible transition between them. However, these transition parameters were not learned from data, but were specified manually. Their contribution is to argue that these kinds of models can be used to model the probabilistic and uncertain nature of motions. We use a Dynamic Bayesian Network model in our work. The "Dynamic" refers to the temporal

relationships between the random variables, which are not modeled explicity by BNs. We learn such a model from data, and use it to synthesize completely new motions. It is also interesting to note that Yu and Terzopoulos' model is similar to our Behavior Planning framework. Their key idea is to use a Bayesian Network to model the motions, whereas our approach uses a planning framework to synthesize long sequences of motions automatically for characters navigating in a complex environment. Finally, Hertzmann [35] mentioned the idea of using DBNs to model motion data. Our work uses the idea of DBNs to solve the problem of modeling and synthesizing variations in motion data.

We model our motion data with a Dynamic Bayesian Network (DBN) [24, 27]. A DBN represents the state of a set of random variables and how they change over time. In our case, each random variable is a DOF of the human skeleton. The DBN represents a multivariate probability distribution of how these DOFs change over time. The variations that we generate are sampled from this distribution. BNs and DBNs have been used for many applications, including medical diagnosis, automated help features in software, and automated traffic prediction. As a specific example, automated traffic prediction [22, 23] collects data on each car and its nearby cars, and makes predictions on their traffic patterns. The data that they collect for each car include: its own velocity, the position and velocity of nearby cars, and whether or not there is a car to its left and right. They can then learn a DBN with random variables corresponding to these high-level descriptions (such as the position of a nearby car). A potential prediction that can be made from the DBN is: if there is a car to my left and its relative velocity is much faster than me, then it is more likely that there will not be a car to my left at the next time step. Such predictions can be potentially used for automated driving and safety warning systems. These models seem to be ideal for modeling and synthesizing variations in motion data, because they model the joint probability distribution of time-series data. The advantages of using DBNs for this problem include: the variations will not be just adding noise as in previous methods; the new output motions will be statistically similar to the inputs; we only need a small amount of data; and no timewarping is needed for the input motion clips.

Bayesian Networks (BNs) have been used in the animation community for solving different problems, in contrast to the variation problem that we explore. BNs have been used to model the motion of virtual humans [112], where the variables in their network correspond to high-level behaviors. Kwon and his colleagues [50] use DBNs for the problem of animating the interactions between two human-like characters. Ikemoto and her colleagues [38] use similar types of probabilistic methods for the problem of motion editing.

There is much interest in the problem of adding variety to virtual crowds. Maim and his colleagues [64] take a fixed number of template character meshes, and vary them by changing

their color and adding different accessories to them. On the other hand, our work takes a fixed number of template *motions* and synthesize new variant motions from them. McDonnell and her colleagues [67] perform user experiments to study the perception of clones in virtual crowds. They assume that the motion of crowds of characters must be cloned. In contrast, our approach creates motion with *no exact clones*, even though the new variants are visually similar. The focus of their paper is to study the perception of appearance and motion clones, whereas our approach tries to model input data and synthesize new motion variants.

There has been work on learning the style of motions from training data [9] and transferring the style between motions [36]. Style and variation differ in the following way: a happy walk and a sad walk are different styles of walking, while two happy walks are different variations of a motion. Interpolation methods [88, 108] have been developed to generate a spectrum of new motions that are interpolated from the original data. Interpolation and variation are also different approaches: we interpolate a five-foot jump and a ten-foot jump to get an eight-foot jump, while we take two five-foot jumps to generate variations of that jump.

There is work in the motor control community that tries to explain variability that has been observed in many tasks. For example, Todorov and Jordan [100, 101] explains that one optimal strategy of motor control is to allow for variability in redundant dimensions. They describe a "minimal intervention" principle where deviations are only corrected if it interferes with task goals. From their perspective, variability is also not viewed as noise or error, but is a useful component of achieving certain tasks. This supports our approach of assuming that variation is *not* just a separate additive noise component (as previous work do). We do not provide a new perspective to explain variability, but instead we use a data-driven approach to actually model variation from example data and synthesize new variants from input data.

Our Variation Approach: Additional value over previous work. We feel that the area of modeling and synthesizing motion variation is largely unexplored. The major previous method for generating variations in human motion is to add noise to existing motions. However, biomechanical research [32, 90] has shown that variation should not be viewed as just "error" or noise. Variation is rather a functional component of motion. We therefore provide a data-driven approach to take a few examples of a particular type of motion, model these inputs with a DBN, and generate spatial and temporal variants of the original inputs. The reason for using a DBN is that, in the machine learning literature, DBNs have been developed to model the joint probability distributions of time-series data. Hence this model works well for building a distribution of existing motion data, in order to model and synthesize variants of the original data.

Chapter 3

Behavior Planning

While it is relatively easy to create virtual models of elements in a static scene, it is challenging to populate these scenes with autonomous characters. Creating 3D models of objects or even a 3D model of an entire city has become increasingly easy. One important challenge is to generate realistic virtual humans: we want these virtual cities to be inhabited by lifelike and purposeful characters. Generating these autonomous characters are important for applications such as games and crowds. Currently, these systems often contain many simple scripted behaviors or rules to govern how the characters behave. Our *Behavior Planning* approach applies motion planning techniques to automatically generate the motions for a large number of characters.

We developed our method with these properties in mind:

- We provide a simple goal-driven interface to direct where the characters should go. This interface allows the user to have a high-level control of the autonomous characters.
- We develop a single planning scheme that uses a combination of existing motion planning techniques. Our scheme can handle multiple characters and dynamic obstacle avoidance.
- Our behavior graph data structure leads to our *Precomputed Search Trees* method (next chapter), which is much more efficient than our basic planning approach.

The key difference of our planning approach compared to previous planning methods is the space in which we perform the planning in. Figure 3.1 shows the possible range of planning configuration spaces for generating the motions of a human-like character. On the left of the figure, we have the simplest type of planning: we consider only a 2D gridmap of the environment. In this case, we have a bird's eye view representation of the environment. The character is bounded by a cylinder for collision detection purposes, and a 2D solution path is generated. The character's motion can then be added on with, for example, a proportional derivative controller

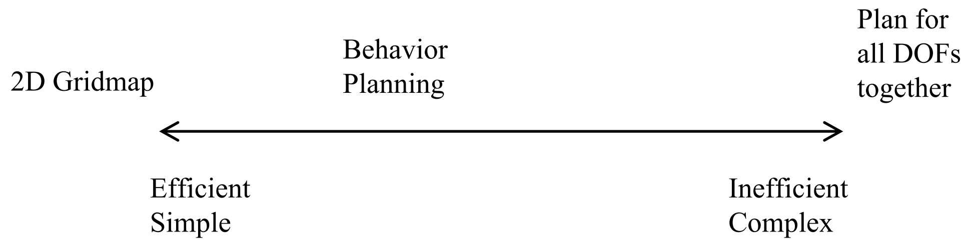


Figure 3.1: The range of planning configuration spaces for generating the motions of a human-like character. Our *Behavior Planning* approach differs from previous methods in that it lies in between the two extremes in this range.

[49]. This 2D gridmap planning is the most efficient: we cannot further reduce the dimensions of the search space. However, the solution only gives us a simple 2D path and no human motion is actually generated by the planning algorithm. On the right of the figure, we can plan for all the degrees of freedom (DOFs) of the human-like character at the same time. The solution would give us the motions for all the DOFs. However, this is inefficient and intractable due to the large number of dimensions (typically 50 to 60) in human motion. It is also very difficult to guarantee that the motions be humanlike. Hence no previous work that we know of plans directly in this space. This is an extreme possibility that we describe here for the sake of thinking about the possible approaches that we can take.

In this chapter, we present our *Behavior Planning* approach [52]. This approach lies in between the two extremes in the range of planning spaces. We segment motion data into short motion clips of high-level behaviors such as jogging forward or jumping. Our method then applies planning techniques at the level of motion clips; the *actions* of the planning method are these whole clips. Thus, we abstract the motion clips in a way that the motion synthesis problem becomes a planning problem. We generate sequences of these motion clips as output.

Planning at the level of these motion clips is important to the success of the algorithm. The advantages include:

- **Ideal for Navigation Motion.** Our approach works well for generating motion for many characters navigating in complex and dynamic environments. Because of the motion clip abstraction, a surprisingly small data set is enough for generating a variety of output motion. We can start with a basic setup for motion generation with one character in static environments. We can then apply a collection of planning techniques to extend this basic setup. For example, we add time as another dimension in the planning space to handle dynamic obstacles, and we use prioritized planning to handle multiple characters.

- **Efficiency.** Our data structure is structured and compact. This allows for a small branching

factor in our search, and hence a fast algorithm. Since we have already segemented our motion data to the level of meaningful motion clips, the number of nodes in our graphs (25 in our largest example) is relatively small compared to motion graph approaches [4, 45, 59]. Each node of our behavior graphs contain entire pieces of motion clips. In contrast, motion graphs typically have on the order of thousands of nodes corresponding to individual poses of motion (and potentially tens of thousands of edges). Gleicher and his colleagues [28] described how unstructured motion graphs can be inappropriate for interactive systems that require fast response times.

- **Memory Usage.** Our method requires a small amount of data in order to generate interesting motions, making it particularly appealing to resource-limited game systems. As an example, our synthesized horse motions are generated from only 194 frames of data.

- **Intuitive Structure.** Because of the structured form of our behavior graph, the solutions that the planner return can be understood intuitively. The high-level structure of behaviors makes it easier for a non-programmer or artist to understand and work with our system. For example, a virtual character that wants to retrieve a book from inside a desk in another room needs to do the following: exit the room it is in, get to the other room, enter it, walk over to the desk, open the drawer, and pick up the book. It is relatively difficult for previous techniques to generate motion for such a long and complex sequence of behaviors. Because we have already partition the motions into distinct high-level behaviors, our planner can generate a sequence of behaviors corresponding to the high-level descriptions above.

- **Generality.** We can apply our algorithm to different characters and environments without having to design new behavior graphs. For example, we generated animations for a skateboarder and a horse using essentially the same graph.

- **Optimality.** Again because of the structured nature of our graph, our method can compute optimal sequences of behaviors. The optimality is respect to the cost of the behaviors. The cost of a particular behavior is the distance that the character travels multiplied by a user weight, which by default is one.

3.1 Problem Statement and Overview

The algorithm takes as input a graph of behaviors, information about the environment, and starting and goal locations for each character. Figure 3.2 shows an example. The problem is to find a sequence of behaviors or motions that allows each character to move from the start to the goal.

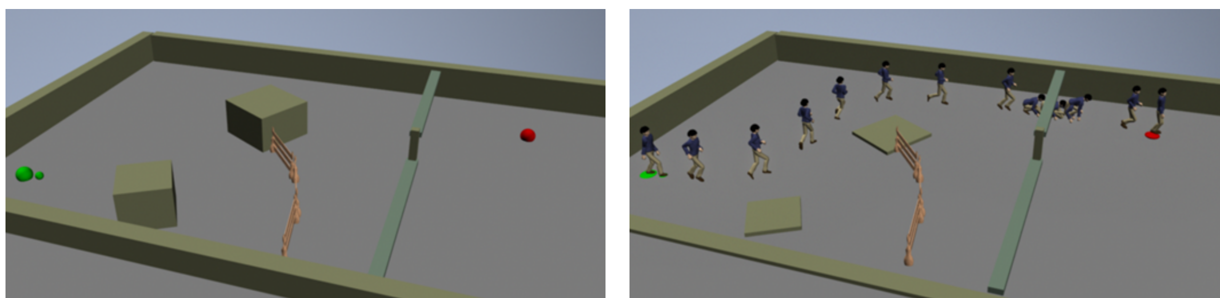


Figure 3.2: *Left:* The problem inputs include a description of the environment, a starting position (larger green dot) and orientation (smaller green dot points toward the direction), and a goal position (red dot). *Right:* The output is a motion sequence.

The sections in this chapter discuss each of these parts in more detail:

Behavior Graph. Motion clips are abstracted as high-level behaviors and associated with a behavior graph that defines the movement capabilities of the character. We describe how to construct this graph, explain the costs associated with the motion clips, and provide details about the data that we used.

Environment Abstraction. We describe the environment representation, which is used mostly for collision avoidance. Some interesting cases include: obstacles that the character must jump over or duck under, and dynamic obstacles.

Behavior Planner. We describe an A*-search planner [56] that performs a global search of the nodes in the graph and computes a sequence of behaviors for each character to reach their corresponding goal position.

Motion Generation and Blending. We convert the sequence of behaviors into motions by concatenating and blending the motion clips.

Results. We show results of synthesized animations with multiple characters planning simultaneously in both static and dynamic environments.

3.2 Behavior Graph

The behavior graph defines the movement capabilities of the character. Each *node* or *behavior* of the graph consists of a collection of motion clips that represent a high-level behavior, and each directed *edge* represents a possible transition between two behaviors. Figure 3.3 left shows a

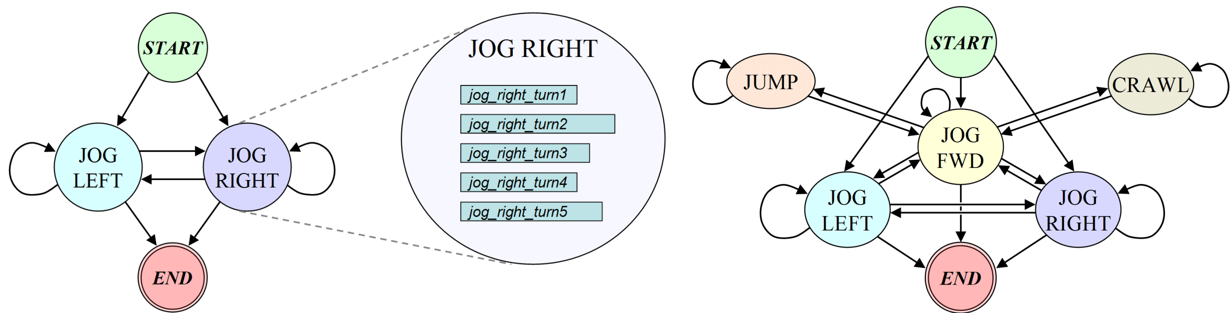


Figure 3.3: *Left:* A simple graph of behaviors. Each node or behavior contains a set of example motion clips for that behavior. Each edge indicates allowable transitions between behaviors. *Right:* An example graph used for a human character that includes special jumping and crawling behaviors.

simple example. The *start* node allows the character to transition from standing still to jogging, and the *end* node allows the character to transition from jogging to standing still.

Most of the motion clips are segmented so that the beginning and end poses are similar. Transitions are possible if the end of one clip is similar to the beginning of another. Hence most clips can transition to each other, allowing for a larger variety in the number of possible output motions. The edges of the graph encode these transition possibilities.

In practice, we have found that it is a good idea to include some motion clips that are relatively short compared to the length of the expected solutions. This makes it easier for the planner to globally arrange the clips in a way that avoids the obstacles even in cluttered environments.

There can be multiple motion clips within a behavior in the graph. Having multiple clips that differ slightly in the style or details of the motion adds to the variety of the synthesized motions, especially if there are many characters utilizing the same graph. However, clips of motions in the same node should be fairly similar at the macro scale, differing only in the subtle details. For example, if a “jog left” clip runs a significantly longer distance than another “jog left” clip, they should be placed in different nodes and assigned different costs.

The cost of each clip is computed by the distance that the root position travels multiplied by a user weight, which by default is one. The distance that the root position travels is the length of the corresponding 2D path if we project the root position of the character to the ground plane. Each node in the graph has only one cost. For multiple clips in the same node, we take the average of the cost of each clip. The user weight allows an animator to control preferences for certain types of behaviors.

Figure 3.3 right shows an example of the graph used for the human-like character. The most complicated graph that we used has a similar structure, except for: (1) additional jogging and turning states; and (2) more specialized behaviors such as jumping. We used seven types of

jogging behaviors: one moving forward, three types of turning left, and three of turning right. We also have a few jogging behaviors that are relatively shorter in length. In addition to "jump" and "crawl", there are also states for: duck under an overhanging obstacle, different types of jumps, and stop-and-wait. The "stop-and-wait" behavior allows the character to stop and stand still for a short while during a jog. For our experiments, the raw motion capture data was downsampled to 30 Hz. Our most complicated graph has 1648 frames of data, 25 states, and 1 to 4 motion clips per state. Each of the seven types of jogging mentioned above has about 25 frames of data per motion clip. Each of the specialized behaviors has about 40 to 130 frames, and each of the shorter jogging states has about 15 frames. In addition, there are 8 frames of data before and after each motion clip that are used for blending.

We also have motion data for a skateboarder and a horse. Their graphs are similar to the one in Figure 3.3 right. For the skateboarder, there are five gliding behaviors: one moving forward, two types of left turns, and two types of right turns. In addition, there are states for jumping, ducking, and stopping-and-waiting. The graph has 835 frames of data, 11 states, and 1 motion clip per state. Each motion clip has about 60 frames of data, and an additional 16 frames used for blending. For the horse, we only had access to one keyframed forward gallop motion. We defined a total of five galloping behaviors: one moving forward, two types of turning left, and two turning right. All turning motions were keyframed from the forward motion. The graph has 194 frames of data, 5 states, and 1 motion clip per state. Each of the clips consists of 20 frames of data, and an additional 12 frames used for blending.

3.3 Environment Abstraction

For *static environments*, we represent the environment e as a 2D heightfield gridmap. This map encodes the obstacles that the character should avoid, the free space where the character can navigate, and information about special obstacles such as an archway that the character can crawl under. This information can be computed automatically given the arrangement of obstacles in a scene. The height value is used so that we can represent terrains with small slopes or hills.

The virtual character is bounded by a cylinder with radius r. The character's root position is the center of this cylinder. The character is not allowed to go anywhere within a distance r of an obstacle. As is standard in robot path planning, we enlarge the size of the obstacles by r so that the character can then be represented as a point in the gridmap [63].

In order to handle special obstacles, we compute a set of *near regions* and *within regions*. A *near region* is where the character is near the obstacle and some special motions such as jumping can be performed, and a *within region* is where the character can be in the process of executing

the special motions. We assume our environments are bounded by obstacles that prevent the character from navigating into infinite space.

Each of the special motions such as crawling need to be pre-annotated with the type of corresponding special obstacle. In addition, the motion clips that are more complex can be optionally pre-annotated with the time that the special motion is actually executed. For example, a long jumping motion clip where the character might take a few steps before the jump can be annotated with the time where the jump actually takes place. If there is no such annotation, we simply assume that the jump occurs in the middle of the motion clip.

Our algorithm handles *dynamic environments*, given that we know a priori how each obstacle moves. Given the motion trajectories of all the moving objects, we define a function $E(t)$ that given a time t, returns the environment e at that time. For static environments, this function is constant.

3.4 Behavior Planner

Given the behavior graph and environment as input, we search for a sequence of behaviors to allow each character to navigate towards its goal position. The search algorithm uses two interrelated data structures: (1) a tree of nodes that is continually expanded during the search; and (2) a priority queue of nodes ordered by cost, which represent potential nodes to be expanded during the next search iteration. Each node in the tree stores the motion clip or action a chosen, and the position, orientation, time, and cost. This means that if we choose the path from the root node of the tree to some node n, the position stored in n corresponds to the character's global position if it follows the sequence of actions stored along the path. The purpose of the queue is to select which nodes to expand next by keeping track of the cost of the path up to that node and expected cost to reach the goal. The priority queue can be implemented efficiently using a heap data structure.

In order to get the position, orientation, time, and cost at each node during the search, we first compute automatically the following information for each action a: (1) the relative change in the character's root position and orientation, (2) the change in time (represented by the change in number of frames), and (3) the change in cost.

The behavior planner uses an A*-search approach; psuedocode is shown in Algorithm 1. The search algorithm is optimal with respect to the cost of the nodes. The total cost at each node is the sum of the cost of the path up to that node and the expected cost to reach the goal (DistToGoal). The planner initializes the root of the tree with the state s_{init}, which represents the starting configuration of the character at time $t = 0$. The planner then iteratively expands the

Algorithm 1: BEHAVIOR PLANNER

```
Tree.Initialize(s_init);
Queue.Insert(s_init,DistToGoal(s_init,s_goal));
while !Queue.Empty() do
    s_best ← Queue.RemoveMin();
    if GoalReached(s_best,s_goal) then
        return s_best;
    end
    e ← E(s_best.time);
    A ← F(s_best, e);
    foreach a ∈ A do
        s_next ← T(s_best, a);
        if G(s_next, s_best, e) then
            Tree.Expand(s_next,s_best);
            Queue.Insert(s_next,DistToGoal(s_next,s_goal));
        end
    end
end
return no possible path found;
```

lowest cost node s_{best} in the queue until either a solution is found, or until the queue is empty, in which case there is no possible solution. If s_{best} reaches s_{goal} (within some small tolerance ϵ), then the solution path from the root node to s_{best} is returned. Otherwise, the successor states of s_{best} are considered for expansion in the tree.

The function F returns the set of actions A that the character is allowed to take from s_{best}. This set is determined by the transitions in the graph. Some transitions may only be valid when the character's position is in the *near regions* of the special obstacles. Moreover, F random selects a motion clip within each chosen node, if there are multiple clips in a node. The function T takes the input state s_{in} and an action a as parameters and returns the output state s_{out} resulting from the execution of that action.The function f represents the translation and rotation that may take place for each clip of motion.

The function G determines if we should expand s_{next} as a child node of s_{best} in the tree. First, collision checking is performed on the position of s_{next}. This also checks the intermediate positions of the character between s_{best} and s_{next}. The discretization of the positions between these two states should be set appropriately according to the speed and duration of the action. The amount of discretization is a tradeoff between the search speed and the accuracy of the collision checking. For the special actions such as jumping, we also check to see if the character is inside the *within regions* of any corresponding obstacles during the execution of the action. In the case of a jumping motion, for example, since we have annotated when the jump occurs, we

can add this time to the accumulated time at that point and use the total time to index the function E.

To handle dynamic obstacles in the environment, our planning algorithm considers time as another DOF [56]. As we build our search tree of nodes, we keep track of the total time it takes to arrive at each node and perform collision detection with respect to the obstacles at that time. This assumes that the trajectories of the obstacles are known in advance. In the psuedocode, we see that the time at s_{best} ($s_{best}.time$) is used to find the configuration of the environment e at that time.

As a final step for function G, we utilize a state-indexed table to keep track of locations in the environment that have previously been visited. If the global position and orientation of a potential node s_{next} has been visited before, the function G will return false, thereby keeping it from being expanded. This prevents the search from infinitely cycling between different previously explored positions. Hence the algorithm will terminate in finite time if no solution exists.

3.5 Motion Generation and Blending

After the search algorithm returns a sequence of behaviors, we convert that sequence into actual motion for the character. To smooth out the discontinuities where the transitions between behaviors occur, we linearly interpolate the root positions and use a smooth-in, smooth-out slerp interpolation function for the joint rotations.

3.6 Results

We synthesized motions for multiple characters by doing prioritized planning. We plan the motion for the first character as usual; each additional character's motion is then synthesized by assuming that all previous characters are moving obstacles. Prioritized planning does not guarantee a *globally optimal* solution for a given group of characters, as solving this multi-agent planning problem is known to be PSPACE-hard [56]. Although it is neither fully general nor optimal, we have found that prioritized planning is efficient and performed very well in our examples.

We present some experimental results that demonstrate the effectiveness of our approach. Figure 3.4 shows three human characters navigating in an environment with a cylinder-shaped tree obstacle that gradually falls down. The first character jogs past this obstacle before it falls, while the two that follow jump over it after it has fallen. Our planner takes less than one second to synthesize about 10 seconds of animation for each character. In general, the amount of search time is significantly less than the amount of motion that the planner generates.

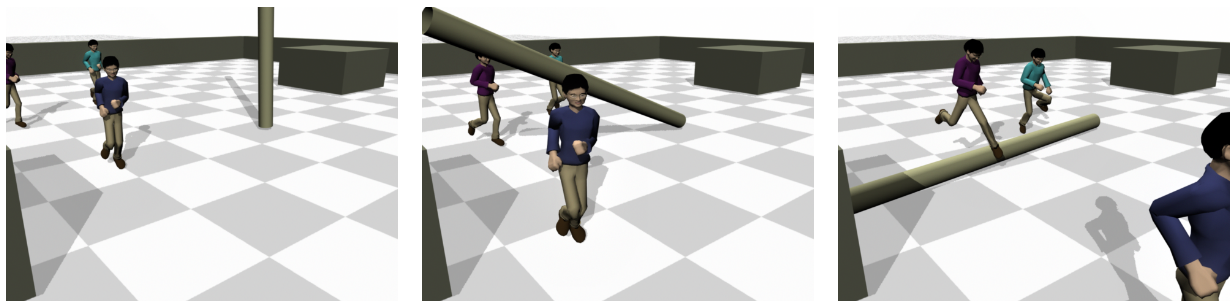


Figure 3.4: A dynamic environment with a falling tree. *Left:* Before it falls, the characters are free to jog normally in the open space. *Center:* As it is falling, the characters can neither jog past nor jump over it. *Right:* After it has fallen, the characters can jump over it.

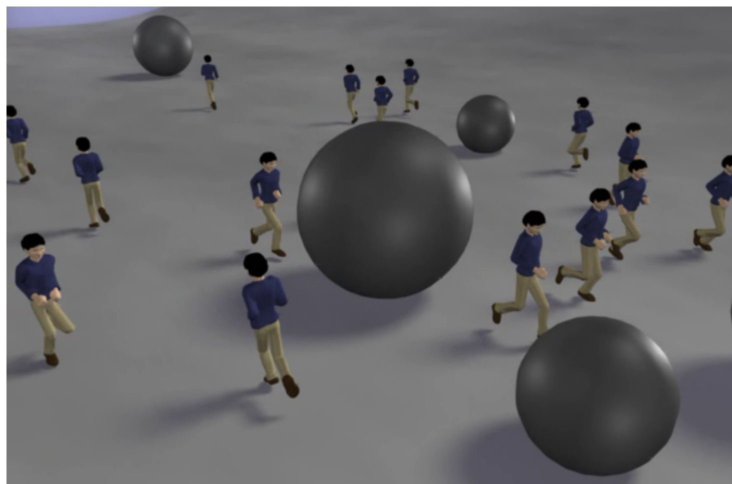


Figure 3.5: The characters avoid each other and the dynamic obstacles (spheres).

Figure 3.5 shows an example of twenty characters simultaneously avoiding each other and a number of moving obstacles. The motions of the moving obstacles are pre-generated from a rigid-body dynamics solver. Their positions at discrete time steps are then automatically stored into the time-indexed gridmaps representing the dynamic environment. In the animation, the characters appear to move intelligently, planning ahead and steering away from the moving obstacles in advance.

3.7 Discussion

We have presented a *Behavior Planning* approach to automatically generate realistic motions for animated characters. We model the motion data as abstract *high-level behaviors*. Our behavior planner then performs a global search of a data structure of these behaviors to synthesize mo-

tion. The main **contribution** of our approach is in the abstraction of motion data such that we can apply planning techniques to efficiently generate goal-driven navigation motion for virtual human-like characters. We have shown that our approach works well for generating navigation motions. It is able to generate motions for many characters navigating simultaneously in complex and dynamic environments. In addition, because of our small graph size, the search time is fast.

As there is a lot of previous work in the areas of planning and animation, we view our approach as one motion planning method among a spectrum of methods (Figure 3.1). Our approach **differs from previous methods** in that it has a carefully chosen planning space. This spectrum of methods should be well understood before one method is chosen among them.

The main **limitation** of our approach is that we need to manually build a behavior graph, and have motion data that is appropriately segmented. We have found that it is not difficult to construct and re-use our graphs because of their small size. However, if we have a large data set that we wish to use, then it might be more appropriate to consider motion graph techniques. In addition, we assume that we are given a set of segmented and blendable motion clips as input. The motion clips have to be easily blendable into the other ones, in order for them to be blended smoothly into longer sequences.

Another limitation is that the output sequence must be a concatenation of the input clips, since no changes to the original input motion clips are made in our current system. Hence our planner does not allow the virtual character to exactly match precise goal postures. Our focus in on efficiently generating complex sequences of large-scale motions across large and complex terrain involving different behaviors. Given a small number of appropriately designed "go straight", "turn left", and "turn right" actions, our planner can generate motions that cover all reachable space at the macro-scale. No motion editing is required to turn fractional amounts or traverse a fractional distance because we are computing motion for each character to travel over relatively long distances (compared to each motion clip). The algorithm globally arranges the motion clips in a way that avoids obstacles in cluttered environments while reaching distant goals. The character stops when it is within a small distance ϵ from the goal location. If matching a precise goal posture is required, motion editing techniques [13, 109] may be used after the blending stage.

One important **insight** of our work is that the abstraction of motions as high-level behaviors and the carefully chosen planning space lead to both the strengths and weaknesses of the approach. The weakness is that we require segmented and blendable motion clips corresponding to the high-level behaviors. The strength is that since the number of these behaviors and input clips are small, the search algorithm is fast and works well for generating navigation motions. In addi-

tion, we empirically found that using an extremely small data set is enough for generating many types of navigation motions. For example, having five to seven jogging behaviors is enough to generate navigation motions for many characters in large environments.

A possible direction for **future work** is to parametrize the nodes in the graph. Instead of having a "jog left" behavior state, we can have a "jog left by x degrees" state. Such a state might use interpolation methods [88, 108] to generate an arbitrary turn left motion given a few input clips. This can decrease the amount of input data needed, while increasing the variety of motion the planner can generate. We can also have behaviors such as "jump forward x meters over an object of height h". This would allow our system to work in a larger variety of environments.

Our behavior graph has a compact and structured form. This is important to the approach presented in the next chapter, which vastly improves the efficiency for finding a sequence of motion clips that allows a character to reach a goal.

Chapter 4

Precomputed Search Trees

In the previous chapter, we described a method for autonomously generating the motions for multiple characters navigating in a complex environment, given that we have a set of pre-segmented motion clips. This previous technique applies traditional A*-search methods. It builds a search tree during runtime and performs a forward search to find a solution path. We refer to this as a *forward search* as it can solve problems with one specific start location and one goal location, and a tree is built in the forward direction from the start towards the goal. However, the search can be very slow if there are many characters and we have to build a tree to perform a search for each character. In our examples in the previous chapter, this search cannot be done in real-time if we have about 20 or more characters. The overall approach of taking one starting location and one goal location for each character, and performing some kind of search to find a solution is typical to traditional forward search methods.

The goal of the work in this chapter is to develop a method to generate these motions much faster than before, so that we can have a real-time framework for multiple characters navigating in large and dynamic environments. This is an important problem for real-time graphics applications because existing methods often do not scale well to a large number of characters, especially if we want these characters to exhibit human-like motions.

This problem of how to develop a real-time framework motivated us to try to speed up our *Behavior Planning* approach by exploring what we can compute in advance and what we have to do during runtime. Instead of performing a forward search during runtime to solve the planning problem with one starting location and one goal location, the key idea of our new approach [53] is to first compute this search tree beforehand without considering the obstacles and start/goal locations, given that we have some pre-segmented motion clips. We then use the precomputed tree to efficiently find a solution for any configuration of obstacles and any start/goal queries. The runtime process performs a *backward search*, as it begins from the goal and attempts to

backtrace a valid path towards the start. Hence we make a tradeoff of memory for speed: we have to use extra memory to store this tree, but we get a much faster runtime for our search. We found that, by doing this precomputation, we get a runtime that is two orders of magnitude faster than traditional forward search methods.

There has been much work [6, 16, 21, 77, 78, 89] that first computes a map of the environment before motions are actually generated. This can be considered as a type of precomputation, but these maps are usually built for a given environment and the maps have to be rebuilt for different environments. More "dynamic" versions of these maps have been developed [26, 96, 111]. The key difference here is that they build a map given the environment. In our *Precomputed Search Trees* approach, instead of starting with the environment, we start from the motions that are available to the character and build a tree of the motions. We then deal with the obstacles only when necessary as the character moves about in the environment. This has a number of advantages: (i) we can re-use the same tree for arbitrary environments, for any obstacle configuration and goal position in the same environment, and for different parts of a large environment; (ii) we can use the same tree for all the characters in the scene; and (iii) we can handle dynamic obstacles on-the-fly instead of taking them into account beforehand.

The contributions of our work in this chapter is twofold. First, we present a novel planning approach based on precomputation: we precompute a search tree of possible motion paths and then use a backward search method during runtime to solve planning queries. We experimentally show that this approach is more than two orders of magnitude faster than traditional forward search methods. Using our approach, we built an interactive system where multiple characters continuously navigate and respond to user changes to obstacles and goal positions. Second, we present a simple but effective randomized-based technique for precomputing large and diverse trees. As there have been a lot of recent work on the topic of path diversity, we experimentally compare the advantages and disadvantages of our technique with other methods for building diverse trees.

4.1 Problem Statement and Overview

The problem setup is the same as in the previous chapter. The inputs to the system include: a starting position and orientation, a goal position, a description of the environment geometry, and a behavior graph of motion clips. In this chapter, we focus on synthesizing sequences of motions efficiently. Figure 4.1 shows an overview of our system. The sections in this chapter discuss each of these parts in detail:

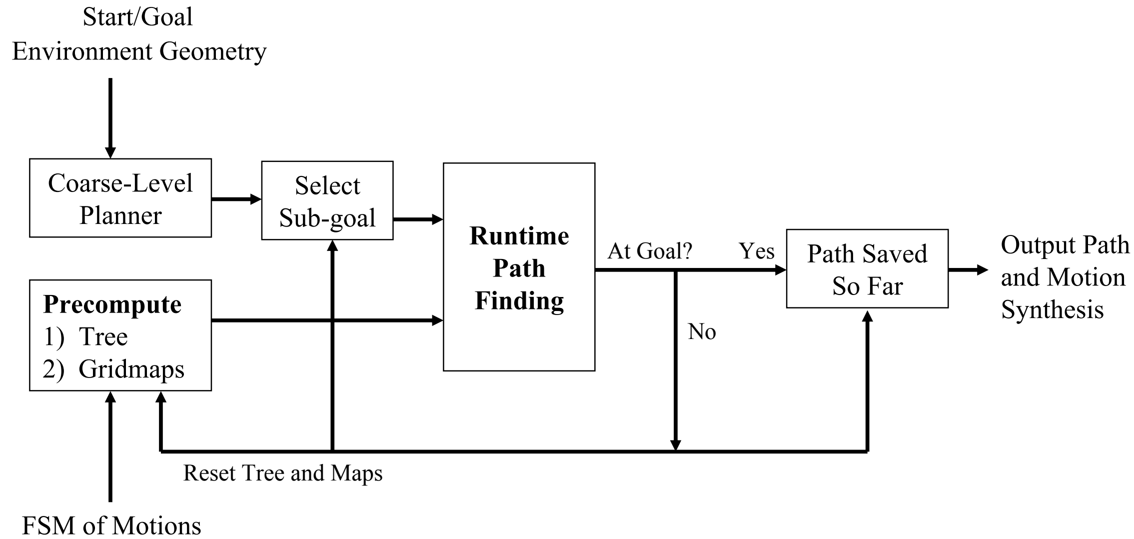


Figure 4.1: Overview of the system.

Precomputation of Tree. In the precomputation phase, we build a search tree of the nodes in the behavior graph. At this stage, we do not take into account the obstacles and goal positions. We first describe the exhaustive tree and the original "pruned" version [53] of the tree. We then describe a simple but effective randomized-based technique to build scalable and diverse trees.

Precomputation of Environment and Goal Gridmaps. This is also a precompututation step. We compute gridmaps over the tree so that we can access the nodes and paths in the tree more efficiently later.

Mapping Obstacles to Environment Gridmap. This is the first main runtime step. The locations of the obstacles are mapped to the environment gridmap, which allows for more efficient collision checks during the runtime backward search.

Runtime Backward Search. This is the second main runtime step. This phase first selects a sub-goal (which can just be the final goal) that we want to reach. We perform a backward (from goal to start) search in the precomputed tree to find a sub-path that leads to the sub-goal. This sub-path is added as part of the final solution.

Coarse-Level Planner. This is an optional step needed for distant goals. This step is performed

only once for each goal query, and precedes the two main runtime steps. If the final goal is far away from the starting location, we first use a bitmap planner to search for a coarse path. This coarse path is then repeatedly used to select intermediate sub-goals for use in repeated iterations of the runtime phase. For each sub-goal, the two main runtime steps are performed.

Evaluation. We present an application where the user can move the obstacles and the goal locations, and the virtual characters will respond to these changes interactively. We compare our tree precomputation methods with previous methods for building similar types of diverse trees. We compare our overall precomputation approach with traditional forward search methods.

4.2 Precomputation of Tree

This is the first precomputation step. We do not take into account the obstacles and goal positions yet. We take the behavior graph of motions, and compute a tree of nodes of these motions. Each path (of nodes) in the tree corresponds to a sequence of motions (if we take the corresponding motion clips and concatenate them). This tree represents all the possible paths and motions that the character can take starting from any position and orientation.

In all three cases described below, each node of the tree has the position, orientation, cost, and time of the path up to that node starting from the root node. Since each node has only one parent, we can trace back the path to the root to find the sequence of behavior nodes that can reach that point. Hence each node also represents the path up to and including that point. Each node also has a *blocked* variable, initialized here with *UNBLOCKED*. If this variable is set to *BLOCKED*, this means we know for sure that we can neither reach that point nor any of the corresponding descendant nodes. However, the path from the root to the parent node of that point may still be reachable. This *blocked* variable is used in the runtime steps.

We describe three cases of tree precomputation: (i) the exhaustive tree includes all nodes up to a certain depth level; (ii) the "pruned" tree includes a subset of the exhaustive nodes; and (iii) the scalable and diverse randomized-based tree has diverse paths that scale to large environments.

4.2.1 Exhaustive Tree

We build a tree of all nodes up to a certain depth level. In general, the tree size is $O((average\, b)^d)$, where b is the branching factor and d is the depth level. If we build the exhaustive tree (Figure 4.2 left), its size grows exponentially with respect to depth level.

The exhaustive tree finds optimal solutions, since it includes all paths up to a depth level. However, this also means that the solutions are limited by that depth level. In practice, the path

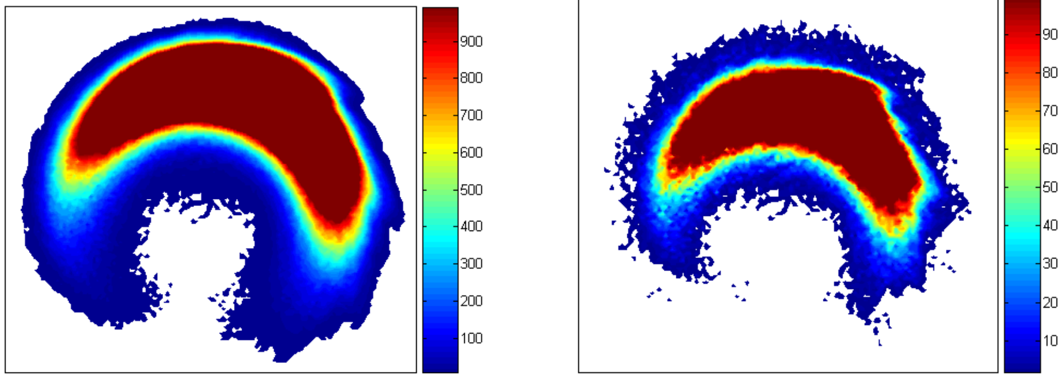


Figure 4.2: Frequency plot of the precomputed tree. Each point represents the number of paths that can reach that point from the root of the tree. The root is near the middle of each figure and the tree progresses in a forward direction (or up in the figure). The tree covers an area that is approximately a half circle of radius 16 meters, with the character starting at the center of the half circle. The majority of paths end up in an area between 8 and 14 meters away from the start. We used about 1,500 frames of motion at 30 Hz. **Left:** Exhaustive tree of 6 depth levels built from graph with 21 behavior nodes. This tree has over 6 million nodes (over 300 MB). **Right:** The pruned tree has 220,000 nodes (about 10 MB).

sizes are small (ie. up to 5 or 6 depth levels). The memory used to store a tree is large, up to about 1 GB in our examples. Therefore, we do not recommend using the exhaustive tree. We only discuss this case here for completeness.

4.2.2 Pruned Tree

The motivation for having a pruned tree is that the exhaustive case requires a very large memory and is not practical for actual use. The “pruned” version is the tree we developed in [53], and used in our interactive character system. We use the term “pruned” tree because the nodes are a subset of the ones in the exhaustive case. However, the tree construction process starts from an empty tree and adds one node at a time.

We initialize the tree with an empty root node. We also initialize an empty grid similar to the one in Figure 4.3(left). To build the nodes in the $d + 1$ level, we consider all the child nodes of the nodes in the d level. We must consider the transitions (as specified in the graph) when we build this set of child nodes. We randomly go through this set of child nodes, and decide if each one should be added. If the node’s corresponding cell (x_i, y_i) (see Figure 4.3 left) has less than a prespecified number of k nodes already in it, we will add the node and increment the number of nodes in that cell. We thereby limit the number of nodes in each cell to k. In Figure 4.2(right), k is set to 100. We also limit the total number of nodes we built.

4.2.3 Scalable and Diverse Randomized-based Tree

Motivation. We have already used the "pruned" tree in the overall precomputation approach to show that the concept of precomputation can be useful [53]. However, two issues remain with the "pruned" version of the tree.

The first issue is that the pruned trees are so small (typically up to 5 or 6 depth levels) that it is either not possible or difficult to use them in real planning problems (requiring solutions of up to 50 levels in our experiments). This is also the case for previous work that builds trees with diverse paths [10, 30]. Furthermore, these methods [10, 30] do not scale to the time and memory needed for precomputing trees of a reasonable size (ie. even just 8 depth levels or greater). These methods require a set of paths to prune from, and it is not clear how we can get this set to begin with. Building the exhaustive tree for even a small depth level is extremely memory intensive, and starting with a subset of paths to prune from is problematic as it is not clear where this subset comes from. Hence we would like to generate *scalable* trees. By *scalable*, we refer to the large size of the tree, in terms of the length of the paths. These trees should also scale to large environments.

The second issue is that many of the paths in our pruned trees are still somewhat redundant, as they may cover similar regions or they may partially overlap with each other. We would therefore like to build trees with more *diverse* paths. By *diverse*, we intuitively mean that the paths should be evenly scattered around the region that they cover. To achieve this intuitive idea, we provide a density metric and greedily minimize this metric. We empirically show that this simple idea leads to precomputed trees that can solve more randomly-generated planning queries than previous methods for building diverse trees.

In this subsection, we show how to precompute a tree that has diverse paths and that can scale to large environments. Although our method is simple, it has many advantages. We can precompute a tree with a more efficient computation time than the algorithms that have been previously proposed [10, 30]. Our precomputed tree can solve more planning queries than trees built with previous methods, given the same amount of memory for storing the tree. We can build a tree for any memory size available for storing it, and we can build a tree that can cover a region of arbitrary shape and size.

Algorithm. Our algorithm precomputes a search tree by incrementally adding a node and its corresponding edge to the tree. We use a "density" metric to essentially scatter the edges or paths of the tree evenly throughout the region that we would like to build the tree in.

We use the following notation in the algorithm below. Let A be the set of actions. Recall

Algorithm 2: Precomputation of Scalable and Diverse Tree

```
   function Trace(e)
1  m_temp.Init();
2  foreach (x, y) ∈ Path(e) do
3      map_x = Map(x);
4      map_y = Map(y);
5      m_temp(map_x, map_y) = 1;
   end
6  return m_temp;
   function PrecomputeTree(A, K, R)
7  T.Init(n_root);
8  m.Init();
9  d_overall ← 0;
10 for k = 1 to K do
11     n_near ← Nearest(N, α(k));
12     Δd_best = FLT_MAX;
13     a_best = NULL;
14     foreach a_j ∈ A do
15         if ¬Transition(n_near.action, a_j) then continue;
16         if T.AlreadyExpanded(n_near.childs, a_j) then continue;
17         e ← T.SimulateAddChild(n_near, a_j);
18         if OutsideRegion(R, Trace(e)) then continue;
19         m_current ← m ⊕ Trace(e);
20         Δd_current ← (Density(m_current) − d_overall) / Length(Trace(e));
21         if Δd_current < Δd_best then
22             Δd_best = Δd_current;
23             a_best = a_j;
           end
       end
24     if a_best == NULL then continue /* do not increment k */;
25     e ← T.AddChild(n_near, a_best);
26     m ← m ⊕ Trace(e);
27     d_overall ← Density(m);
   end
28 return T;
```

that we are given whether or not each action can transition to other actions. For every i, j (i can equal j), if action a_i can transition to action a_j, $Transition(a_i, a_j)$ is true. Otherwise, it is false. Let T be the precomputed tree, n be each node of T, and N be the set of all nodes. Each node corresponds to one action, denoted by $n.action$. $n.childs$ denotes the child nodes of node n. Let e be each edge of T. Every time we add a new node n to T, we also add a corresponding edge that connects n and its parent node; we refer to this combination as a node/edge. We refer to the path that the action of a node covers as a *traced path*; more details are given below when we describe the $Trace()$ function. Let m be a 2D grid that covers the region occupied by the tree.

Our algorithm can be summarized as follows: we iteratively add a node and its corresponding edge to T. At each iteration, we "randomly" select which node in the existing T to expand from, and then use a density metric to locally decide which child of the selected node to expand. This iterative strategy is greedy and leads to non-optimal solution paths. However, since it is not possible to precompute large-scale trees that can provide optimal solutions, we choose a strategy that is fast and provides near optimal solutions (as shown later in our experimental evaluation).

The intuition for the density metric is that since the tree is precomputed for any obstacle and any start/goal queries, a simple but effective way to increase the likelihood of finding a solution is to "scatter" the paths evenly in the region that T should be built in.

We now describe Algorithm 2 in more detail. In the PrecomputeTree() function, K is the number of nodes to be built in T, and is a parameter that can be set depending on the memory available for storing the tree. R is a 2D region that we want to build the tree in, and can be arbitrarily large and be in any shape. Note that the obstacles and goal are not taken into account during precomputation. The function starts by initializing T with a root node (n_{root}), which is a placeholder node that contains no action and can transition to all other actions. This root node is initialized with position $(0, 0)$, orientation 0 and total cost 0. It initializes the grid m by setting all its gridcells to 0. This grid provides a discretized "count" of the space that the tree covers. $d_{overall}$ is the density (described below) measure of m. We incrementally select a node/edge to add to T. $\alpha(k)$ is a randomly-sampled point in R, and $Nearest()$ selects the node in the existing set N that is nearest to $\alpha(k)$. This randomly-sampled selection scheme is the same as in RRTs as we explained in Chapter 2. We then try to add a child node to n_{near}: we choose to associate this child node with an action whose traced path locally minimizes the density measure if that path is added {lines 14-23}. The $AlreadyExpanded()$ function checks to see if a_j is already a child of n_{near}. $SimulateAddChild(n_{near}, a_j)$ simulates the effect of adding a new node representing a_j as a child node of n_{near}. It does not add the new node and its corresponding edge to T here; instead it returns information about the corresponding edge (which is represented by e on line 17). The Trace() function marks all the gridcells covered by $Path(e)$. $Path(e)$ {line 2} takes an edge e that connects a parent node and a child node, and generates the "traced" 2D path of motion if we start from the overall position at the parent node and take the action at the child node. $Path(e)$ then returns a set of discretized 2D points passing along this path. These points have to be chosen so that we neither generate too many points and make the algorithm inefficient, nor generate too few points and have them not cover all the gridcells that the path covers. $Map()$ {lines 3-4} maps from the coordinate system of the action/motion space to the coordinate system of the grid m_{temp}. m_{temp} has the same shape and grid structure as m. To avoid accessing all the cells of m_{temp} in each execution of $Trace()$, m_{temp} is initialized once in the algorithm, and the (map_x, map_y) points are saved for reseting m_{temp} each time. Once we have $Trace(e)$, $OutsideRegion()$ returns $true$ if at least one of the gridcells marked by $Trace(e)$ is outside of R. R has the same grid structure as m. The $\oplus$ operator performs component-wise addition to the grids. $Density(m)$ takes a grid m (which does not have to be rectangular shape) with the "count" in the gridcells labelled c_i (i from 1 to $ncells$), and computes the "density" of the paths in the tree:

$$Density(m) = \sum_j \left[\left(\frac{\sum_i c_i}{ncells} - c_j \right)^2 \right] \quad (4.1)$$

Intuitively, a smaller density value means that the paths are more evenly spread out. The algorithm locally minimizes this density metric to achieve this. This density metric actually measures uniform density, as minimizing this metric corresponds to having paths that evenly or uniformly spread out. $Length()$ is the distance that the "traced" 2D path travels. Since we have discretized this path, we compute the number of gridcells that the discretized set of 2D points cover. The reason for dividing by the length is to normalize for the length of the traced path when considering the density value. $AddChild(n_{near}, a_{best})$ adds a new node representing a_{best} as a child node of n_{near}, and also adds the corresponding edge.

In Figure 4.2, we showed the frequency plots of the exhaustive and pruned tree cases. We do not show the plot of the scalable and diverse tree here as the plot simply has uniform density and therefore has a constant color. Note that this is the case by construction as we wanted to add paths to the tree in a such way that they would be evenly spread out. As we add more nodes and paths to the tree, the density increases but the overall plot remains to have approximately uniform density.

Properties of Tree. We show that our algorithm satisfies several desirable properties.

The execution time of the tree precomputation is $O(K(log\ K + \|A\| * F))$. The $log\ K$ term comes from the nearest neighbor computation. F is due to a faster way to compute the $Density(m)$ value: instead of iterating through all the gridcells of m, we keep a frequency count of the values in each gridcell and compute $Density(m)$ using this information. F is the largest value with at least a count of one; it starts at 0 and increases as k increases. F is a function of K, R, the cell sizes of m, and A (the space that each action covers). In practice, the $K * \|A\| * F$ term is more significant than the $K\ log\ K$ term. In our experiments, the largest K we used is about $2e6$, A is about 10-20, and the largest F we have is about 500.

Given a specific precomputed tree, our approach is not complete. However, we have a weaker notion of "complete"-ness *with respect to the given tree*: if there is a solution *in the precomputed tree*, the algorithm will find it in finite time; if there is no solution *in the precomputed tree*, it will stop and report failure in finite time.

We now show that given enough time (and memory), all the nodes in the exhaustive tree will eventually be expanded. We define $Exh(d)$ to be the exhaustive tree with finite depth d and finite average branching factor b. We define a notion called **Probabilistic Expansion**:

$$\lim_{k \to \infty} P\ (\ n_i\ will\ be\ expanded \mid n_i \in Exh(d)\ ) = 1 \tag{4.2}$$

where d can be arbitrarily large. Algorithm 2 follows this notion of Probabilistic Expansion.

Proof: We prove by contradiction: there is at least one node, n, that is not expanded. If n's parent node is not expanded (but exists in the exhaustive tree), we instead set n to be its parent node. We continue this until n's parent node is expanded. We now have an unexpanded node n whose parent node p is expanded. We must always at least have such a case because the tree's root node must be expanded at the $k = 1$ iteration. Let $\mu()$ be the measure of volume in a metric space [56] and $V(p)$ be the Voronoi region of p. We must have $\mu(V(p)) > 0$ regardless of the number of nodes in the current tree and k, since the tree has a finite size. Let the branching factor of p be b, which is finite. As $k \to \infty$, we must eventually sample $V(p)$ b times (recall that we only sample from finite region R), and n must be expanded.

4.3 Precomputation of Environment and Goal Gridmaps

This is the second precomputation step.

Environment Gridmap. The intuition for this gridmap is that we will map the obstacles to this grid and thereby the tree during runtime. The discretized grid is then used for efficient runtime collision checks. We build an environment gridmap over the tree as shown in Figure 4.3(left). The gridcells are all initially marked as *UNOCCUPIED*. Each node of the tree can then be associated with a gridcell. For example, node i corresponds to cell (x_i, y_i) in Figure 4.3(right). We precompute and explicitly store the corresponding (x_i, y_i) value in each node so that we can quickly access the cell that a node is in during runtime. This is important for the efficiency of the algorithm.

The size of the gridcells is a parameter of the system, and we used a range of sizes from 14 cm to 28 cm. This parameter, however, affects the runtime phase significantly. It may increase the time for mapping the environment to the tree (Section 4.4) by approximately nine if the size of each cell is cut to a third of the original. We want to balance between a large cell size which decreases the runtime, and a small cell size which represents the environment more accurately.

For each tree node i, we also precompute and store the values $x_{midtime_i}$ and $y_{midtime_i}$ (Figure 4.3 right). We first take half (with respect to time duration) of motion clip i to reach node $midtime_i$. The values $x_{midtime_i}$ and $y_{midtime_i}$ are then stored in node i. Node $midtime_i$ is used temporarily in this calculation and does not exist in the tree. If the motion clip is one of the special motions such as jumping, we do not take half of the clip to get node $midtime_i$. Instead

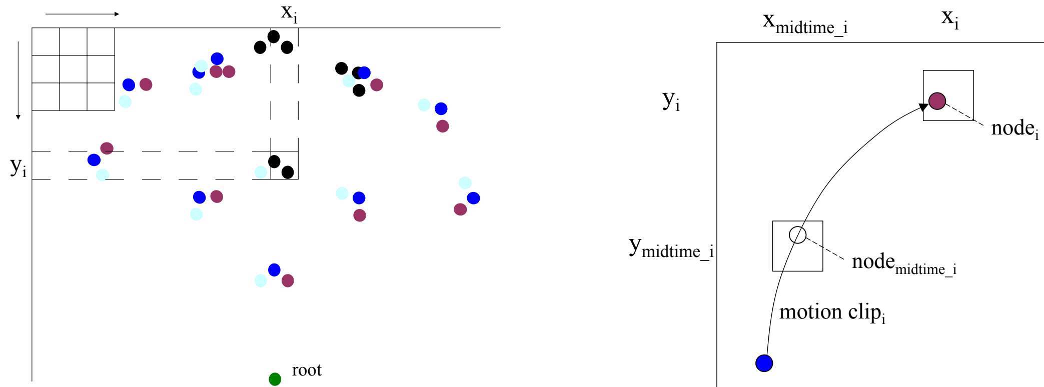


Figure 4.3: **Left:** An *environment gridmap* initialized with *UNOCCUPIED* cells. The intuition for this gridmap is that if cell (x_i, y_i) is occupied by an obstacle, the tree nodes corresponding to this cell and their descendant nodes (the black ones) are *BLOCKED*. **Right:** For each node i, we precompute and store the corresponding values x_i, y_i, $x_{midtime_i}$, and $y_{midtime_i}$.

we use the point where the special motion is actually executed. For the example of jumping, this is where the character is at or near the highest point of its jump. This information should already be pre-annotated in the motion data. These "midtime" positions are used for collision checking in the runtime phase. For the special motions, they are used to see if the character successfully passes through the corresponding special obstacle. The choice of taking half of the clip is only a discretization parameter. For more accurate collision checking, we can continue to split the clip and compute similar information. We choose only the "midtime" discretization because our motion clips are short in length (hence midtime in enough), and a smaller discretization gives a faster runtime.

Goal Gridmap. The intuition for the goal gridmap is that we want to start searching with the lowest cost path first during runtime. We precompute a goal gridmap (Figure 4.4 left) used in the runtime backward search phase (Section 4.5). For every node in the tree, we place it in the corresponding cell in the gridmap. Each cell then contains a sorted list of nodes (Figure 4.4 left). We used a range of cell sizes from 45 to 90 cm. The environment gridmap does not explicitly store this list of nodes, but the goal gridmap does.

Figure 4.4(left) shows why a straight-forward discretization of the goal gridmap may not work well. In this case, the node representing the "best nearby path" is close to the goal. However, it was placed into a nearby gridcell because of the discretization of the space. Another path is returned as the solution even though in some cases the "best nearby path" is clearly better. This happens when the goal is close to the boundary of the discretization. The goal gridmap itself cannot be adjusted or changed easily during runtime because it is precomputed and we do

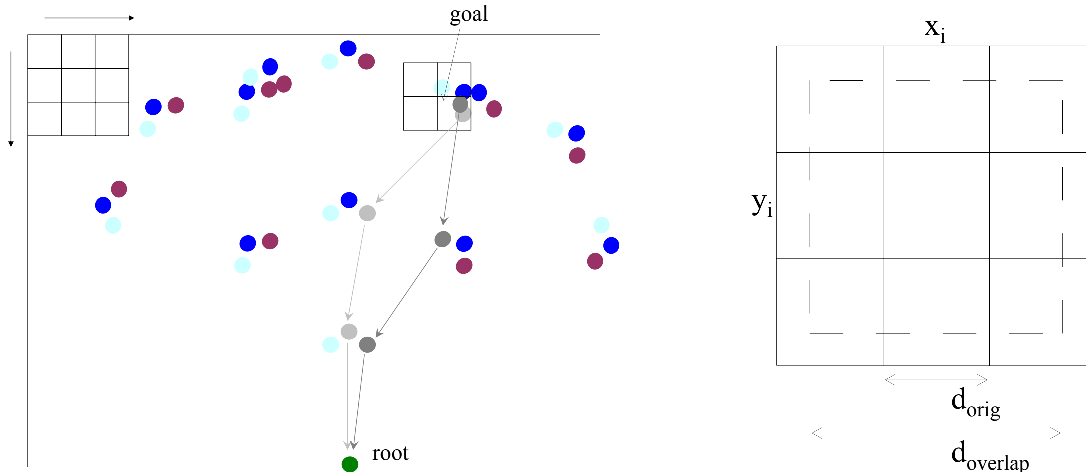


Figure 4.4: **Left:** In the *goal gridmap*, each cell contains a sorted list of paths. Each path's total cost is the sum of the cost of the motion states. The sorting is based on this total cost. Since each node in the tree corresponds to a unique path if we trace the node back towards the root of the tree, we can also say that each goal cell contains a sorted list of nodes. We will use this gridmap during runtime; the intuition is that if we know the gridcell that the goal position is in, the paths or nodes in that cell correspond to the potential solutions. **Right:** A straight-forward discretization of the *goal gridmap* may not work well. An "overlapped discretization" works well.

not know the goal position beforehand. Figure 4.4(right) shows our solution to this problem. For each cell (x_i, y_i), we extend the size of the cell from d_{orig} to $d_{overlap}$. We then place the nodes of the tree that are in the extended cell into cell (x_i, y_i). This means that some of the nodes will be placed into more than one cell. Our values of d_{orig} are between 45 and 90 cm, and $d_{overlap}$ are between 105 and 210 cm.

4.4 Mapping Obstacles to Environment Gridmap

This is the first main runtime step. The precomputed tree and the environment are in different coordinate spaces. We must first align these spaces. We do not transform the tree towards the space of the environment because there is much more information in the tree and this would be more costly. Instead, we translate and rotate the starting position (of the character in the environment) to match the root node of the precomputed tree, and the starting orientation (of the character) to face the forward direction of the tree's root node (Figure 4.5 left). All the obstacles in the environment are translated and rotated similarly.

We then map each transformed obstacle to the environment gridmap. We want to mark each

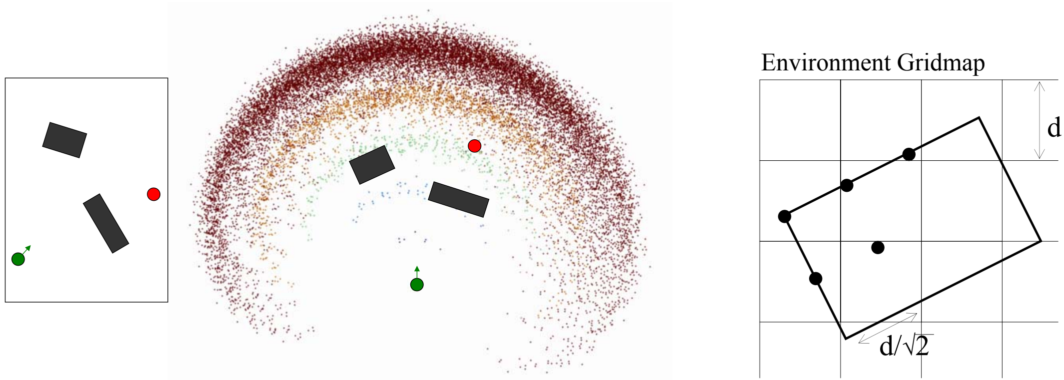


Figure 4.5: **Left:** We align the coordinate spaces between the environment and the tree. We translate and rotate the obstacles and the goal position so that the starting position and orientation (of the character in the environment) match with that of the precomputed tree. **Right:** If the size of the gridcell is d, we can guarantee that the mapping of an obstacle to the environment gridmap is correct if the sampling of points for the obstacle is at most $d/\sqrt{2}$ apart.

cell of the environment gridmap as either *OCCUPIED* or as a *valid region* of a special obstacle. If an obstacle is outside the region covered by the tree, we can safely ignore it. Otherwise, we map it to the environment gridmap by iterating through a discretized set of points inside the obstacle (Figure 4.5 right).

If a gridcell in the environment gridmap is *OCCUPIED*, we know that the tree nodes in that cell are *BLOCKED*. But in order to save time, we will not mark them as such until it is necessary to do so in the runtime backward search step. In addition, the indices of each gridcell that gets marked as being occupied are saved as the mapping proceeds. We use this information to quickly reset the environment gridmap every time we re-execute this mapping process.

4.5 Runtime Backward Search

This is the second main runtime step. If the final goal is within the region covered by the tree, we can execute the runtime path finding algorithm just once. If the final goal is further away, we first use the bitmap planner (Section 4.6) to generate a coarse solution path. We then repeatedly use this coarse path to select sub-goals, which are used in repeated iterations of the runtime path finding algorithm.

Sub-goal Selection. This part is only necessary if we use the coarse bitmap planner. The coarse level path has many points that we can use as sub-goals. Intuitively, we would like to find ones that will be within the dark red regions (Figure 4.2) of the precomputed tree. We choose the

```
Algorithm 3: RUNTIME PATH FINDING
   Goal_GoalGrid ← T(Goal_Global)
   P ← GoalGrid[Goal_GoalGrid.x][Goal_GoalGrid.y].Nodes()
   foreach p ∈ P do
       while ( (p.BLOCKED == false) and
             (EnvGrid[p.x_i][p.y_i] == UNOCCUPIED) and
             (p != rootNode) ) do
1          p.BLOCKED ← true
           // midtime collision check
           if isSpecialMotion(p.motionState) then
2              if EnvGridmap[p.x_midtime_i][p.y_midtime_i] !=
                 specialObstacle(p.motionState) then
                   continue to next p
               end
           else
3              if EnvGridmap[p.x_midtime_i][p.y_midtime_i] ==
                 OCCUPIED then
                   continue to next p
               end
           end
           p ← p.parent
       end
       // this path traced through before
4      if p.BLOCKED == true then continue to next p
       // this path is blocked by an obstacle
5      if EnvGridmap[p.x_i][p.y_i] == OCCUPIED then
6          p.BLOCKED ← true
           continue to next p
       end
       // reached rootNode and path found
7      return node representing current path
   end
   return no path found
```

sub-goal to be the point in the coarse path that is closest to a fixed distance away from the start (Figure 4.6). A distance between 10 and 12 meters worked well in our examples. Note that the start is different for each iteration of the runtime path finding phase.

Runtime Path Finding Algorithm. Algorithm 3 takes a goal position as input, and returns the tree node that represents the solution path. If there is no possible path in the precomputed tree that can reach the goal, it will recognize that there is no solution (among the possible paths in the tree). The inputs also include the precomputed tree, the environment gridmap, and the goal gridmap. The goal position is first translated and rotated from its global coordinates to the coordinate system of the goal gridmap (function T). The transformed indices are used to find P, the list of nodes sorted in increasing cost. We then go through each node p in P, and try to trace

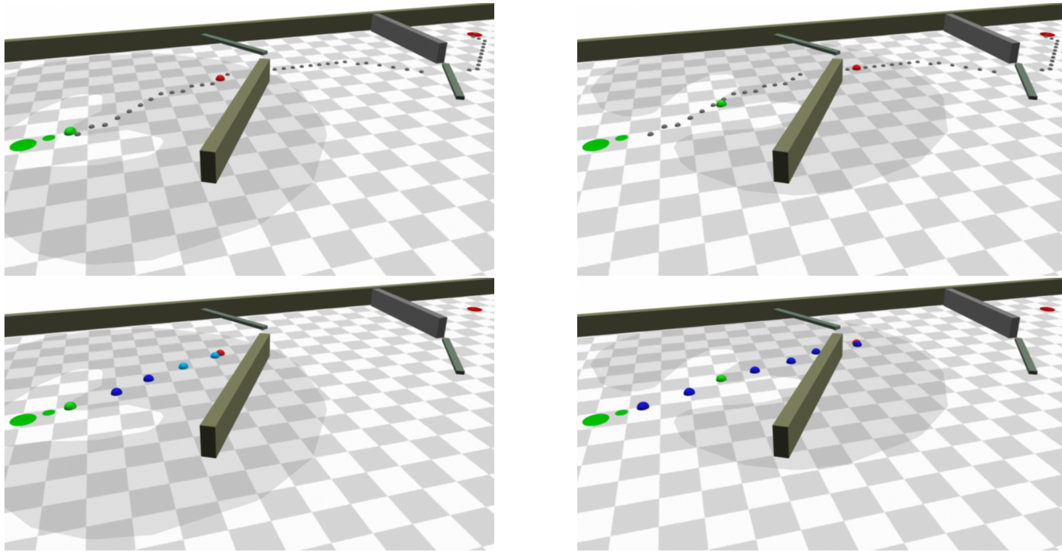


Figure 4.6: The 2 columns correspond to the first 2 iterations of the runtime path finding phase for this example. The top row shows the start (green sphere) in each iteration, and the sub-goal (red sphere) selected from the coarse-level path. The bottom row shows the path returned by the runtime path finding algorithm (light and dark blue) and the partial path chosen (dark blue only). An estimate of the outline of the precomputed tree is shown. The tree is transformed to the global space only in the figure to show how it relates to the other parts of the environment. There is only one precomputed tree, and it is never transformed to the global space in the algorithm.

it back towards the root node (which is where the start is in the current iteration). As we trace back towards the root, we mark each node as *BLOCKED*, if it is not already *BLOCKED* or not obstructed by an obstacle. The intuition behind this is that we want to find the shortest path that is not obstructed in any way. We also check to see that the "midtime" point of the motion clip reaching that node is not obstructed (line 3) before tracing back to its parent node. Furthermore, if we have arrived at that node through a special motion (line 2), we check to see if the motion successfully goes through the corresponding special obstacle by checking to see if the "midtime" point is a *valid region* of that type of special obstacle. The $specialObstacle()$ function returns the type of this corresponding obstacle. If the "midtime" point is obstructed in any way, the algorithm will continue to the next possible node in P.

There are three conditions under which each trace of node p towards the root node stops (Figure 4.7):

1. Pointer p arrives at a node that is obstructed by an obstacle (case 1 in Figure 4.7(top) and line 5 in Algorithm 3). When this happens, the path from the root node to p cannot be a solution. We mark that node as *BLOCKED* (these are the black nodes in Figure 4.7 and also the ones marked *BLOCKED* in line 6) and proceed to test the next node in P.

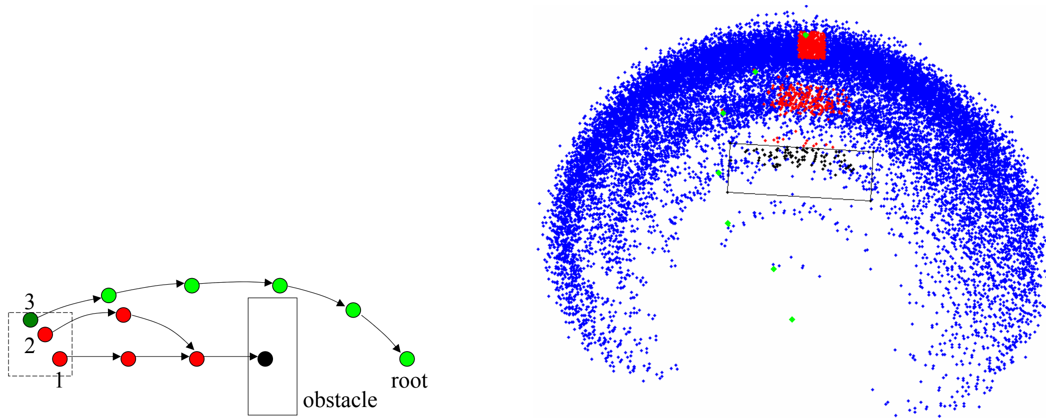


Figure 4.7: The process of tracing back the list of sorted nodes P towards the root node in Algorithm 3. **Left:** The 3 cases under which each trace of node p stops. The sub-goal is inside the dashed square (a cell of the goal gridmap). **Right:** Simple example. The blue nodes are the nodes of the precomputed tree. The sub-goal is somewhere in the square-shaped box of red nodes. The other colored nodes correspond to the 3 cases.

2. Pointer p arrives at a *BLOCKED* node (case 2 and line 4). When this happens, the path from the root node to p also cannot be a solution. The algorithm then continues to test the next node in P. The red nodes in Figure 4.7 are the nodes that were traced back. These are the ones that are marked as *BLOCKED* in line 1 of Algorithm 3.

3. Pointer p arrives at the root node (case 3 and line 7). The green nodes correspond to the nodes traced back for this p. Note that these are also marked as *BLOCKED*, but it does not matter since we already have the solution. We stop testing the list of nodes P, and return the original node that p points to in this case (the darker green node in Figure 4.7 top) to represent the solution path. If we have gone through the whole list P without having reached the root node, there is no possible solution.

The solution path from the algorithm is in the coordinate system of the precomputed tree. We must therefore transform (T^{-1}) each node in the path back to the global coordinate system. The bottom row of Figure 4.6 shows examples of the algorithm's output.

Before running a new iteration of the path finding phase, we need to *UNBLOCK* all the nodes of the precomputed tree. As we run Algorithm 3, we save the number of nodes in P that were examined. To "reset" the tree, we go through the same number of nodes in P again (in the sorted order). For each node p, we trace the path back towards the root node and *UNBLOCK* every node along the path, stopping when p arrives at an already *UNBLOCK*-ed node. We are therefore able

to only traverse the nodes that were *BLOCK*-ed.

Properties of Runtime Path Finding. The execution time of this algorithm is $O(N_{goal_{largest}} * d_{largest})$. $N_{goal_{largest}}$ is the largest number of nodes/paths in one gridcell among all the cells of the goal gridmap, given a precomputed tree. It is typically in the thousands, and up to a few tens of thousands. $d_{largest}$ is the largest depth in the given precomputed tree. It typically lies between ten and fifty. This execution time explains the efficiency of the runtime search.

The runtime backward-tracing is a "lazy" way to discover nodes that cannot be reached for a given goal and obstacle configuration. In addition, the runtime search checks through *the smallest number of nodes* with respect to a specific precomputed tree and a planning query, if we use a backward search approach. This is because: (i) we only need to search through the nodes/paths that are in the cell of the goal gridmap that the goal is in; (ii) we have to check the least cost nodes/paths first because we want to return the least cost solution; and (iii) we must go through at least the nodes in cases 1 and 2 of the runtime search (the red nodes in Figure 4.7) to check for potential obstacle collisions. The validity of the third point is not obvious. Recall that we first map the obstacles to the region covered by the tree during runtime. At that point, we could have identified all the nodes that are blocked by the obstacles, and the descendant nodes of these nodes since they are also blocked by the obstacles (we cannot get to these descendant nodes from the root of the tree). Instead we discover these blocked nodes lazily, by applying cases 1 and 2 of the runtime search. The nodes that these two cases go through must be at least less than the total number of "blocked" nodes. Finally, it is possible that a forward search (from start towards goal) can lead to a smaller number of nodes we have to check. However, our backwards search approach is simpler to implement because there is only one unique path towards the tree's root as we move from the goal back towards the start. Hence we only need to keep a pointer to the parent node of each node, and move one pointer along the nodes to check through them.

Partial Paths. This part is only necessary if we use the coarse bitmap planner. In Section 4.6, we annotate the points in the coarse-level path that appear just before the special obstacles and the final goal. These annotations are used to allow the character to not get too close to these obstacles or the final goal before re-executing the runtime phase. Intuitively, if there is a large obstacle just beyond the current planning horizon, the runtime algorithm may generate a path that allows the character to move just before this obstacle. In the next iteration, it may be too close to it that there is nowhere to go given the motion capabilities of the character. To avoid this issue, we keep only a part of the solution path so that each iteration can better adjust to the global environment.

More specifically, if the current sub-goal is not annotated as being near a special obstacle or

the final goal, we keep the whole solution path. Otherwise, we only keep the first two nodes of the solution path (Figure 4.6 bottom). In addition, if the solution path includes a special node, we take all the nodes up to and including it. This is an optional adjustment that again helps the character in adjusting itself before executing a special motion. Our experience shows that without this adjustment, the algorithm may sometimes inaccurately report that there is no solution.

Furthermore, we have to make sure that the last motion state of the path we keep and the root node of the precomputed tree can transition to the same states. If this is not the case, we need to add an additional state or leave out the last one. This is because the next iteration of the path finding phase uses the same precomputed tree and therefore starts at the root node. Since the majority of our motion states transition to each other, this is not a major concern.

Motion Synthesis. The path finding phase eventually returns a sequence of motion states that allow the character to navigate from the start to the goal. This sequence is converted to character motion by concatenating together the motion clips that represent the states. For the frames near the transition points between states, we linearly interpolate the root positions and apply a smooth-in/smooth-out slerp function to the joint rotations. The joint rotations are originally expressed as euler angles. They are converted into quaternions, interpolated with the slerp function, and converted back into euler angles.

4.6 Coarse-Level Planner

If the goal position lies within the space covered by the precomputed tree, we can apply the runtime path finding module once and immediately find a solution path. However, the precomputed tree has a finite size and the goal position can be further away than the region covered by the tree. Hence in general, we first use a fast bitmap planner to generate a coarse-level path from the start to the goal. This path is then used as a guideline for picking sub-goals to run each iteration of the runtime path finding phase. In our implementation, we used a bitmap planning algorithm [48] optimized for 2D grids.

A coarse-level map of the environment is used as input to the bitmap planner. The size of the gridcells was about 70 cm. We map the obstacles to this coarse gridmap using the technique discussed in Section 4.4. In particular, we only use the regular obstacles in this step, and not the special ones (ie. ones that the character has to jump over). A path is returned that may go through these special obstacles. This is fine since the character has the motion capabilities (ie. jump) to go through these obstacles when we perform the runtime path finding step.

The special obstacles are then added to the coarse-level map. We can eliminate parts of the

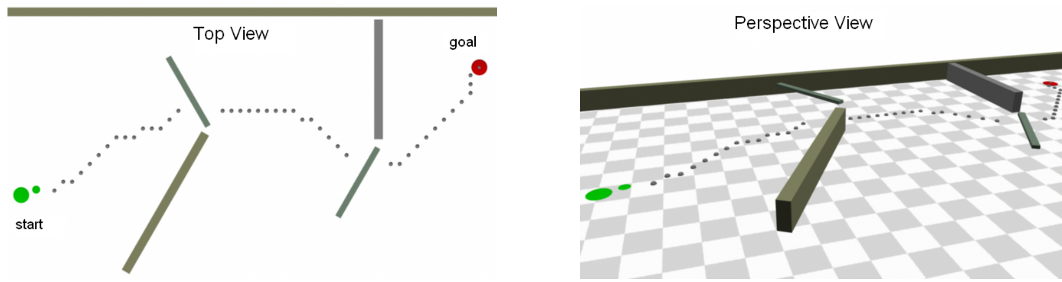


Figure 4.8: The points of the path that are eventually chosen by the coarse-level planner for this environment.

returned path that collide with these obstacles. In addition, we annotate the parts of the path that appear just before the special obstacles, and the parts that appear just before the final goal position (but not the final goal position). Figure 4.8 shows an example of the points in the path that are eventually chosen. Note that in this example, there is an obstacle that the character must duck under, and one that it must jump over.

4.7 Evaluation

First, we demonstrate the effectiveness of our algorithm by building an interactive system where multiple characters can navigate and respond to user changes to the obstacles and goal positions. Second, we empirically compare our scalable and diverse tree to other previous tree precomputation methods. Third, we empirically compare our overall precomputation approach and traditional forward search methods. Fourth, we explore the effect of different grid resolutions on the runtime cost and success rate of finding a solution.

4.7.1 Interactive System

We describe our system and additional issues we have to handle for synthesizing multiple characters. The motions for multiple characters are generated in real-time. We do not generate the full path of each character at the beginning. For each character, we execute a "runtime path finding" phase to synthesize the next partial path only after we start rendering the first frame from the previous partial path. Hence the characters can re-plan and respond to user changes to the environment as the simulation is running.

Algorithm 4 shows the pseudocode of the planning and rendering system. We specify a maximum planning time ($T_{planning}$), and plan as many characters as possible within each iteration of the draw loop. The exception is that we plan for every character once when we execute the

Algorithm 4: MULTIPLE CHARACTER INTERACTIVE SYSTEM

```
   Initialize environment
   Precompute tree and gridmaps
   // draw loop
   while true do
       Read Input
       if input detected then
           Update environment state
       end
       // planning loop
       while current planning time < T_planning do
           Advance to next character
           if all characters planned in current planning loop then
               Stop current planning loop
           end
1          if ready to plan current character then
2              Generate next partial path/motion
               Store results in buffer
           end
       end
       foreach character do
           Draw pose from buffer data depending on time
       end
   end
```

planning loop for the first time. We are *ready to plan the current character* (line 1 of Algorithm 4) after we start rendering the first frame (of the current character) from the previously planned partial path.

To *generate the next partial path/motion* (line 2), we execute the runtime path finding phase. We run the coarse-level planner with the updated starting location as the end of the last partial path. The sub-goal selection and runtime iteration is done just once, since we only need the first partial path here. We use the same precomputed tree for all the characters, so we reset the gridmaps and precomputed tree after each character's runtime iteration.

We also precompute the blending frames. The characters' poses are blended at the transition points between motion clips. We precompute the blending frames for all possible pairs of motion clips so we can efficiently use them at runtime. We place the correctly blended poses in the data buffer as we store the results.

In addition, we need to deal with collision avoidance between characters. We apply these 3 steps. First, we use a *global characters gridmap* to store the time-indexed global positions of the characters after their poses are stored to the data buffer. These positions are placed into the correct bucket in the gridmap for efficient access later. Second, in addition to mapping the obstacles to the environment gridmap (Section 4.4), we map the characters' positions to a *local characters gridmap* using a similar procedure. During the runtime iteration, we select the

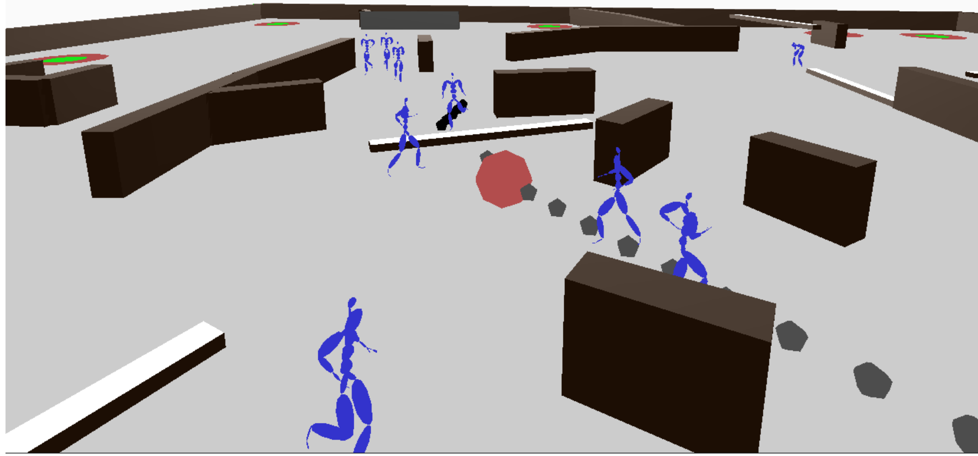


Figure 4.9: Screenshot of the interactive system. The characters interactively respond to user changes to obstacles and their respective goal locations while navigating in a large environment.

cells in the global characters gridmap that are relevant in each local sub-case. This assures that the collision check between characters is linear in the number of characters, instead of being quadratic in the naive case. The positions are also placed into the appropriate bucket in the local gridmap. Third, as we trace through the tree nodes in Algorithm 3, we check to see if each node can collide with the locally relevant characters. An additional test is needed in the *while* loop that performs a Euclidean distance check between the node's position and each of the relevant characters' position. With the use of the local characters gridmap, this step is fast because we rarely have to perform a distance check.

Each iteration of the runtime phase takes about 8.5 ms. The fast runtime allows us to build an interactive application. Our interactive system (Figure 4.9) demonstrates the following strengths of our method: (i) multiple characters following navigation goals and avoiding obstacles that the user can interactively modify; (ii) the ability to incorporate behaviors such as jumping and ducking, so that the characters are not limited to navigating on a flat terrain; and (iii) interactive motion synthesis of up to 150 human-like characters.

4.7.2 Comparison of Tree Precomputation Methods

As there has been recent work on the topic of path diversity, we explore these algorithms and experimentally compare them with our tree precomputation method. We compare our algorithm (Algorithm 2) with four previous methods (including our original PST method). The key here is to only compare the trees that are built. We use the same runtime backward search (our method) with the different trees to solve planning queries. This is because the previous methods only describe how to build trees, but not how to use them to solve planning queries. For the experiments

Method	% success						precomputation time (sec.)						density value				path cost	
	250	100	50	25	10	5 KB	250	100	50	25	10	5 KB	50	25	10	5 KB	mean	std
SPST	**73.44**	**71.75**	**69.48**	**67.88**	**54.55**	**39.38**	1.2	0.8	0.6	0.5	0.4	0.4	**28,752**	**11,403**	**4,162**	**1,650**	110.09	8.80
original PST	60.37	40.30	39.12	21.92	21.16	6.32	**0.043**	**0.021**	**0.014**	**0.011**	**0.009**	**0.008**	109,039	47,976	11,098	5,542	103.79	10.40
Branicky et al. [10] I-P	65.09	54.38	46.04	40.22	23.02	10.37	720	126	38	14	7	5	66,270	23,059	6,960	3,298	**102.19**	**2.28**
Branicky et al. [10] I-E	35.33	26.22	21.84	10.79	7.76	4.72	780	138	49	21	11	9	120,155	42,329	13,347	5,309	121.87	20.13
Green and Kelly [30]	68.89	66.27	62.48	57.93	48.82	31.87	11460	1800	480	80	19	6	55,019	16,526	4,285	1,841	112.02	8.22

Table 4.1: Comparison of Tree Precomputation Methods.

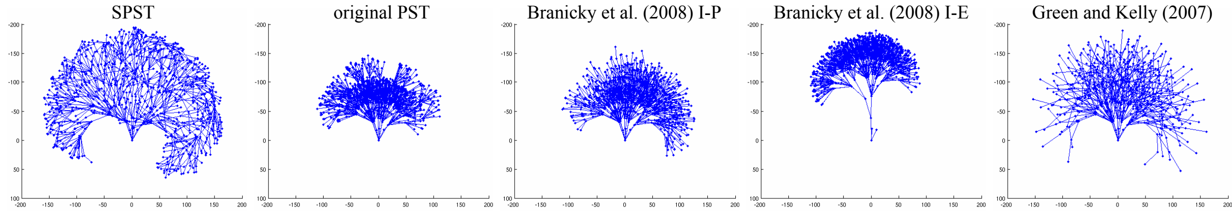


Figure 4.10: Examples of precomputed trees used in our comparison. All trees have the same number (826) of nodes. Each tree's root is at (0,0), and the paths move in a forward (or up in the figure) direction because the input actions/motions allow the character to move forward and/or slightly turn left/right. Note that many paths overlap because of the tree's structure.

in this subsection, we are only able to build trees that fit in a relatively small environment. This is because it is not clear how we can use the previous methods to build large trees.

We generate a large number of random planning queries and try to use the trees precomputed with the different methods to solve them. We select a fixed starting position and orientation, and generate random goal positions. This is equivalent to generating random start/goal queries. Since we build an exhaustive tree with 5 depth levels with which to select paths from for the purposes of some of the other tree building methods, these methods can only solve queries within the region covered by the exhaustive 5-level tree (we let R be this region for SPST, which explains the tree's shape for SPST in Figure 4.10). Hence we select random goal positions within R so we can perform a fair comparison. We generate obstacles randomly by randomly generating the number of obstacles, the positions and orientation of each one, and the sizes of each one given that they have a rectangular shape. Each obstacle must at least overlap with R. We use the same set of random queries for all of the methods; we did not include queries where the start and/or goal collide with obstacles.

In Table 4.1, all the methods use the same runtime backward search technique (our technique described above), since the last three methods only provide algorithms to build the tree; the difference is in the tree precomputation technique. SPST is the *Scalable* and diverse version of our tree. "original PST" is the technique in Lau and Kuffner [53]. "I-P" stands for Inner-Product and "I-E" stands for Inclusion-Exclusion. For the last three methods in the table, we first built the exhaustive tree with 5 depth levels and then selected a subset of paths using each method. Note that SPST can have paths with depth levels larger than 5; for the last three methods, the

exhaustive tree with larger depth levels cannot be built because of its size, and it is not clear how to pick a subset of potential longer (than depth 5) paths to choose from. For all methods, we tried to choose parameters that give the best results. We build trees with varying sizes: the numbers in the top row are the memory in KB that we use to store the tree. We use the same amount of memory to store each node of the tree for all methods, so the trees for each column has the same number of nodes. "% success" is the % of the 1186 total planning queries that can be solved. We also tried to solve this set of queries with the exhaustive tree of 5 depth levels. It took about 2 MB to store this tree and the % success rate was 71.16. The percentages for SPST can be higher than 71.16 since the precomputed trees for SPST can have paths longer than 5 depth levels. The "precomputation time" is the time for building the trees only. We used a 2.4 GHz machine with 1 GB of RAM. The "density value" is from the $Density()$ formula. The "path cost" columns are for the 50 KB case; we have similar results for the other cases. We took the queries (248 of them) where all methods found a solution and compared the costs of these solutions. We normalize the costs for the exhaustive tree case (the optimal case) to be 100, and normalize the other costs correspondingly. We then computed the mean and standard deviation of all the normalized costs for each method (so 100 % is optimal).

The results show that, based on the % success rates, the ranking of the methods starting with the best is: SPST, Green and Kelly [30], Branicky et al. [10] I-P, original PST, and Branicky et al. [10] I-E. This is true for all memory sizes. This is a significant result, as one way to say that a tree has diverse paths is to show that it can handle many types of environments. We have shown experimentally that our SPST method can handle more randomly-generated planning queries (or environments) than other methods. The precomputation time for SPST is longer than that for original PST. However, the precomputation can be done beforehand, and the time for SPST is still reasonable. In contrast, the other three methods' times are significantly slower; their times increase at such a rate that it is difficult to use them in practice for large trees, and we chose to build trees with depth levels of 5 (very small) for this set of experiments just so we can compare the methods. The density values justify our use of the density metric. A smaller density value tends to correspond to a higher % success rate, which matches our intuition that scattering the paths of the tree evenly is more likely to lead to a precomputed tree that can solve more planning queries. The tradeoff of SPST here is that it provides non-optimal, but near-optimal solutions.

Figure 4.10 explains some of the results in Table 4.1. "original PST" and "Branicky I-P" tend to keep shorter and thereby smaller-cost paths. On the other hand, "Branicky I-E" seem to prefer longer paths, and hence their solutions are likely to be further away from optimal. "Green and Kelly" build trees that has more diverse paths. However, its precomputation time is the longest, and is not practical for trees of large sizes. SPST builds the most diverse trees

in the sense that their paths are spread out over the region R, in this case the region covered by the 5-level exhaustive tree. Our results show that: *our simple and randomized-based method is efficient and can outperform other tree precomputation methods based on its ability to solve randomly generated planning queries*. This suggests that *the effectiveness of sampling-based methods also applies to our paradigm of motion planning with precomputation*, although this was not an insight that we originally tried to show.

We would like to mention one recent work on the topic of path diversity. Erickson and LaValle [19] describe an approach to build sets of diverse paths based on a survivability metric. This metric tries to decrease the likelihood that paths will be obstructed by the same obstacle. For example, two paths that mostly overlap with (or are close to) each other will not be as preferred to two paths that cover different regions. It is interesting to note that our simple density metric also implicitly tries to achieve this. As this previous work is only recently published, we are unable to implement their method to compare it with ours. However, this shows that the issue of path diversity is also of concern to others, and it would certainly be one possibility of future work to compare their method with ours.

4.7.3 Comparison between Precomputation Approach and Traditional Forward Search

We explore the benefits and tradeoffs of the overall precomputation approach along with the runtime backward search, as compared to traditional forward methods. The significance of this comparison is that we believe it would useful to understand what we have gained and/or lost from using our overall precomputation approach and backward search. We specifically compare with A*-search methods, which are common forward search methods that provide optimal solutions. We would like to compare the differences between the traditional "forward" search method and our new "backward" search method. By "backward" search, we refer to the runtime search that starts from where the goal location is, and perform a search of the paths in the precomputed tree from the goal back towards the starting location. This is in contrast to "forward" search which builds the tree during runtime from the start to the goal. For the experiments in this subsection, we use relatively larger environments since we are able to build trees of a much larger scale.

We generate random planning queries as before, except that we use a much larger R region and generate a larger number of obstacles. We created one additional test environment with a C-shaped obstacle (similar to the "deep local minima" example in [14]). The random queries contain a mix of simple and complex cases, and this C-shape obstacle case is a complex example with local minima. Since A*-search and SPST search in different directions, we place the

start/goal positions differently in the two cases so that the direction is always moving "into" the C-shape, which makes the problem more difficult.

Method	runtime	collision checks	% success	path cost
A*-search	100.00	100.00	97.91	100.00
wA* (w=2)	79.42	12.31	97.91	105.12
SPST (50 MB)	0.47	7.27	94.76	113.80
SPST (25 MB)	0.44	4.23	93.63	115.48
A*-search	2,411,505	2,885,740	N/A	786
wA* (w=2)	1,276,468	1,559,632	N/A	846
SPST (25 MB)	461	210	N/A	884

Table 4.2: Comparison between Precomputation Approach and A*-search Methods. Top set of results: from random planning queries. Bottom set: from C-shaped obstacle case.

In Table 4.2, SPST took 199 seconds to precompute the 25 MB tree and 477 seconds to precompute the 50 MB tree. The "runtime" of SPST is only for the runtime backward search. "collision checks" is the number of collision checks performed. "% success" is the % of 1774 total queries that each method found a solution for. The top set of results are all percentages. We took the queries (1661 of them) where all methods found a solution and compared the runtime, collision checks, and path cost of these solutions. We normalize these values (runtime, collision checks, path cost each separately) for the A*-search case (the optimal case) to be 100, and normalize the other values correspondingly. We then computed the mean of all the normalized values for each method, and reported these means in the table (top set). The bottom set of results are actual values. The runtime in that case is in μs.

The main benefit of SPST over A*-search methods is the significantly faster runtime (>200 times for the random planning queries). SPST has fewer collision checks than A*-search, although a more greedy version (weighted A*) can also lead to fewer collision checks. The main tradeoffs of SPST are that it gives up completeness and optimality of A*-search. Completeness can be seen in the "% success" column: SPST's rates are a few % smaller. The "% success" of SPST must be smaller than that of A*-search, because SPST is only able to find solution paths that are in the precomputed tree. Hence it is encouraging that SPST is only slightly worse here. Optimality can be seen in the "path cost" column: SPST's path costs are near-optimal, and usually about 10-15 % higher than the optimal costs. In general, as we increase the memory size of the tree, the % success rate increases and the path cost % decreases to the "optimal" percentages. The user can adjust the tree's memory size to explore this tradeoff. The purpose of the C-shaped obstacle case is to make sure that the better results do not just come from simple queries in the random set. This is true as SPST achieves an even faster runtime and fewer number of collision checks for this case.

We have shown the advantages and disadvantages of our precomputation approach compared to A*-search methods. We view our precomputation concept as one approach that can be consid-

ered among various planning methods. It is important to understand the tradeoffs of our approach before deciding to use it.

4.7.4 Effect of Grid Resolution

The obstacle avoidance between the characters and the objects in the environment depend on the grids that we use. There are several grids placed over the tree and the environment. The grid resolution is an important parameter that affects our results. We therefore empirically study the effect of different grid resolutions on the runtime cost and success rate of finding a solution, for A*-search and SPST. We also intuitively explain the failure cases of both methods in more detail.

We used the same experimental setup as for the comparison between A*-search and SPST above. We changed only the grid resolution and kept the other variables the same. The grid resolution here refers to the one for the Environment Gridmap; we adjust the resolution for the other gridmaps accordingly.

	runtime						% success		
Method	270x270		540x540		1080x1080		270x270	540x540	1080x1080
A*-search	100.00	104880	100.00	151770	100.00	340195	97.91	97.91	97.91
SPST (25 MB)	0.41	333	0.52	690	0.80	2652	93.63	94.31	94.76

Table 4.3: Effect of grid resolution on runtime cost and success rate.

Table 4.3 shows the results from our experiments. The success rate is the percent of 1774 total queries that each method found a solution for. For the runtime results, there are two values in each entry. The first value is a percentage, and the second one is the average time for the success cases in μs. To compute the percentages, we took the queries where both methods found a solution and compared the runtime of these solutions. We normalize these values for the A*-search case (the optimal case) to be 100, and normalize the other values correspondingly. We then computed the mean of all the normalized values for each method, and reported these means in the table.

In general, we found that a finer grid resolution leads to an increase in runtime. This makes sense intuitively as the time for mapping the obstacles to the grid takes longer. We also found that a finer grid resolution leads to an increase in the success rate. Intuitively, as the obstacle representation gets finer, there is more space that is represented as collision free, and there is a higher chance that more paths become collision free.

It is interesting to discuss the failure cases and what causes them. For A*-search, there are cases where there are no solutions because of two reasons. The first reason is that the obstacle configuration does not allow the goal to be reached. The second reason is the obstacles are cluttered enough that there is no path that can reach the goal given the existing motion clips. For

SPST, there are cases where there are no solutions because the precomputed tree does not have all the potential paths (given the existing motion clips). In fact, the number of paths is much less than the number for the A*-search case, as A*-search can build an exhaustive tree to cover the free space in the environment. It is therefore reasonable to expect the success rate for SPST to be smaller. The fact that it is only a few percentages smaller shows that the precomputed case performs quite well. We would like to discuss one specific case where A*-search found a solution but SPST did not. In this case, the A*-search solution was not long-winded, but it requires the right combination of motion clips to get to the goal (as the environment was cluttered with many obstacles). SPST did not find a solution because it did not have that specific combination of motion clips in the already computed tree.

4.8 Discussion

We have developed a "Precomputed Search Trees" technique for interactively generating realistic animations for virtual characters navigating in a complex environment. The main **contributions** are twofold. First, we introduce a novel planning approach based on precomputation: we first precompute a search tree of possible motion paths and then use a backward search method during runtime to solve planning queries. We show that our approach is more than two orders of magnitude faster than traditional forward search methods. Second, we present a technique for precomputing scalable and diverse trees, and explore the advantages and disadvantages of our method compared to previous methods for building diverse trees.

While there has been previous work on computing information about the environment or character motions in advance for use during runtime, the main **additional value of our work over previous work** is that we show a *complete* system that demonstrates the concept of precomputation for motion planning: we show how to precompute large and diverse search trees; we describe an efficient runtime backward search method for solving planning queries; we use these methods in actual planning scenarios and show runtime results; and we have an interactive system with many characters navigating in complex environments using our approach.

Some **limitations** for precomputed search trees are similar to our behavior planning approach. We assume that we are given a set of blendable and segmented motion clips as inputs. This is again not a major concern as the number of behaviors and motion clips are small. An output sequence is limited to be a concatenation of the input clips. We have to use other existing methods for motion editing if this is needed.

Another limitation of precomputation is that we give up completeness and optimality of the solution, compared to A*-search. The benefit from this is the two orders of magnitude runtime

speedup (also compared to A*-search).

One **insight** we have gained from our work is that precomputation is certainly a viable approach for motion planning. However, there is a tradeoff between memory, runtime speed, and optimality. These issues should be considered before choosing between precomputation and traditional search methods.

Another insight is that our randomized-based diverse tree works surprisingly well in terms of being able to solve planning queries, even though the method is simple. Our method and also previous methods for tree precomputation are greedy. This makes sense since it is computationally intensive to analyze path sets of large sizes. The lesson to learn here is that if we assume the method is going to be greedy, a randomized-based approach is a simple one that works well. Spending a lot of time to compute information about what paths to pick might lead to over-analyzing in the sense that much time can be spent without improving the result.

There has been more and more recent work in the topic of path diversity. We believe that this is an important issue as well, and more **future work** can be done on this topic. More generally, computing path sets that are diverse in advance is one essential component of the concept of precomputation. However, most previous work do not use these precomputed path sets to actually solve planning problems. We believe that further experiments (perhaps similar to ours) can be done to compare work on precomputing path sets based on their ability to handle different types of environments.

Chapter 5

Modeling Spatial and Temporal Variants in Motion Data

Variation in human motion exists because people do not perform actions in precisely the same manner every time. Even if a person intends to perform the "same" action more than once, each motion will still be slightly different. However, current animation systems lack the ability to realistically produce these subtle variations. For example, typical crowd animation systems [66] utilize a few walking motion clips for every walking cycle and every character of the simulation. This can lead to synthesized motions that look unrealistic due to the *exact repetition* of the original walk cycles. Hence a variation model that can generate even slight differences of the original walk cycles has the potential to greatly improve the naturalness of the output animations. Crowds in games and films [99] also do not produce human-like variations. Films use a technique known as "Cycle Animation": animators use a fixed number of motion cycles to create the motions of multiple characters. Inevitably, there will be cycles that are *exactly repeated* both spatially (in multiple characters) and temporally (at different times for the same character). As soon as even one example of repetition is identified, the whole animation can be immediately deemed un-human-like. This can make games and films less fun and interesting.

There is much interest in the problem of adding variety to virtual crowds. Maim and his colleagues [64] take a fixed number of template *character meshes*, and vary them by changing their color and adding different accessories to them. On the other hand, our work takes a fixed number of template *motions* and synthesize new variant motions from them. McDonnell and her colleagues [67] perform user experiments to study the perception of clones in virtual crowds. The focus of their work is to study the perception of appearance and motion clones, and to provide insights on how to make it less likely for the end-user to detect such clones assuming that clones are being used. In contrast, our approach takes input data, learns a model from the data, and

synthesizes new motion variants with the model. Our approach creates motion with *no exact clones*, even though the new variants can be visually similar. We start by assuming that it is possible to generate variation in motion such that exact motion clones are not necessary.

For the problem of generating variation in motion, previous methods consider variation to be an additive noise component. This is not robust for automatically generating animations. There are methods to add noise to existing motion [8, 75], but there is no guarantee that the added noise will match well with the existing motion.

We believe that variation should not be just an additive noise component; instead, we take a data-driven approach to this problem. Given a small number of examples of a particular type of motion (ie. cheering, walk cycle, swimming breast stroke) as input, we learn a model from the input data, and use this model to synthesize spatial and temporal variants of that motion. We claim that the Dynamic Bayesian Network (DBN) [24, 27] model solves this problem well as it provides a formal and robust approach to model the distribution of the data. A DBN represents a multivariate probability distribution of the degrees-of-freedom of motion, and it is this distribution from which we sample to synthesize our new variants. In addition, one advantage of our approach is that it can handle a small number of input examples. This is useful as it is difficult to acquire a large number of examples of a particular motion. Another advantage is that no post-process smoothing operation is needed, which is beneficial as such an operation may smooth out details of motion that our method generates. There are three major steps for learning a model and synthesizing new variants. First, we learn the structure of the DBN with the input examples. We use a greedy algorithm based on a variant of the Bayesian Information Criterion score to learn a good structure. Second, we use the learned structure and the original data to synthesize new variants. Third and optionally, we can use an inverse kinematics method developed within our DBN framework to satisfy any foot and hand constraints.

The key result of our method is that we can take a few examples of a particular type of motion as input, and produce an unlimited number of spatial and temporal variants as output. A new variant is spatially different as all new poses are distinct from those of the input examples, and temporally different as the timing of the whole motion is distinct from the input examples. The new variants are *statistically and visually similar* to the inputs, but are *not exact copies*. We demonstrate our approach with a variety of full-body human motion data. The memory requirement of our model consists of only the space required to store the few input examples. Most of the processing time is in the learning phase; the runtime for synthesizing new variants is very efficient and can be done as a continuous stream one frame at a time. To evaluate our approach, we perform a user study to show that: (i) our new variants are just as natural as motion capture data, and (ii) our new variants are less repetitive than “Cycle Animation”. In addition, we

demonstrate that "just adding noise" to existing motion can create poses and timings that look obviously awkward. We show this with two methods to add noise to motion: (i) a naive/strawman method, and (ii) the Perlin noise function. Finally, since our input examples have to be similar (so that we can model their variation), it is useful to know what we mean by "similar" and how we get them to begin with. Hence we provide a DBN-based method to take examples from raw data, and reduce them to a small number of examples that can be used well with our approach.

5.1 Problem Statement and Overview

The inputs to our problem are a few examples of a particular type of motion, and the outputs are the spatial and temporal variants (of the inputs). Let the inputs be $X_i[j]^{(l)}$, where there are l motion sequences (usually four), i is an index for DOF, and j is an index for time. We build a model for the joint distribution of the inputs. Let this model be $\{\mu(X_i), var(X_i), G_{prior}, G_{trans}\}$, where $\mu(X_i)$ and $var(X_i)$ are only for the nodes X_i in G_{prior} with $\mathbf{Pa}(X_i) = \emptyset$, and G_{prior} and G_{trans} are the prior and transition networks in the DBN model. In addition, we need to keep the original data $X_i[j]^{(l)}$ because both the learning and synthesis processes require non-parametric regressions. Let a synthesized output motion sequence be $Y_i[j]$, where j does not have to be the same length as the input times.

The following is an overview of the major parts of the rest of this chapter:

Dynamic Bayesian Network. We specify the notations and definitions of a Bayesian Network and a Dynamic Bayesian Network.

Structure Learning. We explain the details of our learning method. Given a small number of motion clips that represent variations of a particular motion, we search for the structure of the prior and transition networks of a DBN model automatically. We describe our non-parametric regression method for computing conditional probability distributions; this is used during both learning and synthesis.

Synthesis of New Variants. The learned model and original input data are used to generate any number of spatial and temporal variants of the inputs.

Constraints. We develop an inverse kinematics framework that fits with our DBN model to satisfy foot and hand constraints.

Evaluation. We show results for full-body human motion data. The same approach is used for

all types of data. We discuss the memory requirements and performance time of our technique. We perform a user study to evaluate our approach. We discuss our experiments with adding noise to motion data.

Inputs that work well with Our Approach. The main limitation of our approach is that the inputs have to be "similar" to begin with. It is difficult to define what is meant by "similar" motions. Instead, we develop a method to characterize the inputs that would work well with our approach.

5.2 Dynamic Bayesian Network

We first describe the basic formulation and notations for a Bayesian Network (BN) model, and then extend this description to a Dynamic Bayesian Network (DBN) model [24, 27].

A BN is a directed acyclic graph that represents a joint probability distribution over a set of random variables $\mathbf{X} = \{X_1, ..., X_n\}$. Each node of the graph represents a random variable. The edges represent the dependency relationship between the variables. A node X_i is independent of its non-descendants given its parent nodes $\mathbf{Pa}(X_i)$ in the graph. This conditional independency is significant because we only use the values of parent nodes of X_i to predict the value of each X_i. This graph defines a joint probability distribution over $\mathbf{X}$ as follows:

$$P(X_1, ..., X_n) = \prod_i P(X_i \mid \mathbf{Pa}(X_i)) \tag{5.1}$$

A DBN models the process of how a set of random variables changes over time. It represents a joint probability distribution over all possible trajectories of the random variables. Figure 5.1 shows an example. In our case of human motion, X_i is the trajectory of values of the ith-DOF of the motion, and $\mathbf{X}[t]$ is the set of values of all the DOFs at time t. $X_i[t]$ is the value of the ith-DOF at time t. We have 62 DOFs in our motion data, so n is 62. The prior network G_{prior} represents the joint distribution of the nodes in the first two time points, $\mathbf{X}[0]$ and $\mathbf{X}[1]$. The transition network G_{trans} specifies the transition probability $P(\mathbf{X}[t+2] \mid \mathbf{X}[t], \mathbf{X}[t+1])$ for all t. Note that the transition network predicts the values at time $t+2$ given those at t and $t+1$. Hence there are no incoming edges into the nodes at time t and $t+1$. In the usual formulation of DBNs, there can be edges between the nodes in time $t+2$. In our case, we do not allow edges between these nodes. We find that this simplification does not affect our results, since we assume that an edge from $X_1[t+1]$ to $X_2[t+2]$ has the same effect as an edge from $X_1[t+2]$ to $X_2[t+2]$. We assume that the trajectories satisfy the second order Markov property: the values at $t+2$

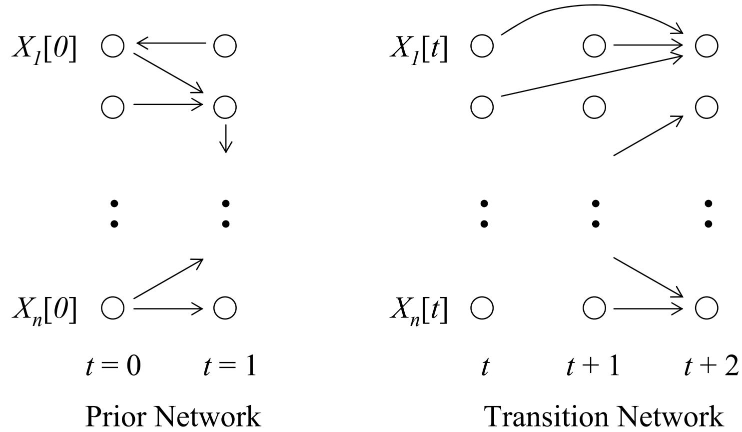


Figure 5.1: A DBN for the variables $X_1, ..., X_n$. Each node X_i represents one DOF in the motion data. We use the prior network to model the first 2 frames. The transition network then models subsequent frames given the previous 2 frames. We assume a second-order Markov property because it is the simplest model that works well.

are conditionally independent of the values before t given the values at t and $t+1$. We found that assuming a first-order Markov property does not work well for our motion data. Hence we assume a second-order Markov property, which is the simplest model that works well. We also assume that the transition probabilities are stationary: the probabilities in G_{trans} are independent of t. The DBN defines a joint probability distribution over $\mathbf{X}[0], ..., \mathbf{X}[T]$:

$$P(\mathbf{X}[0], ..., \mathbf{X}[T]) = P_{G_{prior}}(\mathbf{X}[0], \mathbf{X}[1]) \cdot \prod_{t=0}^{T-2} P_{G_{trans}}(\mathbf{X}[t+2] \mid \mathbf{X}[t], \mathbf{X}[t+1]) \tag{5.2}$$

We apply a non-parametric approach to predict $\mathbf{X}[t+2]$ given $\mathbf{X}[t]$ and $\mathbf{X}[t+1]$. Hence we do not have parameters and we only learn the dependency structure from the data. The data itself represents the "function" defined in a non-parametric approach. Note that our non-parametric regression method for the transition network slightly differs from that of the prior network. This improves the robustness of our approach: no post-process smoothing operation is needed.

5.3 Structure Learning

We take as input a small number (usually four) of motion clips of a particular type of motion. The motion need not be cyclic. These motion clips must be "similar" to each other and each motion clip's starting pose also needs to be similar, as we are trying to model the variation between the

clips. Their lengths can be different, and no timewarping or synchronization of these input clips is needed.

Let n_{seq} be the number of input motion sequences, where the lth motion sequence has length n_l. For each sequence, the data in the first two frames ($\mathbf{X}[0]$ and $\mathbf{X}[1]$) are used to train the prior network. If n_{seq} is large enough, we can use the first two frames from each sequence. Otherwise, we can also take more pairs of frames near the beginning of each sequence. For example, we can take the first ten pairs of frames ($\mathbf{X}[0]$ and $\mathbf{X}[1]$, $\mathbf{X}[1]$ and $\mathbf{X}[2]$, ..., $\mathbf{X}[9]$ and $\mathbf{X}[10]$) as the training data for the prior network. Let n_{prior} be the total number of such instances or pairs of frames. For the transition network, we use the previous two frames to predict each frame. Hence there are a total of $n_{trans} = \sum_l (n_l - 2)$ instances of training data for the transition network. The structure for the prior and transition networks are learned separately given this data.

Given the input data, we wish to learn the best structure that matches the data. This means that we would like to find the best graph or set of edges in the DBN that best matches the data. The set of nodes are already defined as in Figure 5.1. We would therefore like to find the best G that matches the data D: $P(G|D) \propto P(D|G) \cdot P(G)$. This formulation leads to a scoring function that allows us to compute a score for any graph. We then use a greedy search approach to find a graph with a high score. The DBN literature provides many approaches to compute this score. One possibility is the Bayesian Information Criterion (BIC) score: there is one term in this score corresponding to $P(D|G)$ and one penalty term corresponding to $P(G)$. We use a similar score except we do not have a penalty term. Instead we perform cross validation across the data by splitting the data into training and test sets, a common strategy in existing DBN approaches [20]. Doing cross validation allows us to measure how well a given graph matches the data without overtraining the graph on the data and without using a penalty term. Section 5.3.1 describes the greedy search for a graph, and the scoring functions for the prior and transition networks in more detail. To compute our score, we have to compute the conditional probability distribution for each node: $P(X_i \mid \mathbf{Pa}(X_i))$. We use a non-parametric regression approach to compute this probability. Section 5.3.2 provides justification for this approach, and more details about the method.

5.3.1 Structure Search

We learn the structure by defining a scoring function for any graph, and then searching for a graph with a high score. This is done separately for the prior and transition networks of the DBN. The search part of our algorithm is the same as existing DBN techniques; the scoring function however is different because of the non-parametric regression. The problem of finding the graph with the highest score is in general an NP-Complete problem due to the large number

of nodes in the graph. Hence we use a greedy search approach.

Prior Network. The prior network is a BN. We derive the general scoring function by using a maximum likelihood approach: our goal is to find the graph that maximizes $P(D|G)$. Remember that we do not use a $P(G)$ term as we use cross validation and split the data into training and test sets. The score for the prior network G_{prior} is

$$\begin{aligned} & log\ P(D|G_{prior}) \\ = & \ log \prod_{j=1}^{n_{prior}} P(X^{(j)}|G_{prior}) \\ = & \sum_{j=1}^{n_{prior}} log\ P(X^{(j)}|G_{prior}) \\ = & \sum_{j=1}^{n_{prior}} \sum_{i=1}^{2n} log\ P(X_i^{(j)}|\mathbf{Pa}(X_i)^{(j)}) \end{aligned} \tag{5.3}$$

where $X^{(j)}$ represents the jth instance of the prior network training data, and $X_i^{(j)}$ is the value at node X_i of the jth instance of data. We sum over each instance of data for doing leave-one-out cross validation: each jth instance is one example of testing data and the corresponding training data (used in the non-parametric regression) does not include that instance. So the training data for the jth instance is the set of all n_{prior} instances of the prior network training data except the jth instance. Note that we do not model the time component in the prior network even though they represent the first and second frames of the motion. Hence there are $2n$ total nodes. The last equality is due to the conditional independence of the nodes given their parent nodes.

Algorithm 5 shows the pseudocode of the structure search for the prior network. The INITIALIZE section begins with a prior network with any initial set of edges. $X_i^{(all)}$ is the set of all instances for node X_i. For the cross validation, the function $split_samples()$ splits the instances into S equal parts. Each part will be considered as a testing set $X_i^{(tests_s)}$, and the remaining instances form the training set $X_i^{(trains_s)}$ in each of the S cases. In the conditional probability computation, the training sets are used implicitly in the non-parametric regression. n_{tests_s} is the number of instances in the set $X_i^{(tests_s)}$. $curr_score$ stores the current contribution of each node to the total score.

The SEARCH section (Algorithm 5) finds a good prior network based on the score. The $generate_edge_updates()$ function takes the current prior network G_{prior} and computes a set of prior networks $G_{prior}^{(test)}$ by making small changes to the edges in G_{prior}. There are three possible edge updates: (i) an edge addition adds a directed edge between two nodes that were not originally connected, (ii) an edge deletion deletes an existing edge, and (iii) an edge reversal reverses

Algorithm 5: STRUCTURE SEARCH FOR PRIOR NETWORK

INITIALIZE:

$init\ G_{prior}$
for $i = 1$ *to* $2n$ **do**
  $curr_score[X_i] = 0$
end
for $i = 1$ *to* $2n$ **do**
  for $s = 1$ *to* S **do**
    $[X_i^{(train_s)}, X_i^{(test_s)}] \leftarrow split_samples(X_i^{(all)})$
    $[\mathbf{Pa}(X_i)^{(train_s)}, \mathbf{Pa}(X_i)^{(test_s)}] \leftarrow split_samples(\mathbf{Pa}(X_i)^{(all)})$
    $curr_score[X_i] += \sum_{j=1}^{n_{tests}} log\ P(X_i^{(test_s)} \mid \mathbf{Pa}(X_i)^{(test_s)})$
  end
end

SEARCH:

$overall_score_improves = \text{TRUE}$
while ($overall_score_improves$) **do**
  $G_{prior}^{(test)} \leftarrow generate_edge_updates(G_{prior})$
  $G_{prior} \leftarrow take_best_score(G_{prior}^{(test)})$
end

the direction of an existing edge. Note that these are all subject to the BN constraint: so we cannot apply an edge update that creates cycles in the graph. The $take_best_score()$ function recomputes the score for each prior network in $G_{prior}^{(test)}$. Since the total score can be separated into sums of terms for each node X_i, we keep track of each node's contribution to the total score. Each edge update in the greedy search can affect only one or two nodes, so we will not have to recompute the total score every time we update an edge. We then apply the edge update that gives the best improvement towards the overall score, and we continue this process until there is no improvement in the overall score. As this greedy method depends on the initial set of edges, we can repeat the algorithm multiple times by initializing with a different set of edges every time. We then take the set of edges with the highest score among the multiple runs.

Transition Network. For the transition network, we use a similar algorithm to learn the structure. The difference here is that we do not allow any incoming edges to the nodes at time t and $t+1$. The nodes at time t and $t+1$ are assumed to be observed and are used to predict those at time $t+2$. We initialize the graph with the edges from $X_i[t]$ to $X_i[t+2]$ $(\forall i)$, and the edges from $X_i[t+1]$ to $X_i[t+2]$ $(\forall i)$. From our experience with the data, the search almost always selects these edges. Hence we always keep these edges throughout the search to make the process more efficient. The scoring function is similar to the one for the prior network. The score for the transition network G_{trans} is also derived from the $P(D|G)$ term:

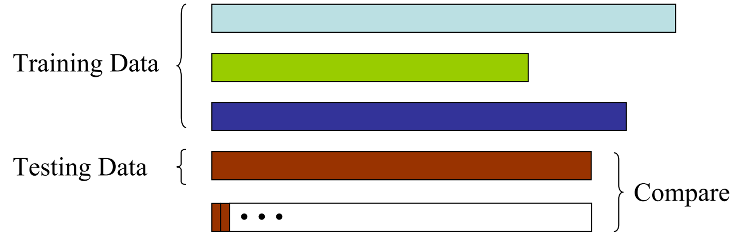


Figure 5.2: When learning the structure for the transition network, we do a cross validation over each motion sequence. We take each sequence as testing data, and use the others as training data. For the testing sequence, we take the first two frames as input and re-synthesize the whole sequence with the given structure. The newly synthesized sequence is then compared to the original data to evaluate the structure. This is what we compute intuitively in the scoring function for the transition network of the DBN.

$$\sum_{l=1}^{n_{seq}} \sum_{j=2}^{n_l-1} \sum_{i=1}^{n} log\ P(X_i[j]^{(l)} | \widehat{\mathbf{Pa}}(X_i[j])^{(l)}) \tag{5.4}$$

where $X_i[j]^{(l)}$ is the value at node $X_i[j]$ of the lth motion sequence of the transition network training data. This score is different from the BN score in that we start with the first two frames in each sequence, and compute the subsequent frames in the sequence by *propagating* the computed frames. So the second frame and the newly synthesized third frame are used to compute the fourth frame, the newly synthesized third and fourth frames are used to computed the fifth frame, and so on. The $\widehat{\mathbf{Pa}}$ notation represents this propagation of frames. The justification for this propagation instead of treating each instance separately is that the learned structure would otherwise not give a good result: the predicted trajectories deviated from the actual ones when we attempted to treat each instance separately. Intuitively, since we propagate the values when we synthesize a new motion given the first two frames, we should do this propagation when we learn the structure. We are effectively trying to compute how good a given structure is by trying to re-synthesize each input motion sequence given the first two frames, and comparing the synthesized sequence with the original data (Figure 5.2). Note that we sum over the n nodes in time $t+2$ as these are the ones we are trying to compute in the transition network.

Algorithm 6 shows the pseudocode of the structure search for the transition network; it is similar to the one for the prior network. The $generate_edge_updates()$ function makes small changes to G_{trans} by using the same edge update rules (as the prior network), and creates a set of transition networks $G_{trans}^{(test)}$. We perform cross validation for each graph in this set. For the cross validation, each lth motion sequence forms the testing data ($test_l$) and the corresponding training data ($train_l$) includes all the motion sequences except for the lth sequence. The $compute_score()$

Algorithm 6: STRUCTURE SEARCH FOR TRANSITION NETWORK

```
overall_score_improves = TRUE
while ( overall_score_improves ) do
    G_trans^(test) ← generate_edge_updates(G_trans)
    foreach G ∈ G_trans^(test) do
        for l = 1 to n_seq do
            compute_score(test_l, train_l)
        end
    end
    G_trans ← take_best_score(G_trans^(test))
end
```

function gives us the score based on the equation given above for the transition network, except that this function computes the score for each lth part of the cross validation and adds each score to the total score. Note that $train_l$ is used implicitly in the non-parametric regression for computing the conditional probability in the score. The $take_best_score()$ function selects the G_{trans} that gives the best improvement towards the overall score; it sets the $overall_score_improves$ variable if necessary. The whole process is repeated until the overall score does not improve.

5.3.2 Non-Parametric Regression for Computing Conditional Distribution

The scoring functions for the prior and transition networks require the computation of the conditional probability $P(X_i|\mathbf{Pa}(X_i))$. We briefly describe the parametric approaches that we attempted to use. As these approaches did not work well, we decided to use a non-parametric regression method.

Many BNs and DBNs that treat X_i as a continuous variable use a linear regression model [71] to describe the relationship between X_i and its parents. We attempted to model the relationship between X_i and its parent nodes as a linear relationship, but we found that it is not appropriate for our motion data. We then attempted to model this relationship by nonlinear regression. We tried to find the parameters of a nonlinear function that takes the parents of X_i as input and X_i as output, where the nonlinear function is a sum of multivariate radial basis functions. While this worked well for the prior network of the DBN, it performed poorly for the transition network. This might be because there is not enough data to accurately estimate the parameters of a nonlinear function. Hence we decided to try a non-parametric method. We found that a kernel regression approach worked well for our data.

Prior Network. We assume that $P(X_i|\mathbf{Pa}(X_i))$ is a guassian distribution, and use kernel regression to find the mean and standard deviation of this distribution. Recall that we are given

the graph and training data. The graph allows us to find the parent nodes of X_i. The training data allow us to find instances of $(\mathbf{px}_k, x_k)$ corresponding to $(\mathbf{Pa}(X_i), X_i)$. Note that we also have the actual value of $\mathbf{Pa}(X_i)$, which we call $\mathbf{pa}(X_i)$. Since a large number of the instances $\mathbf{px}_k$ are far away from $\mathbf{pa}(X_i)$, we pick the k-nearest instances. The notation with the subscript k represents these nearest instances. We measure the distance with a Euclidean-distance metric: $D(\mathbf{px}_k, \mathbf{pa}(X_i))$. We then compute a weight for each instance:

$$w_k = exp\{-D(\mathbf{px}_k, \mathbf{pa}(X_i))^2/K_W^2\} \tag{5.5}$$

where K_W is the kernel width. Next, we compute a weighted mean and variance based on these weights:

$$\begin{aligned} \mu(X_i) &= \frac{\sum_k w_k x_k}{\sum_k w_k} \\ var(X_i) &= \frac{n_k}{n_k - 1} \cdot \frac{\sum_k w_k (x_k - \mu(X_i))^2}{\sum_k w_k} \end{aligned} \tag{5.6}$$

where n_k is the number of non-zero weights w_k, and the standard deviation $\sigma(X_i)$ is the square root of the above variance. With the mean and standard deviation of X_i, we can now compute $P(X_i|\mathbf{Pa}(X_i))$, and thereby the scores for the structure search. For the prior network, we have cases where X_i has no parents. To compute $P(X_i)$, we find instances of x_k corresponding to X_i. The mean and standard deviation of X_i is then the mean and standard deviation of the instances x_k.

Transition Network. We compute one distribution for each node i at time $t+2$ ($X_i[t+2]$). The regression for the transition network is essentially the same as above with two important modifications. The first modification is that we also have a weighted velocity term when computing the distance function $D(\mathbf{px}_k, \mathbf{pa}(X_i[t+2]))$. This velocity term is $(X_i[t+1] - X_i[t])$ (recall that $X_i[t+1]$ and $X_i[t]$ are always parent nodes of $X_i[t+2]$). Including this term allows us to find k nearest instances that better match $\mathbf{pa}(X_i[t+2])$. The second modification is related to $X_i[t+2]$, whose values we have to generate in order to compute probabilities and scores (as in Figure 5.2). Instead of dealing with "absolute" $X_i[t+2]$ values, we deal with "delta" $X_i[t+2]$ values. In Equation 5.6, instead of x_k representing the k instances of $x_i[t+2]$ (where lower x means actual value), x_k now represents the k instances of $(x_i[t+2] - x_i[t+1])$. And instead of X_i representing $X_i[t+2]$, X_i now represents $(X_i[t+2] - X_i[t+1])$. To generate an actual value of $x_i[t+2]$, we take $\mu(X_i[t+2] - X_i[t+1])$ and add this to the existing $x_i[t+1]$ value. Intuitively, since the "absolute" values have a much wider range, sampling from this range will require post-process smoothing. The "delta" values have a small range, and sampling from it is more robust and no post-process smoothing is needed.

5.4 Synthesis of New Variants

We can use the learned structure and the input data to synthesize an unlimited number of new spatial and temporal variants. Since the DBN represents a joint probability distribution, we sample from this distribution to synthesize new variants. We represent the μ's and σ's that are computed for each node as a set $(\vec{\mu}, \vec{\sigma})$. If we pick $\vec{\sigma} = \vec{0}$, this gives the mean motion of the inputs. The set $(\vec{\mu}, \vec{\sigma})$ represents variations of motions away from this mean motion. Note that the μ's are not fixed, since the μ's and σ's from previous time frames can affect the μ's in later time frames.

Prior Network. We synthesize the first 2 frames of a new motion with the prior network. We first find the partial ordering of the $2n$ nodes in the prior network. Such an ordering always exists since BNs are acylic. We generate values for each of these nodes according to this ordering. The nodes at the beginning will be the ones without parents. We sample a value from each of the guassian distribution of these nodes. The rest of the nodes will depend on values already generated. We use the procedure in Section 5.3.2 to find the mean and standard deviation for each node, except that we use the learned structure and all the n_{prior} instances every time. We then sample a value from the distribution of each node.

Transition Network. Given the first 2 frames, we synthesize subsequent frames by "unrolling" the DBN (Figure 5.3). We perform one non-parametric regression for each node at each time frame. We use the learned structure and all the n_{trans} instances every time. We use the procedure in Section 5.3.2 to compute actual values of $X_i[t+2]$. The main difference is that after computing $\mu(X_i[t+2]-X_i[t+1])$ and $var(X_i[t+2]-X_i[t+1])$, we sample from this distribution and add the value to the existing $x_i[t+1]$ value to get the $x_i[t+2]$ value. *No post-process smoothing operation* is needed. If the input motions are cyclic, we can synthesize a continuous and unlimited stream of new poses.

5.5 Constraints

The synthesized poses from the previous section might need to be cleaned up for handling foot and hand constraints. This fixes footskate problems and also deals with cases where the foot/hand has to be at a specific position. We develop an inverse kinematics framework that fits with our DBN approach. Intuitively we have to satisfy three constraints: (i) the foot/hand needs to be at specific positions at certain times, (ii) the solution should be close to the mean values (at each node and time) predicted by the DBN, and (iii) the solution should maintain smoothness with

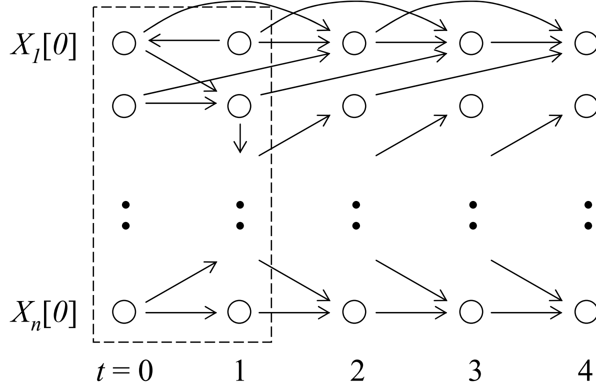


Figure 5.3: We "unroll" the DBN from Figure 5.1 to synthesize new variants. We show here the unrolled network for 5 time frames. Note that the first two frames come from the prior network of the DBN and may not contain cycles. Since the DBN represents a joint probability distribution over the possible trajectories of each DOF, we sample from this distribution to generate new variants. It is important to recognize that the synthesized motion does not have a one-to-one correspondence with any one of the input motions. This means that the synthesized motion is not just a copy of one of the input motions plus some slight differences, but the timing of the whole motion itself is different. Furthermore, no new pose is exactly the same as any previous pose.

respect to the previous frames. The first constraint is a hard inverse kinematics constraint while the last two are soft constraints. This naturally leads to an optimization solution:

$$\begin{aligned} \min_{\mathbf{q}_t} \ & \{w_1\|\mathbf{q}_t - \overline{\mathbf{q}}_t\|^2 + w_2\|\mathbf{q}_t - 2\mathbf{q}_{t-1} + \mathbf{q}_{t-2}\|^2\} \\ \text{s.t.} \ & \|\mathrm{f}(\mathbf{q}_t) - \mathrm{pos}\|^2 = 0 \end{aligned} \tag{5.7}$$

where $\mathbf{q}_t$ is the set of DOFs for one foot or hand at time t. There are 6 joint angles for each foot, and 7 for each hand. $\overline{\mathbf{q}}_t$ is the set of mean values (of the corresponding nodes and time) predicted by the DBN, $\mathbf{q}_{t-1}$ and $\mathbf{q}_{t-2}$ are the DOFs from the previous two frames, f() is the forward kinematics function that gives the end-effector 3D position corresponding to $\mathbf{q}_t$, and "pos" is the 3D position that we want the foot/hand to be at. We run an optimization for each foot/hand and time frame separately. If there is a large amount of motion, these 3D positions and frames can be found with automated methods [46]. But we find that it is not difficult to identify these manually for our motions. We initialize the optimization with the solution we sampled from the DBN. Since the solution we get from Section 5.4 is already close to what we want, the optimization only makes minor adjustments and is therefore efficient. The optimization uses a sequential quadratic programming method. We set w_1 to 1 and w_2 to 5.

5.6 Evaluation

A main result of our work is that we can synthesize spatial and temporal variants of the input examples. *Spatial variation* means that no new pose is exactly the same as any of the input poses or previously synthesized poses. Spatial differences can usually be better seen in images of poses. *Temporal variation* means that a new variant motion has a different timing than any of the input motions or previously synthesized variant motions. It is important to recognize that a new variant does not have a one-to-one correspondence with any of the input motions. This means that the new variant is not just a copy of one of the input motions plus some slight differences (as is the case in previous work), but the timing of the whole motion itself is different. Temporal differences can usually be better seen in animations. The new variants are therefore statistically and visually similar to the inputs, but are *not exact copies* of them.

In general, we expect our approach to work on time-series data with DOFs that are correlated. This means that some DOFs are correlated with others, but it is not necessary that all DOFs are related to each other. The DBN model, by design, works on these types of data. Experimentally, we show that our approach works for many types of human motion data.

We assume that our data satisfies a 2nd-order Markov property in our DBN model. There are two questions that arise from this assumption. The first question is why we used a 2nd-order model instead of 1st, 3rd or higher orders. We tried our algorithm by assuming a 1st-order property. While we can learn a structure and generate the first frame of motion from the prior network, the subsequent frames that are generated by the transition network do not make sense at all. After a few frames, the new poses will diverge away from the poses in the input motions. Intuitively, the algorithm is unable to find nearest instances that are truely "near" the existing previous frame, and hence it cannot generate the corresponding next frame accurately. However, we find that using two previous frames works well in finding the k nearest instances, and this is the reason that a 2nd-order model works well. We did not try 3rd or higher order models, since we already have a simpler (2nd-order) model that works well. We believe that higher order models will produce similar results while taking a longer runtime. The second question is whether considering the previous 2 frames is enough. Although there are only 2 frames, for our human motion with 62 DOFs, there are actually 124 pieces of information. This information is enough for the algorithm to find the nearest "patches" (or nearest instances) of input data, in order to perform the non-parametric regression to generate a subsequent frame.

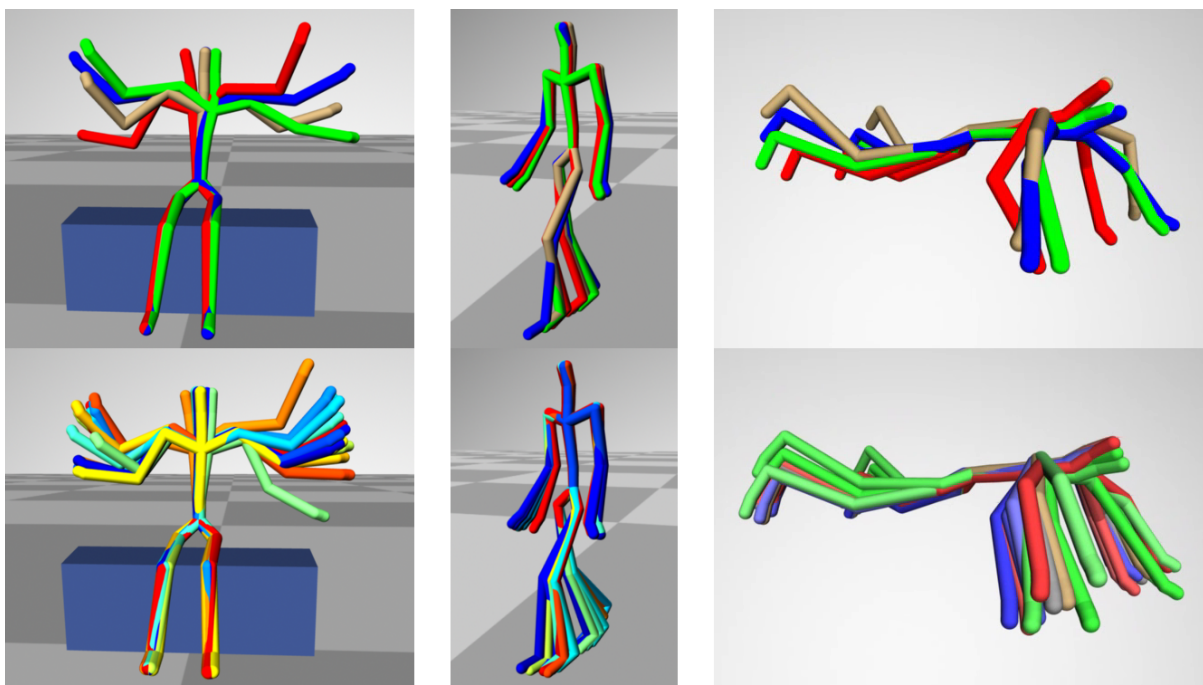


Figure 5.4: Results for cheering, walk cycle, and swimming motion. In each column, the top image shows the 4 inputs (overlapped, each with different color) and the bottom image shows the 15 outputs (overlapped, each with different color). These are frames from the animations.

5.6.1 Results for Full-body Human Animation

We show results for four types of human motion data: cheering, walk cycle, swimming breast stroke, and jumping. We use respectively 433, 322, 384, and 309 frames of data (at 60 frames per second) as input. These are the total number of frames for each motion type. We have four input motion clips in each case.

We find that four input motions is the smallest number that learns a DBN structure that gives good results. We can learn a structure with a smaller number of inputs, but it does not synthesize reasonable motion at all. A larger number of inputs also works well, but we show the robustness of our method by showing that it works with only a few inputs. We use values from 15 to 60 to find the k nearest instances. In the learned DBN structure, each node has between 2 and 15 parent nodes (except for the nodes in the prior network that have no parents). After learning a structure, we can synthesize variants of the four inputs. The results for cheering, walk cycle, and swimming breast stroke motions (Figure 5.4) show variants generated with the four inputs in each case. Given the learned structure and just one input motion clip, we can also use the same approach to synthesize variants of that single input. The results for jumping motions (Figure 5.5) show variants generated with just one input. In addition, if the motion is cyclic, we can

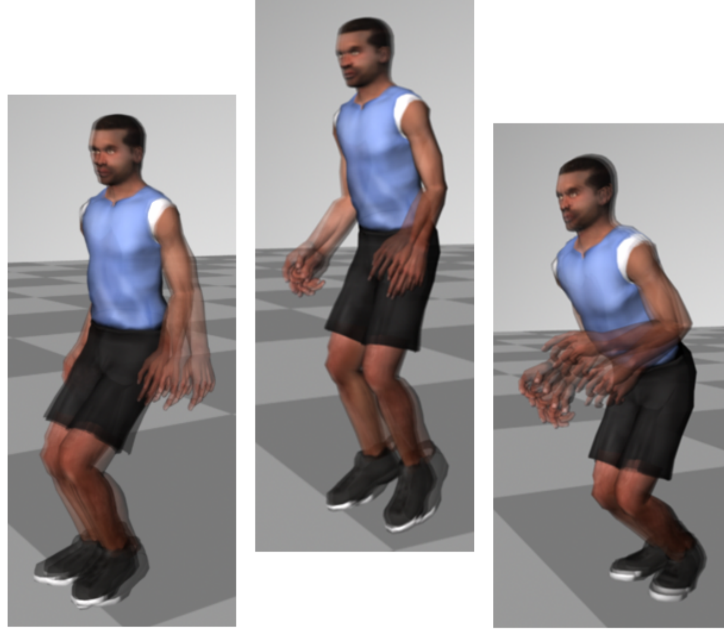


Figure 5.5: Given the learned structure and just one jumping motion as inputs, we synthesize four new variant motions. We overlap poses from these four new motions at similar time phases of the jump. We can see the variations in the poses at these time phases. The poses for the head vary the least because the head poses also vary the least in the input data.

synthesize a continuous stream of new cycles. We have examples of animations for walk cycles and swimming motion.

Figure 5.6 show graphs of the input and output cheering motions. While these graphs are for cheering motion, they are typical of similar graphs of other motion types. Note that the new output variants follow the general trajectories of the inputs, but are not exactly the same. In the middle column of the figure, we can see some of the joint correlations. For example, knowing the value of the right shoulder can help us predict the value of the left shoulder. These joint relationships are learned automatically. For the right column of the figure, we performed PCA of the input and output data, and plotted the results from the first few PCA dimensions. The PCA reduces the 62-DOF data to 11 dimensions, keeping more than 99% of the energy.

5.6.2 Memory and Performance Time

Memory is needed for storing the learned DBN structure and the four input motions. The DBN structure consists of a set of sparse directed edges in G_{prior} and G_{trans}, and the means and standard deviations of the nodes in G_{prior} that have no parents. The memory for the DBN structure is small, and hence the total memory is essentially the four input motions. It takes between half

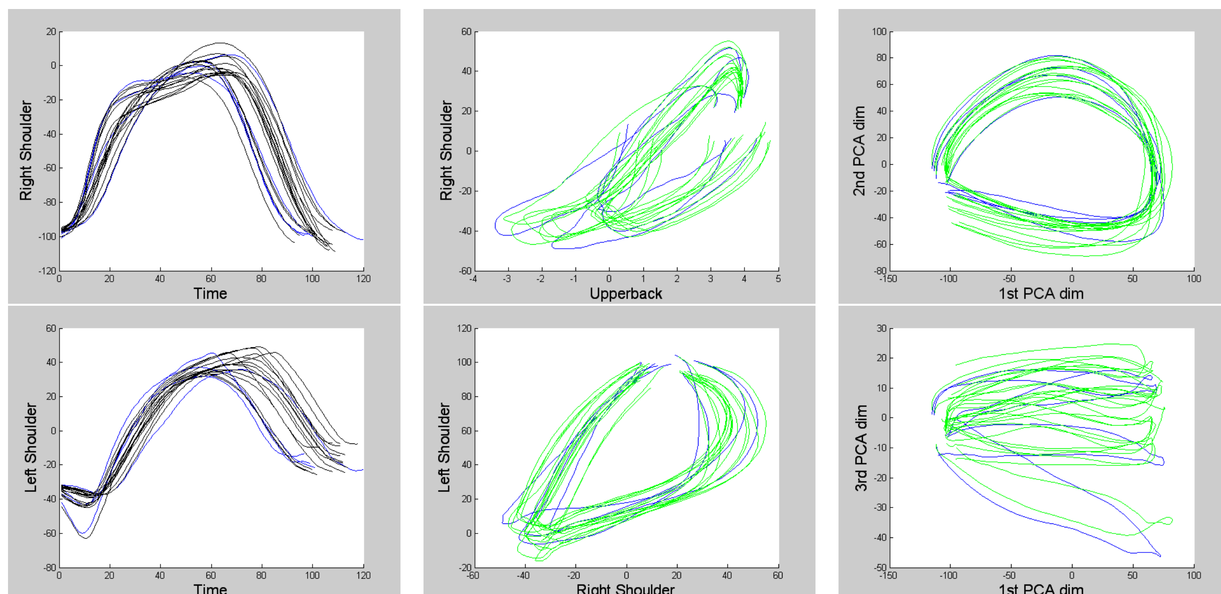


Figure 5.6: Plots of four inputs (in blue) and fifteen output variants (in black or green) for cheering motion. Each curve represents one motion clip. Note that these motions are not cyclic. Left Column: Two selected plots of DOF vs. time. Middle Column: Two selected plots of DOF vs. DOF. Right Column: Two selected plots of PCA-dimension vs. PCA-dimension.

an hour and two hours to learn the DBN structure for each type of human motion. This learning process can be done offline. The runtime process of synthesizing new human motion takes about 200 ms. to generate 1 second of motion.

5.6.3 User Study

We performed two experiments in the user study. For Experiment A, we compare "Our Variants" with motion capture data. "Our Variants" are motion clips generated by our approach. The purpose is to decide which is more natural. We ran this experiment for cheering motion and walk cycles separately. Each user watches a random mixture of 15 of these motion clips. After watching each motion, we ask the user to provide a score from 1 to 9 (inclusive) of how natural or human-like that motion is. A higher score corresponds to more naturalness. We tested 15 users, and we have a total of 225 scores. We performed ANOVA on these scores. For cheering motion, p is 0.930 and this suggests that the means from the two samples (of "Our Variants" and motion capture data) are not significantly different. For walk cycles, p is 0.578 and this again suggests that the means from the two samples are not significantly different. Therefore, for both cheering motion and walk cycles, motion synthesized by our approach was not found to be significantly less natural than motion capture data.

For Experiment B, we compare "Our Variants" with "Cycle Animation". "Our Variants" are long sequences where each sequence consists of at least 15 motion clips generated by our approach. These motion clips are all slightly different. A "Cycle Animation" is a long sequence consisting of at least 15 motion clips: each of these is randomly selected from the 4 input motions. The name of "Cycle Animation" is inspired by common techniques used in films to generate the motions for many characters using a small number of motion clips. Since this is a general term, we cannot say that our sequences are similar to the ones used in films, as there are many other parameters that can be considered in an actual film setting. The purpose is to decide which is more repetitive. Specifically, a long sequence is repetitive if many of the motion clips within the sequence are exactly repeated. We ran this experiment for cheering motion and walk cycles separately. Each user watches a random mixture of 15 of these long sequences. After watching each sequence, we ask the user to provide a score from 1 to 9 (inclusive) of how repetitive that sequence is. A higher score corresponds to more repetition. We tested 15 users, and we have a total of 225 scores. We performed ANOVA on these scores. For cheering motion, p is 1.02e-8 and this suggests that the means from the two samples (of "Our Variants" and "Cycle Animation") are significantly different. For walk cycles, p is 4.49e-8 and this again suggests that the means from the two samples are significantly different. Therefore, for both cheering motion and walk cycles, "Our Variants" are less repetitive than "Cycle Animation". In Experiment B, note that each long sequence has at least 15 motion clips. It takes some time to recognize whether or not there are clips that are exactly repeated. Hence Experiment B does not apply to relatively short animations, since "motion clones" are difficult to detect in short animations (as shown in [67]).

5.6.4 Experiments with Adding Noise

A simple possible approach to generate variation is to add noise to existing motion. We experimented with two such methods on the walk cycle data. The first is a naive or strawman method. We time-warp the four input walk cycles, compute simple statistics of the time-warped data for each DOF separately, and use this information to add smoothed noise to one of the four input cycles. We check that the noise-added motion is changed by a similar amount compared to the variants that our DBN approach generates. We do so by taking pairs from our fifteen variants and the four inputs (each pair has one variant and one input), computing the normalized sum of squared differences of joints between each pair, and modeling these sums as a normal distribution. We also compute the normalized sum of squared differences of joints between the noise-added motion and its corresponding input, and check that this sum is within one standard deviation of the mean of the normal distribution above. We tried an example where we added

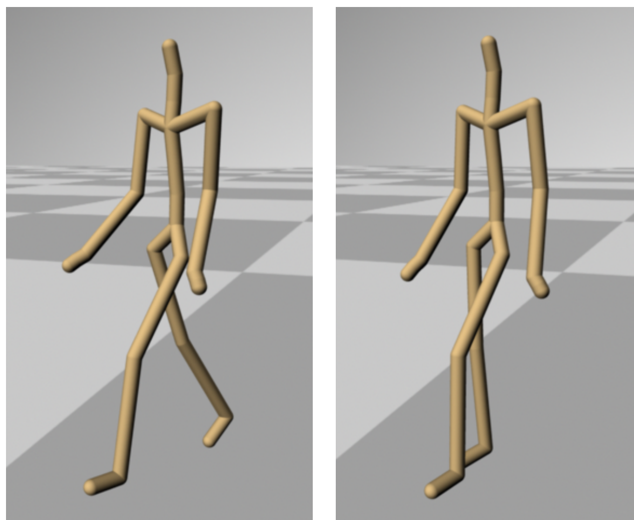


Figure 5.7: Left: Example frame from walk cycle motion clip with strawman noise added to left shoulder/arm. The left shoulder turns in a way that does not synchronizes with the right arm. Right: Example frame from walk cycle motion clip with Perlin noise added to right hip/knee. The right hip/knee pause and move in a way that do not synchronize with the rest of the walk cycle. In both cases, the unnaturalness of the timing of the whole walk cycle can be better seen in the animations.

noise only to the left shoulder and elbow (Figure 5.7 left). The resulting walking motion shows that the left shoulder/arm motion is unnatural, and does not fit with the rest of the walking motion. In contrast, our DBN approach will learn that the left shoulder is correlated with other joints, and handle these issues autonomously. We tried another example where we added noise to all joints. While the overall walk motion still exists, it is obvious that the poses and timing of the motion are awkward. Furthermore, adding noise requires a smoothing process that can take away details from the original motion.

The second method is to add band-limited noise to one of the four input cycles with the Perlin noise function [75]. We also perform the same noise-addition check as in the first method. Figure 5.7 (right) shows an example where we added Perlin noise to the right hip/knee joints of a walk cycle from motion data. The right hip/knee pause in such a way that they do not synchronize with the rest of the motion. We also tried examples where we added noise to several joints. In general, we found that a trial-and-error process of manual parameter tuning is needed. Most importantly, a human understanding of the motion (ie. if the left arm swings higher, the right arm is more likely to swing higher) is required to add noise the right way. Otherwise, the motion can become spatially or temporally awkward.

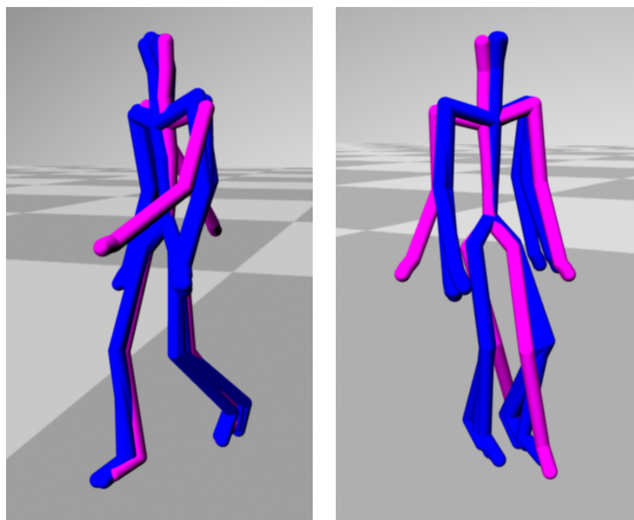
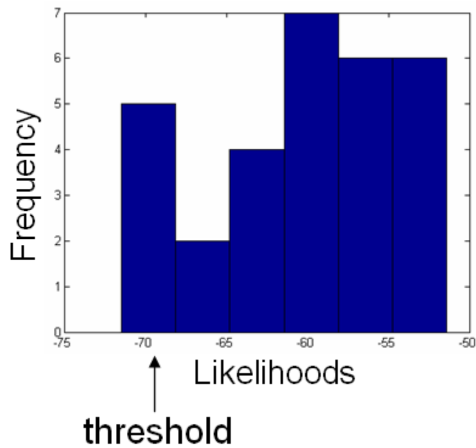


Figure 5.8: Two examples of motion sets that do not work with our approach. There are five overlapped walk cycles in each case. Four of them are inputs (in blue) that are similar, and the other one (in magenta) does not fit together with these four. Left: For the one that does not fit, the arms swing higher than the other four. Right: For the one that does not fit, the motion turns slightly to the right.

5.7 Inputs that work well with Our Approach

One limitation of our approach is that the input motion examples have to be “similar but slightly different”. They have to be “similar” because we are learning a model for that particular type of motion. They have to be “slightly different” because the small differences among the inputs are where we get the variation from. In the results section, we have shown examples that work with our approach. Here, we describe and show examples of inputs that do not work with our approach. For our walk cycle data, we have four input motions where the character walks two steps forward. If there is a walk cycle where the character swings the arms much higher (Figure 5.8 left), it will not fit with the original four motions and will not work as another input motion. We are still able to use the five motions to learn a DBN structure, but the synthesis step may not produce a reasonable output motion. For this and similar cases, we recommend using the method below to first eliminate the ones (ie. higher arm swing one in this case) that do not fit with the rest of the motions. However, if we have four or more of such higher-arm-swinging walk cycles, they can be used together in our approach effectively. Another example is a walk cycle where the character turns slightly to one side while walking forward (Figure 5.8 right). This will also not fit with the original four inputs.

It is difficult to precisely define what is meant by “similar” motions. Instead of making such a definition, we introduce a method (that we can precisely describe) to characterize the types of



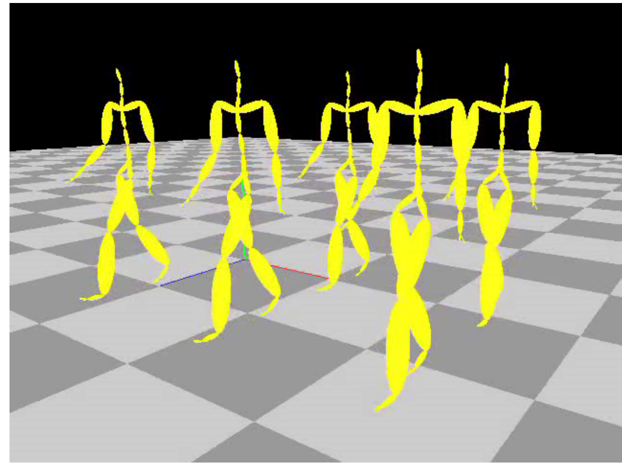


Figure 5.9: Left: Plot of frequency versus likelihoods for the training data. Right: We started with a testing set of eight walk cycles, and our method selected these five to be similar to the ones in the training set.

inputs that work well with our approach. Our method is data-driven and based on DBNs. The overall idea is to take a given set of training data that we already know works well with our approach. We then take a new set of testing data, and eliminate motion clips from this set one by one until we are left with a set that can also work well with our approach.

We start with a given set of training data that we know works well. This data can either be selected manually (ie. we tested them on our approach) or can come from the results of this method. As an example, we started with eight walk cycles that we have already selected. We split these eight into groups of six and two. We learn a DBN with the group of six and compute likelihoods with the learned DBN for each of the other two. We use the likelihoods that are described in Section 5.3. We repeat this process for different combinations of six and two. The idea is to get a number of likelihoods that we can use to characterize the training data. Figure 5.9 (left) shows a plot of the likelihoods that we got from this procedure. The training data is only for walk cycles. We tried to include a training set of cheering motions. However, we found that because the legs and feet of the character do not move during the cheering motion, those joint values can be well predicted by default. We therefore decided that the training data should only contain one type of motion (ie. walk cycles).

With the set of likelihoods from the training data, we can set a threshold (Figure 5.9 left) for deciding the likelihoods that we should accept in the new testing data. This is a parameter that we choose by ourselves. We set the threshold to be the tenth percentile of all the likelihoods. We can now take a new testing set of motion clips. We used a new test set of eight walk cycles in our example. We again separate this set into different groups of six and two, so that we can compute

a likelihood for each motion clip. We then eliminate the motion clip with the lowest likelihood if it is lower than the threshold. We now have seven motion clips and we repeat the process to computing likelihoods for each of the seven clips. We stop this process until the lowest likelihood is above the threshold. In our example, this process stops with five walk cycles (Figure 5.9 right). The clips that got eliminated have different speeds, heights of arm swings, and angles of turning compared to the remaining five. If all the clips in the test set are different from the training data, the method will eliminate all of them.

5.8 Discussion

We have presented a method for modeling and synthesizing variation in motion data. We use a Dynamic Bayesian Network to model the input data. This allows us to build a multivariate probability distribution of the data, which we sample from to generate new motion. Given input data of a type of motion, our model can be used to generate new spatial and temporal variants of that motion. We show that our approach works with a variety of full-body human motion. For applications such as crowd animation, our method has the advantage of being able to take small, pre-defined example cycles of motion, and generate many variations of these cycles.

Our **contribution** and **additional value over previous work** is in the study of the problem of generating variation in motion data, which is still relatively unexplored. Instead of considering variation as an additive noise component, we take a data-driven approach and apply learning techniques to this problem. We introduce a novel method to model and synthesize variation for many types of motion data. Our model takes a small number of input motions, and synthesizes spatial and temporal variants that are statistically similar to the inputs.

One important **limitation** is that the input examples must come from a particular type of motion (ie. walk cycles, swimming). Our current approach cannot combine different motion types. The inputs also have to be "similar", but we specifically describe how we can get inputs that work well with our approach.

We have shown that it is possible to automatically generate spatial and temporal motion variants from a small amount of data. One interesting **insight** is that our non-parametric technique is similar to texture synthesis methods. Non-parametric texture synthesis methods search for similar patches of the previous pixels in order to generate the next pixel. Our method also searches for similar patches of previous frames in order to generate the next frame of motion.

In general, we believe that the overall problem of generating motion variation is still relatively unexplored and there are many open issues. Our work is just the beginning and we think of it as one step towards solving the overall problem.

Chapter 6

Putting it All Together

We started with the realistic generation of navigation motion for many characters as our goal. In Chapter 3, we described a method to plan for sequences of motion for virtual characters to reach user-specified goal locations. We then explored a technique to speed up the motion synthesis process by using the idea of precomputation in Chapter 4. And in Chapter 5, we have a method for generating motion variations to a small number of existing motion clips. We now discuss ways to combine these approaches.

We first discuss how to incorporate our variation method into the behavior planning framework. We begin with four motion clips of captured data for each behavior. For example, we have four similar but slightly different motion clips in the "forward jogging" node. There are two choices we can take here: (i) we can either learn a DBN structure and generate a number (ie. ten to a hundred) of clips for each behavior in advance, and store them for use in real-time; or (ii) we can learn a structure in advance, and then generate the variants in real-time. We suggest the first choice since it does not require a lot of memory to store the generated variants. We then have the same set of behaviors or nodes as before, except with more motion clips per node. The number of clips required depends on the number of characters, camera viewpoint, size of environment, and (most importantly) user perception. It would be a good direction of future work to explore the number of different clips needed for the user to perceive enough motion variation. Given the extended set of motion clips, we can build the tree of our behavior planning framework in the same way as before, except that we instantiate one specific clip from every node that we choose during runtime. Even though the motion clips can be slightly different in the overall position and timing, this will not affect the rest of the algorithm. The planner can run the same way as before.

We can also incorporate our variation method into the precomputed search trees framework. Since the precomputation technique uses the same data structure and motion clips as the ones for behavior planning, we are using all three techniques together here. As described above, we can

also start with four clips for each behavior or node. For incorporation into the precomputation framework, we particularly suggest generating the new variants for each node in advance. This will maintain the speed of the runtime search, which is an important feature of the whole technique. The multiple clips in each node will again be slightly different in the position and timing. In the tree precomputation step, as we expand a node in the tree, we instantiate one specific motion clip randomly from the ones for that node. This should provide the variety that we can get from the existing clips if we assume that each clip in the tree is equally likely to be used. Another possibility is to timewarp the motion clips to get the same position and time between the first and last pose in each clip of every node. However, this would take away from the variation that our method generates, and hence we do not suggest timewarping them. Once the tree is built, the gridmaps that are built for the tree and the runtime backward search step can execute the same way as before.

In our implementation of the combination of the three methods, we precomputed the tree by doing runtime instantiation of a specific motion clip for each node. In the motion data that we were using for the precomputation method, we usually have only one clip of most types of motion. Hence we took one of each of these motion: forward jogging, jogging and turning left about 30 degrees, jogging and turning left about 45 degrees, jogging and turning right about 30 degrees, and jogging and turning right about 45 degrees. We take this original data set as it is likely to be a more typical data set in practice to begin with than having multiple clips of each motion type. It would be a good test of our method to show that it can work with only these few motion clips. Since the foot motions among these clips are different, we do not change them in the new variants. We change only the arms and upper body motion. We have tested that these upper body motions can be used well to learn a DBN structure. When learning a structure, we also use the foot DOFs as potential independent variables. This allows the upper body motion to synchronize with the lower body. We take the foot motions for the forward jogging, and generate the upper body and arm motions using our method. Each variant has a slightly different positioning of the character, but the same change in overall time because we are re-using the foot motions. This is a tradeoff we have to make given the small amount of original data. We generate twelve new variants of forward jogging this way. We also generate five new variants of each type of jogging and turning motions. As we explained above, the number of variants that we should generate depend on several factors that can be a perception study for future work. We precompute one tree using the original five motion clips, and another tree using the new variants together with the original five clips. We generated the motions for multiple characters using these two trees. Figure 6.1 shows example frames of the resulting animations from the two cases. Near the beginning of the generated motions for the case with five inputs, we can see the repetition of

Figure 6.1: Left: Example frame of resulting animation generated with five original motion clips. Right: Example frame of resulting animation generated with thirty-two variants together with the five original clips.

the forward jogging clip since it happened to use that clip for multiple characters side-by-side. For the case with the new variants, we cannot observe this repetition quite as much. Another main difference between the two cases is that the arm swinging motions are slightly different, although one has to watch the animations for some time before recognizing the differences.

Chapter 7

Discussion and Future Work

We have studied two main problems for generating virtual crowds: (i) how to model human behaviors such that intuitive sequences of motions for a large number of characters can be generated efficiently, and (ii) how to model and synthesize variation in motion data.

Our contributions are:

- A planning approach that applies heuristic search methods to efficiently generate goal-driven navigation motion for virtual human-like characters. Compared to methods that use large data sets of motion, we show that we can use a small set of segmented motion clips to generate motions for a large number of characters navigating simultaneously in dynamic environments. Specifically, we can use about twenty segmented clips to generate the navigation motions for one hundred characters.

- A novel precomputation-based approach to use human motion data to generate navigation motion: we first precompute a search tree of possible motion paths with the data, and then use a backward search method during runtime to solve planning queries. We show that our approach is more than two orders of magnitude faster than traditional forward search methods such as A*-search. Although there are tradeoffs of memory and completeness (of solutions) for the approach, there are many situations where these tradeoffs are worthwhile given the faster runtime speed.

- We present a technique for precomputing large diverse trees, and explore the advantages and disadvantages of our method compared to previous methods for building diverse trees. We have learned that a randomized-based approach for precomputing our trees works well in terms of the trees being able to handle as many environments as possible.

- We study the problem of generating variation in motion data. Instead of considering variation as an additive noise component, we take a data-driven approach and apply learning techniques to this problem. We show that we can use Dynamic Bayesian Networks to synthesize an unlimited number of variants automatically. This process does not require manual parameter tuning and is not tedious compared to the major previous approach of adding noise.

- We show that we can use our method to model and synthesize variation for many types of human motion data. Our model takes a small number of input motions, and synthesizes spatial and temporal variants that retain original features of the inputs but are not exact copies of them. Our approach is novel in that there is no previous automated method that can generate such variants for human motion data.

We think of our *Behavior Planning* framework as one method among a spectrum of methods. The planning action space is carefully chosen such that animations can efficiently and easily be generated. In general, we believe that our framework can also be applied to other areas such as robotics. The underlying planning algorithm can be used for robots if a set of well-defined actions are given as input. For example, if a set of actions are defined for manipulation or arm reaching tasks, we can use the same framework to generate sequences of such motions. This can be done for both virtual characters and robots.

We believe that our *Precomputed Search Trees* approach can be used to generate motions for multiple characters. Currently, our system builds a tree for the motions of one character. It should be possible to extend this for two characters or more. For example, the branches of the tree can alternate between the motions of two characters, and executing the motions along a path in the tree can correspond to the motions for both characters at the same time.

For both our planning frameworks, we can further test their capabilities by using them to generate as many characters' motion as possible. Is there a limit to the number of characters that can be generated? What is the bottleneck, and can we do anything to deal with the bottleneck to extend the existing systems?

For our variation technique, we believe that our work is just the beginning and there are many open issues within the overall problem (some of which we describe below). There has been some recent work on this problem and we expect to see more on this topic in the future.

One interesting area for future work is to provide a method for the user to control the variation that is generated. A simple way to "control" the output motion is simply by taking different input data to begin with. If the motion is jumping and we have input data that has large variations in the swinging of the arms, then the synthesized motions will also have large variations in the arm

swing. If the input data has more variation in the head movement, the synthesized motions will have more variation in the head. Another method that we have tried is to use parameters such as the kernel width to tune the variation that can be generated. While this provides the ability to generate variants that are closer to or further away from the mean motion, it can be difficult to guarantee that the motion will look natural. One possible challenge is therefore to enable the user to more intuitively control the variation in a motion while automatically constraining the output to lie within the "natural" range of movement.

We can further analyze the range of variation that can be created. We typically use four input motions because it is the smallest number that works well, and it is common to only have a small number of motions available. We have tried to use up to ten input motions, and we can create more variants with ten inputs as long as they span a wider range of space. For future work, we can analyze more formally the range of variation we can get given a set of inputs. Perhaps we can take the ten inputs and reduce them to six (or some smaller number) because we can still use the six inputs to re-create the range of variation that the ten inputs can create.

There are possibilities for performing more perception experiments, which can be shorter-term but useful future work. We can further study motion variation by adjusting these variables: different characters (other than the one skeleton that we have), a larger number of characters, skinned characters, and camera viewpoint. Do these variables affect how users perceive motion variation? In addition, the level-of-detail is important to the user's perception. How many variants are needed at different levels? Or at different camera viewpoints? We can try to add noise to existing motion using the strawman method to see if users can perceive those as more unnatural. These are all possible questions that can be explored.

Another area of future work that can be a longer term goal is to use the idea of variation to compress motion data. If we can say that a set of motion clips are variations of each other, it may be possible to discard some of these motions. This is beause we can potentially re-synthesize a discarded motion from the remaining motions, since the discarded one is a variation of the remaining ones.

Bibliography

[1] Ai-implant, 2003. www.ai-implant.com. 2.1

[2] Massive, 2006. www.massivesoftware.com. 2.1

[3] Matt Anderson, Eric McDaniel, and Stephen Chenney. Constrained animation of flocks. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 286–297, 2003. 2.1

[4] Okan Arikan and D. A. Forsyth. Interactive motion generation from examples. *ACM Transactions on Graphics*, 21(3):483–490, July 2002. 2.2, 3

[5] Z. Bar-Joseph, R. El-Yaniv, D. Lischinski, and M. Werman. Texture mixing and texture movie synthesis using statistical learning. *IEEE Transactions on Visualization and Computer Graphics*, 7(2), 2001. 2.4

[6] O. Burchan Bayazit, Jyh-Ming Lien, and Nancy M. Amato. Better group behaviors in complex environments using global roadmaps. In *ICAL 2003: Proceedings of the eighth international conference on Artificial life*, pages 362–370, Cambridge, MA, USA, 2003. MIT Press. 2.2, 4

[7] V.J. Blue and J.L Adler. Cellular automata model of emergent collective bi-directional pedestrian dynamics. In *Artificial Life VII, The Seventh International Conference on the Simulation and Synthesis of Living Systems*, 2000. 2.1

[8] Bobby Bodenheimer, Anna V. Shleyfman, and Jessica K. Hodgins. The effects of noise on the perception of animated human running. In N. Magnenat-Thalmann and D. Thalmann, editors, *Computer Animation and Simulation '99*, pages 53–63. Springer-Verlag, Wien, September 1999. Eurographics Animation Workshop. 1.3, 2.4, 5

[9] Matthew Brand and Aaron Hertzmann. Style machines. In *SIGGRAPH 2000*, pages 183–192, 2000. 2.4

[10] Michael S. Branicky, Ross Alan Knepper, and James Kuffner. Path and trajectory diversity: Theory and algorithms. In *International Conference on Robotics and Automation*. IEEE RAS, May 2008. 1.2, 2.3, 4.2.3, 4.7.2, 4.7.2

[11] O. Brock and L. Kavraki. Decomposition-based motion planning: A framework for real-time motion planning in high-dimensional configuration spaces, 2001. 2.2

[12] O. Brock and O. Khatib. Elastic strips: A framework for motion generation in human environments. *Int. Journal of Robotics Research*, pages 1031–1052, 2002. 2.2

[13] Armin Bruderlin and Lance Williams. Motion signal processing. *Computer Graphics*, 29 (Annual Conference Series):97–104, 1995. 2.2, 3.7

[14] Joel Chestnutt, James Kuffner, Koichi Nishiwaki, and Satoshi Kagami. Planning biped navigation strategies in complex environments. In *Proceedings of the 2003 Intl. Conference on Humanoid Robots*, October 2003. 4.7.3

[15] Wallace Ching and Norman Badler. Fast motion planning for anthropometric figures with many degrees of freedom. *International Conference on Robotics and Automation*, pages 2340–2345, May 1992. 2.2

[16] Min Gyu Choi, Jehee Lee, and Sung Yong Shin. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Trans. Graph.*, 22(2):182–203, 2003. 2.2, 4

[17] Mira Dontcheva, Gary Yngve, and Zoran Popovic. Layered acting for character animation. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 409–416, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-709-5. 2.2

[18] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *ICCV '99: Proceedings of the International Conference on Computer Vision-Volume 2*, page 1033, 1999. 2.4

[19] Lawrence Erickson and Steven LaValle. Survivability: Measuring and ensuring path diversity. In *IEEE International Conference on Robotics and Automation*, 2009. 2.3, 4.7.2

[20] Jason Ernst, Oded Vainas, Christopher T Harbison, Itamar Simon, and Ziv Bar-Joseph. Reconstructing dynamic regulatory maps. *Molecular Systems Biology*, page 3:74, 2007. 5.3

[21] Claudia Esteves, Gustavo Arechavaleta, Julien Pettre, and Jean-Paul Laumond. Animation planning for virtual characters cooperation. *ACM Trans. Graph.*, 25(2):319–339, 2006. 2.2, 4

[22] Jeff Forbes, Timothy Huang, Keiji Kanazawa, and Stuart J. Russell. The BATmobile: Towards a bayesian automated taxi. In *IJCAI*, pages 1878–1885, 1995. 2.4

[23] Jeffrey Forbes, Nikunj Oza, Ronald Parr, and Stuart Russell. Feasibility study of fully automated traffic using decision-theoretic control, 1997. Cal. PATH Research Report UCB-ITS-PRR-97-18, Inst. of Transportation Studies, U. C. Berkeley. 2.4

[24] Nir Friedman, Kevin Murphy, and Stuart Russell. Learning the structure of dynamic probabilistic networks. pages 139–147, 1998. 1.3, 2.4, 5, 5.2

[25] John Funge, Xiaoyuan Tu, and Demetri Terzopoulos. Cognitive modeling: knowledge, reasoning and planning for intelligent characters. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 29–38, 1999. 2.1

[26] Russell Gayle, Avneesh Sud, Ming C. Lin, and Dinesh Manocha. Reactive deformation roadmaps: motion planning of multiple robots in dynamic environments. In *Intelligent Robots and Systems (IROS)*, pages 3777–3783, 2007. 2.2, 4

[27] Zoubin Ghahramani. Learning dynamic Bayesian networks. *Lecture Notes in Computer Science*, 1387, 1998. 1.3, 2.4, 5, 5.2

[28] Michael Gleicher, Hyun Joon Shin, Lucas Kovar, and Andrew Jepsen. Snap-together motion: assembling run-time animations. *ACM Trans. Graph.*, 22(3):702–702, 2003. ISSN 0730-0301. 2.2, 3

[29] Jared Go, Thuc D. Vu, and James J. Kuffner. Autonomous behaviors for interactive vehicle animations. *Graph. Models*, 68(2):90–112, 2006. ISSN 1524-0703. 2.3

[30] Colin Green and Alonzo Kelly. Toward optimal sampling in the space of paths. In *13th Intl. Symposium of Robotics Research*, November 2007. 1.2, 2.3, 4.2.3, 4.7.2, 4.7.2

[31] Keith Grochow, Steven L. Martin, Aaron Hertzmann, and Zoran Popović. Style-based inverse kinematics. *ACM Trans. Graph.*, 23(3):522–531, 2004. 2.4

[32] C. M. Harris and D. M. Wolpert. Signal-dependent noise determines motor planning. *Nature*, 394:780–784, August 1998. 2.4

[33] Dirk Helbing and Peter Molnar. Social force model for pedestrian dynamics. *Physical Review E*, 51:4282, 1995. 2.1

[34] Dirk Helbing, Illes Farkas, and Tamas Vicsek. Simulating dynamical features of escape panic. *Nature*, 407:487–490, September 2000. 2.1

[35] Aaron Hertzmann. Machine learning for computer graphics: A manifesto and tutorial, 2003. 2.4

[36] Eugene Hsu, Kari Pulli, and Jovan Popovic. Style translation for human motion. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1082–1089, New York, NY, USA, 2005. ACM Press. 2.4

[37] Leslie Ikemoto, Okan Arikan, and David Forsyth. Learning to move autonomously in a hostile world. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 46, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1187112.1187167. 2.3

[38] Leslie Ikemoto, Okan Arikan, and David Forsyth. Generalizing motion edits with gaussian processes. *ACM Trans. Graph.*, 28(1):1–12, 2009. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1477926.1477927. 2.4

[39] M. Kallmann, A. Aubel, T. Abaci, and D. Thalmann. Planning collision-free reaching motions for interative object manipulation and grasping, 2003. 2.2

[40] A. Kamphuis and M. H. Overmars. Finding paths for coherent groups using clearance. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 19–28, 2004. 2.1

[41] Lydia Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Trans. on Robotics and Automation*, 1994. 2.2, 2.3

[42] Lydia E. Kavraki, P. Svestka, Jean claude Latombe, and Mark H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration space. In *International Transactions on Robotics and Automation*, pages 566–580, 1996. 2.3

[43] Ross Knepper and Matthew Mason. Path diversity is only part of the problem. In *IEEE International Conference on Robotics and Automation*, 2009. 2.3

[44] Yoshihito Koga, Koichi Kondo, James Kuffner, and Jean-Claude Latombe. Planning motions with intentions. *Computer Graphics*, 28(Annual Conference Series):395–408, 1994. 2.2

[45] Lucas Kovar, Michael Gleicher, and Frédéric Pighin. Motion graphs. *ACM Transactions on Graphics*, 21(3):473–482, July 2002. 2.2, 3

[46] Lucas Kovar, Michael Gleicher, and John Schreiner. Footskate cleanup for motion capture editing. In *ACM SIGGRAPH/Eurographics symposium on Computer animation (SCA) 2002*, pages 97–104, 2002. 5.5

[47] J. Kuffner and S. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA'2000), San Francisco, CA*, April 2000. 2.2

[48] James J. Kuffner. Efficient optimal search of euclidean-cost grids and lattices. *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, July 2004. 4.6

[49] James J. Kuffner, Jr. Goal-directed navigation for animated characters using real-time path planning and control. *Proc. of CAPTECH '98 : Workshop on Modelling and Motion Capture Techniques for Virtual Environments*, November 1998. 2.2, 3

[50] Taesoo Kwon, Young-Sang Cho, Sang I Park, and Sung Yong Shin. Two-character motion analysis and synthesis. *IEEE Transactions on Visualization and Computer Graphics*, 14 (3):707–720, 2008. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2008.22. 2.4

[51] Yu-Chi Lai, Stephen Chenney, and ShaoHua Fan. Group motion graphs. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 281–290, 2005. 2.1

[52] Manfred Lau and James J. Kuffner. Behavior planning for character animation. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 271–280, August 2005. 1.1, 2.2, 3

[53] Manfred Lau and James J. Kuffner. Precomputed search trees: Planning for interactive goal-driven animation. In *2006 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 299–308, September 2006. 1.2, 2.3, 4, 4.1, 4.2.2, 4.2.3, 4.7.2

[54] S. LaValle. Rapidly-exploring random trees: A new tool for path planning, 1998. 2.2

[55] S. LaValle and J. Kuffner. Rapidly-exploring random trees: Progress and prospects, 2000. In Workshop on the Algorithmic Foundations of Robotics. 2.2

[56] S. M. LaValle. *Planning Algorithms*. Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/), 2006. 3.1, 3.4, 3.6, 4.2.3

[57] Steven M. Lavalle and James J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308, 2001. 2.3

[58] Jehee Lee and Kang Hoon Lee. Precomputing avatar behavior from human motion data. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 79–87, Aire-la-Ville, Switzerland, 2004. Eurographics Association. ISBN 3-905673-14-2. 2.3

[59] 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 Transactions on Graphics*, 21(3):491–500, July 2002. 2.2, 3

[60] P. Leven and S. Hutchinson. A framework for real-time path planning in changing environments. In *Intl. J. Robotics Research*, 2002. 2.3

[61] Yan Li, Tianshu Wang, and Heung-Yeung Shum. Motion texture: a two-level statistical model for character motion synthesis. In *SIGGRAPH 2002*, pages 465–472, 2002. 2.4

[62] Ying Liu and Norman I. Badler. Real-time reach planning for animated characters using hardware acceleration. In *CASA '03: Proceedings of the 16th International Conference on Computer Animation and Social Agents*, page 86, Washington, DC, USA, 2003. IEEE Computer Society. 2.2

[63] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979. 2.2, 3.3

[64] J. Maim, B. Yersin, and D. Thalmann. Unique instances for crowds. In *Computer Graphics and Applications*, 2008. 2.4, 5

[65] James McCann and Nancy Pollard. Responsive characters from motion fragments. *ACM Trans. Graph.*, 26(3):6, 2007. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1276377.1276385. 2.3

[66] Rachel McDonnell, Simon Dobbyn, and Carol O'Sullivan. Crowd creation pipeline for games. In *International Conference on Computer Games*, pages 181–190, 2006. 1, 1.3, 5

[67] Rachel McDonnell, Michéal Larkin, Simon Dobbyn, Steven Collins, and Carol O'Sullivan. Clone attack! perception of crowd variety. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–8. ACM, 2008. doi: http://doi.acm.org/10.1145/1399504.1360625. 2.4, 5, 5.6.3

[68] Alberto Menache. *Understanding Motion Capture for Computer Animation and Video Games*. Morgan Kaufmann Publishers Inc., 1999. ISBN 0124906303. 2.2

[69] Mark Mizuguchi, John Buchanan, and Tom Calvert. Data driven motion transitions for interactive games. *Eurographics 2001 Short Presentations*, September 2001. 2.2

[70] Shmuel Moradoff and Dani Lischinski. Constrained synthesis of textural motion for articulated characters. *Visual Computer*, 20(4):253–265, 2004. 2.4

[71] Richard E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, 2003. 5.3.2

[72] Dirk Ormoneit, Hedvig Sidenbladh, Michael J. Black, and Trevor Hastie. Learning and tracking cyclic human motion. In *NIPS*, pages 894–900, 2000. 2.4

[73] N. Pelechano, K. O'Brien, B. Silverman, and N. Badler. Crowd simulation incorporating agent psychological models, roles and communication. *First International Workshop on Crowd Simulation*, 2005. 2.1

[74] N. Pelechano, J. M. Allbeck, and N. I. Badler. Controlling individual agents in high-density crowd simulation. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 99–108, 2007. 2.1

[75] Ken Perlin. Real time responsive animation with personality. In *Transactions on Visualization and Computer Graphics*, pages 5–15, 1995. 1.3, 2.4, 5, 5.6.4

[76] Julien Pettre and Jean-Paul Laumond. A motion capture-based control-space approach for walking mannequins. *Comput. Animat. Virtual Worlds*, 17(2):109–126, 2006. 2.2

[77] Julien Pettre, Jean-Paul Laumond, and Thierry Simeon. A 2-stages locomotion planner for digital actors. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 258–264, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-659-5. 2.2, 4

[78] Julien Pettre, Jean-Paul Laumond, and Daniel Thalmann. A navigation graph for real-time crowd animation on multilayered and uneven terrain. *First International Workshop on Crowd Simulation (V-CROWDS'05)*, 2005. 2.2, 4

[79] Katherine Pullen and Christoph Bregler. Animating by multi-level sampling. In *Proceedings of the Computer Animation*, pages 36–42. IEEE Computer Society, 2000. 2.4

[80] Katherine Pullen and Christoph Bregler. Motion capture assisted animation: Texturing and synthesis. *ACM Transactions on Graphics*, 21(3):501–508, July 2002. 2.2, 2.4

[81] Katherine Pullen and Christoph Bregler. Synthesis of cyclic motions with texture (unpublished), 2002. 2.4

[82] S. Quinlan and O. Khatib. Elastic bands: Connecting path planning and control. In *Proc. of IEEE Conf. on Robotics and Automation*, pages 802–807, 1993. 2.2

[83] P. S. A. Reitsma and N. S. Pollard. Evaluating motion graphs for character navigation. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 89–98, Aire-la-Ville, Switzerland, 2004. Eurographics Association. ISBN 3-905673-14-2. 2.3

[84] Paul S. A. Reitsma and Nancy S. Pollard. Evaluating motion graphs for character animation. *ACM Transactions on Graphics*, 26(4):18:1–18:24, 2007. 2.3

[85] Craig Reynolds. Interaction with groups of autonomous characters. 2.1

[86] Craig Reynolds. Big fast crowds on ps3. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121, 2006. 2.1

[87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, New York, NY, USA, 1987. ACM Press. 2.1, 2.2, 2.3

[88] C. Rose, M. Cohen, and B. Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Application*, 18(5):32–40, 1998. 2.4, 3.7

[89] Brian Salomon, Maxim Garber, Ming C. Lin, and Dinesh Manocha. Interactive navigation in complex environments using path planning. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 41–50, 2003. 2.2, 4

[90] Terrence J. Sejnowski. Making smooth moves. *Nature*, 394:725–726, August 1998. 2.4

[91] Wei Shao and Demetri Terzopoulos. Autonomous pedestrians. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 19–28, 2005. 2.1

[92] Hedvig Sidenbladh, Michael J. Black, and David J. Fleet. Stochastic tracking of 3d human figures using 2d image motion. In *ECCV (2)*, pages 702–718, 2000. 2.4

[93] Hedvig Sidenbladh, Michael J. Black, and L. Sigal. Implicit probabilistic models of human motion for synthesis and tracking. In *ECCV (1)*, pages 784–800, 2002. 2.4

[94] T. Sim'eon, J. Laumond, and C. Nissoux. Visibility-based probabilistic roadmaps for motion planning. 1999. 2.2

[95] Avneesh Sud, Erik Andersen, Sean Curtis, Ming Lin, and Dinesh Manocha. Real-time path planning for virtual agents in dynamic environments. *IEEE Virtual Reality Conference*, pages 91–98, 2007. 2.3

[96] Avneesh Sud, Russell Gayle, Erik Andersen, Stephen Guy, Ming Lin, and Dinesh Manocha. Real-time navigation of independent agents using adaptive roadmaps. In *VRST '07: Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, pages 99–106, 2007. 2.2, 4

[97] Gita Sukthankar, Michael Mandel, Katia Sycara, and Jessica Hodgins. Modeling physical capabilities of humanoid agents using motion capture data. In *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 344–351, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 1-58113-864-4. doi: http://dx.doi.org/10.1109/AAMAS.2004.173. 2.3

[98] Mankyu Sung, Lucas Kovar, and Michael Gleicher. Fast and accurate goal-directed motion synthesis for crowds. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 291–300, 2005. 2.2, 2.3

[99] Daniel Thalmann, Laurent Kermel, William Opdyke, and Stephen Regelous. Crowd and group animation (siggraph course notes). 2005. 1, 1, 1.3, 5

[100] Emanuel Todorov and Michael I. Jordan. A minimal intervention principle for coordinated movement. In *NIPS*, pages 27–34. MIT Press, 2002. 2.4

[101] Emanuel Todorov and Michael I. Jordan. Optimal feedback control as a theory of motor coordination. In *Nature Neuroscience*, pages 1226–1235, 2002. 2.4

[102] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. *ACM Trans. Graph.*, 25(3):1160–1168, 2006. 2.1

[103] Adrien Treuille, Yongjoon Lee, and Zoran Popović. Near-optimal character animation with continuous control. *ACM Trans. Graph.*, 26(3):7, 2007. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/1276377.1276386. 2.3

[104] S. Udupa. *Collision Detection and Avoidance in Computer Controlled Manipulators*. PhD thesis, Department of Electrical Engineering, California Institute of Technology, 1977. 2.2

[105] Jur van den Berg, Ming Lin, and Dinesh Manocha. Reciprocal velocity obstacles for real-time multi-agent navigation. In *Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA'2008)*, 2008. 2.3

[106] Jur van den Berg, Sachin Patil, Jason Sewall, Dinesh Manocha, and Ming Lin. Interactive navigation of multiple agents in crowded environments. In *SI3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 139–147, 2008. 2.3

[107] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 355–360, 2001. 2.4

[108] D. Wiley and J. Hahn. Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Application*, 17(6):39–45, 1997. 2.4, 3.7

[109] Andrew P. Witkin and Zoran Popović. Motion warping. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 105–108, August 1995. 2.2, 3.7

[110] Katsu Yamane, James J. Kuffner, and Jessica K. Hodgins. Synthesizing animations of human manipulation tasks. In *ACM SIGGRAPH*, pages 532–539, New York, NY, USA, 2004. ACM Press. 2.2

[111] Y. Yang and O. Brock. Elastic roadmaps: Globally task-consistent motion for autonomous mobile manipulation. *Proceedings of Robotics: Science and Systems*, 2006. 2.2, 4

[112] Qinxin Yu and Demetri Terzopoulos. A decision network framework for the behavioral animation of virtual humans. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2007. 2.1, 2.4