Chapter 6

Parameterized Motion

Many actions are naturally described through a relatively small number of high-level features. For example, a punch might be characterized by its speed and target location, and a walk cycle might be characterized by the final position and orientation of the character relative to its initial configuration. It would be convenient to animate these kinds of motions simply by stating what these features should be, rather than by adjusting low-level properties such as joint orientations. One way to accomplish this is to collect multiple variations of a motion (typically through motion capture) and then using blending methods like the one discussed in Chapter 5 to create interpolations. The set of interpolations is a continuous space of related motions, and if the interpolation weights can be mapped to relevant motion features, then the result is a *parameterized motion* that provides the intuitive control we seek.

Existing methods for constructing parameterized motions are designed for small data sets consisting exactly of example motions that uniformly (though perhaps not regularly) sample a predetermined range of variation [96, 76, 78, 68]. For instance, if one wanted a character that could reach to any location inside of a square, then the data set would consist of several reaching motions (separated into individual data files with analogous starting and ending poses) that target positions distributed evenly within that square. Most real-world data sets are not so contrived. At the very least, the data is almost never limited to a single kind of action, since characters generally must be able to perform a variety of tasks. It is also common for a given data file to contain more than just a single, logically independent action. If nothing else, the action of interest is usually surrounded

by "filler" where the actor prepares for or recovers from the movement, and a single data file might contain multiple actions because a continuous sequence of motion is often more natural than performing individual movements in isolation. To extract variations of a specific action, at present there is little recourse but to scan through the data and manually identify the start and end frames of each example motion segment. This can be tedious and time consuming, even when the data is annotated with descriptive labels. For example, a data file labelled "punching" might contain both several individual punches and related but distinct actions such as dodging a counter-blow. Also, in general one does not know in advance what range of variation is spanned by the data. Indeed, an important step in analyzing the utility of a data set is determining what sorts of actions can be derived from the available examples.

This chapter provides automated tools for constructing parameterized motions from large data sets. Specifically, this chapter introduces a method for extracting logically similar motions segments from a data set (i.e., the example motions) and shows how to construct efficient and accurate parameterizations of the space of interpolations. We start in Section 6.1 with an overview of our methods and a summary of our contributions. Section 6.2 then presents an algorithm for efficiently searching a data set for motion segments similar to a query motion, and it analyzes experiments performed on a test data set containing 37,000 frames (about ten minutes of motion sampled at 60Hz). Next, Section 6.3 introduces an algorithm for creating accurate parameterizations of the space of interpolations and demonstrates this algorithm on several examples. Finally, Section 6.4 concludes with a brief discussion of the advantages and limitations of our methods.

6.1 Overview

6.1.1 Searching Motion Data Sets

Given a segment of the data set (the *query*), our system automatically locates and extracts motion segments that are "similar", i.e., that represent variations of the same action or sequence of actions. Our method uses three key ideas, each of which is a contribution toward solving the problem of searching a motion data set.

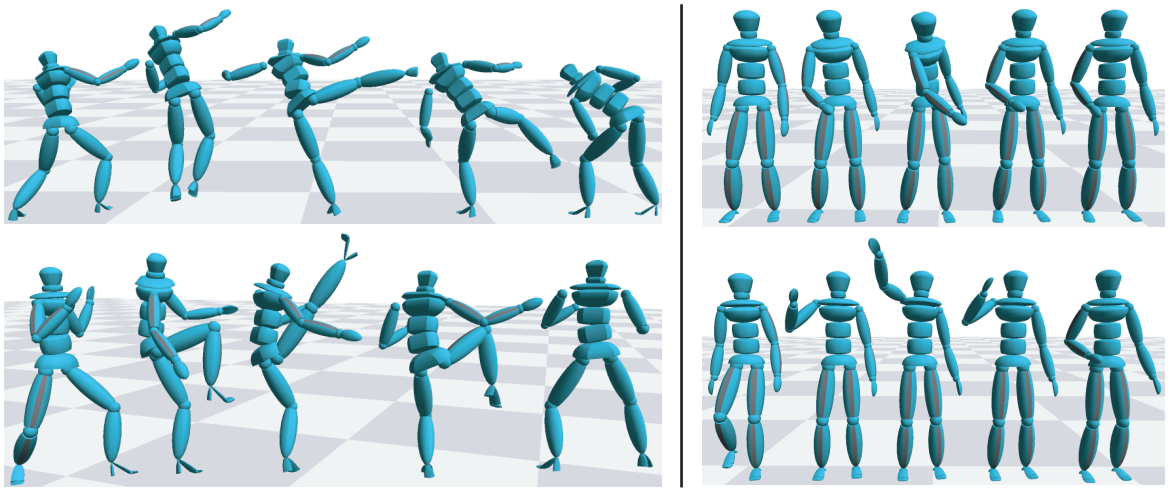

Figure 6.1: Logically similar motions may be numerically dissimilar. **Left:** A standing front kick vs. a leaping side kick. Note the differences in the arms, torso posture, and kick trajectory. **Right:** While these two reaching motions have somewhat similar skeletal postures, the changes in posture are in completely opposite directions.

1. **Multi-step search.** Existing search methods determine motion similarity through direct numerical comparison; i.e., they compare the "distance" between the motions against a threshold. This strategy can only reliably find motions that are *numerically* similar to the query in the sense that corresponding skeletal poses are roughly the same — a large distance may reflect either that motions are unrelated or that they are different variations of the same action (Figure 6.1), and there is no way to distinguish between these cases. On the other hand, in a large data set it is likely that some logically similar motions will also be numerically similar. We therefore add robustness to the search by concentrating on finding these closer motions and then using them as new queries in order to find more distant motions.

2. **Using time correspondences to determine similarity.** A simple way of measuring numerical similarity is to identify corresponding frames and compare their average distance to a threshold value. We complement this by analyzing the correspondences themselves: similar motions are required to have "clear" time correspondences. This is based on the intuition that if two motions are similar, it should be easy to pick out corresponding events.

3. **Interactivity through precomputation.** To provide interactive speeds, we do not start each search from scratch. Instead, we precompute a *match web*, which is a compact and efficiently searchable representation of all possibly similar motion segments.

6.1.2 Creating Parameterized Motions

Once example motions have been collected, they can be interpolated using the methods of Chapter 5, and the space of interpolations can be converted into a parameterized motion through a user-specified *parameterization function* $\mathbf{f}$ that picks out relevant motion features. For example, $\mathbf{f}$ might compute the position of the hand at the apex of a reach or the average speed and curvature of the root path during a walk cycle. Abstractly, $\mathbf{f}$ maps interpolation weights to motion parameters, and our goal is to compute $\mathbf{f}^{-1}$: given a set of target parameters, we want interpolation weights that produce the appropriate motion. Since $\mathbf{f}^{-1}$ in general has no closed form representation, it is common to approximate it with scattered data interpolation[1] methods that assign each example motion a weight based on the distance between its parameters and the target parameters [76]. We adopt a similar approach, but offer several improvements over previous work:

1. **Interpolation Weight Constraints.** Interpolation can only reliably create new motions near the examples, which means only a finite region of parameter space is accessible. In particular, interpolation weights should be convex or nearly convex in order to prevent unreasonable amounts of extrapolation. Existing methods place no constraints on interpolation weights and can break down when specified parameters are far from those of the example motions. Our algorithm ensures that interpolation weights are "almost" convex, and the user can directly control the allowable deviation from strict convexity. This limits the amount of extrapolation performed during motion synthesis and effectively projects unattainable parameter requests to be projected back onto the accessible portion of parameter space.

[1]The phrases "scattered data interpolation" and "motion interpolation" should not be confused: the former is a general method for inferring function values in between measurements, and the latter is a motion synthesis technique (namely, blending with constant weights).

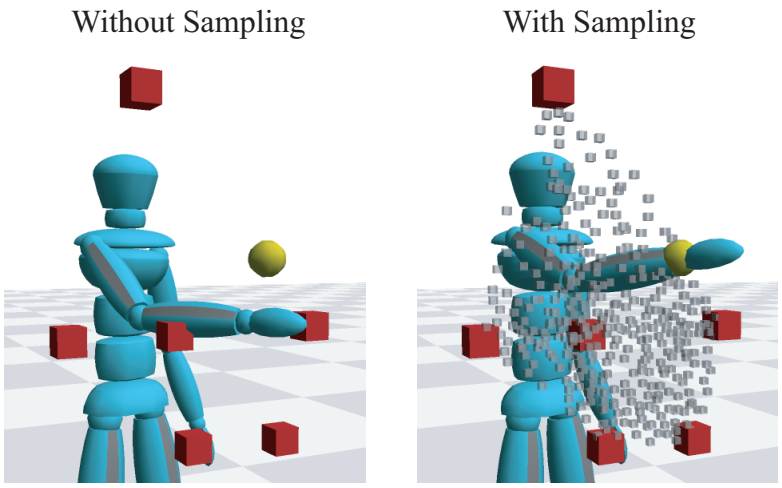

Figure 6.2: Left: Six example reaching motions create a sparse sampling of parameter space that leads to an inaccurate parameterization. The red cubes indicate parameters (i.e., wrist locations) of the example motions and the yellow sphere shows the desired location of the wrist. **Right:** We automatically generate a denser sampling that provides greater accuracy; grey cubes represent new samples.

2. **Accuracy.** If the examples are not sufficiently close in parameter space, direct application of scattered data interpolation may yield inaccurate results (Figure 6.2). We correct this by automatically sampling the set of valid interpolations in order to densely sample the accessible region of parameter space.

3. **Efficiency.** Large data sets may contain many example motions. We show how to automatically identify and remove redundant examples to reduce storage requirements. Also, while previous methods have used scattered data interpolation algorithms that require $O(n)$ time for n examples, our method's run time is nearly independent of the number of examples.

6.2 Searching For Motion

This section considers the problem of searching a motion data set for motion segments (called *matches*) that are similar to a query motion $\mathbf{M_q}$, which is itself assumed to be a segment of some motion in the data set. One reason this problem is challenging is because individual frames of motion are high-dimensional objects with non-Euclidean distance metrics (e.g., Equation 4.1 of

Chapter 4). As a result, traditional methods for organizing the data into a spatial hierarchy (such as a BSP-tree) can not be directly applied [12]. A second challenge is that logically similar motions may be numerically dissimilar in the sense that corresponding poses may have very different joint orientations and angular velocities (Figure 6.1). Traditional search algorithms implicitly equate numerical similarity with logical similarity, and as a result they have difficulty distinguishing motions that are unrelated from those that are different versions of the same kind of action.

Our search strategy is to find "close" matches that are numerically similar to the query and then use them as new queries to find more distant matches. Our algorithm for determining numerical similarity allows arbitrary metrics for comparing individual frames, and timing differences are factored out to allow matches to be of different duration than the query. A user executes a search by providing a query and a distance threshold. Numerically similar matches are identified based on this distance threshold, and these matches are automatically submitted as new queries. This process iterates until no new matches are found. Since each match spawns a new query, it is crucial that individual queries be processed quickly. In particular, we would like searching to be fast enough that the distance threshold can be tuned interactively. To make this feasible, the data set is preprocessed into a *match web*, which is a compact and efficiently searchable representation of all motion segments that, given a sufficiently large distance threshold, would be considered numerically similar.

One difficulty is that numerically similar motions might *not* be logically similar. For example, the overall structure of a man walking looks much like that of a woman walking, and reaching for an object appears quite similar to simply touching it. This problem is hard to correct because it involves high-level understanding of what motions mean. For an unlabelled data set, users must independently confirm that matches have the correct meaning. However, most large data sets contain some sort of descriptive labels — if nothing else, a filename or location in the directory hierarchy can serve as clues to a motion's contents. When labels are present, our algorithm restricts its search to semantically relevant parts of the data set.

The remainder of this section proposes criteria for determining whether two motion segments are numerically similar, explains how to build match webs and how to use them to quickly answer similarity queries, and presents experimental results.

6.2.1 Criteria for Numerical Similarity

Two criteria are used to determine whether two motion segments are numerically similar:

1. Corresponding frames should have similar skeleton poses.

2. Frame correspondences should be easy to identify. That is, related events in the motions should be clearly recognizable.

Frame correspondences are generated using the dynamic programming algorithm discussed in Chapter 5. As before, the set of frame correspondences is called a *time alignment*, and time alignments are required to be continuous, monotonic, and non-degenerate. Recall that if a grid is formed where cell (i, j) specifies the distance $D\left(\mathbf{M}_1(t_i), \mathbf{M}_2(t_j)\right)$ between frames $\mathbf{M_1}(t_i)$ and $\mathbf{M_2}(t_j)$, then the time alignment is a path on this grid from the lower left to the upper right such that the total cost of its cells is minimized.

To test the first criterion, the average value of the cells on the time alignment is compared against a user-specified threshold ϵ. The average value is used instead of the total in order to factor out differences in path length. The second criterion can be interpreted in terms of the *local* optimality of the time alignment. If a cell on the time alignment is a horizontal or vertical 1D local minimum, then the frame correspondence is strong in the sense that holding one frame fixed and varying the other only yields more dissimilar skeletal poses. To illustrate, consider the top of Figure 6.3, which shows the distance grid for two different walk cycles. The magenta path on the right is the calculated time alignment and the highlighted cells on the right show all of the horizontal and vertical 1D minima. Note that nearly every cell on the time alignment is one of these minima. This is because frames at the same point in the locomotion cycle are far more similar than frames that are out of phase. The bottom of Figure 6.3 repeats this analysis for two kicking motions. While the average distance between corresponding frames is about 5 times higher, frames

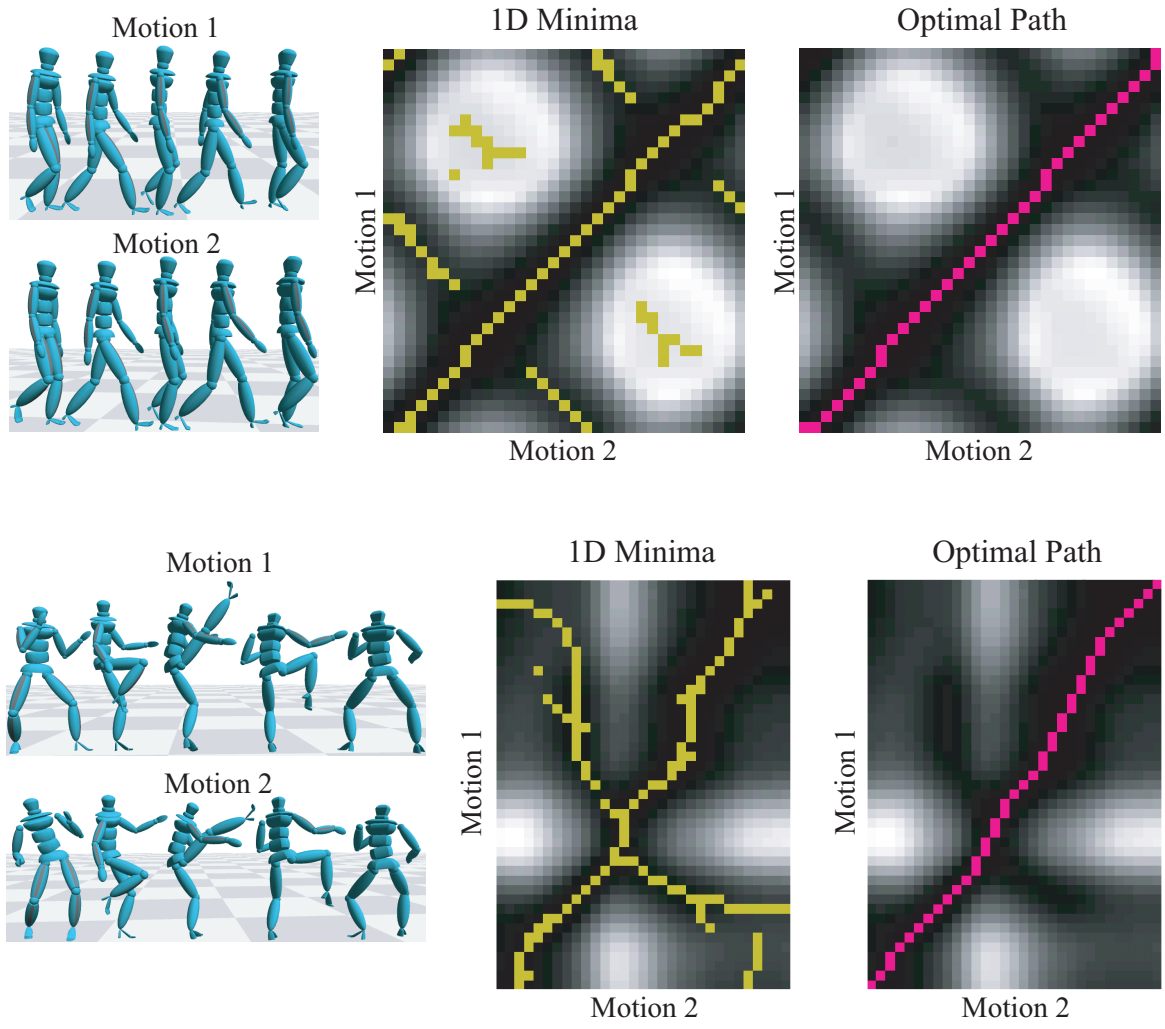

Figure 6.3: Comparing time alignments (yellow cells on the left) with local minima locations (magenta cells on the right). Darker pixels show smaller frame distances. Local minima locations were *not* used when computing the time alignments. **Top:** Two walk cycles. **Bottom:** Two kicks.

at related parts of the kick (chambering, extension, and retraction) are still more similar than pairs of unrelated frames. This is again reflected in the local minima of D: about 60% of the time alignment's cells are 1D minima, and the rest are close to 1D minima.

Ideally every cell on the time alignment would be a local minimum, since then each correspondence would be "obvious". In practice, however, this is overly restrictive. The finite sampling rate causes noise in the exact location of minima, and motions that have stretches of similar frames

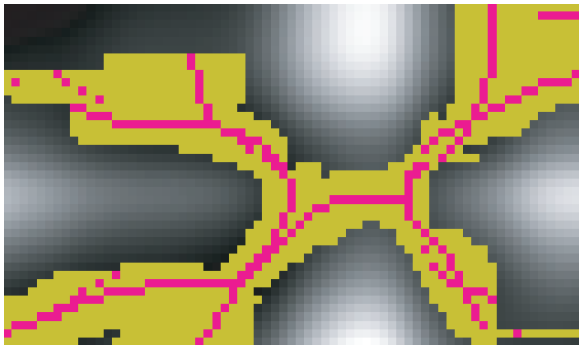

Figure 6.4: Local minima (magenta) are extended to form the valid region (yellow).

(e.g., pauses) produce basins in the distance grid where the locations of minima are effectively arbitrary. We therefore instead require cells on the time alignment to be *near* 1D minima. To enforce this, we compute all 1D local minima and extend each one along the directions in which it is a minimum (horizontal, vertical, or both) until a cell is encountered whose value is at least α percent larger than the minimum's value, where α is defined by the user (see Figure 6.4; we set α to 15%). We call the resulting region on the grid the *valid region*. The time alignment is restricted to lie within the valid region, which can be implemented by modifying the dynamic programming algorithm so it only processes cells in the corresponding portion of the grid. If no time alignment can be created under this restriction, then the motions fail the second criterion and are considered dissimilar.

6.2.2 Match Webs

We now turn to the problem of precomputing potential matches for every motion segment in the database. To illustrate the issues involved, consider a brute force comparison of every possible pair of motion segments. Assume that the query is restricted to be at most m frames and that a distance threshold is provided in advance. The goal is to build a lookup table that lists all the matches for each possible query. If the data set has n frames total ($n \gg m$), then there are $O(mn)$ possible queries, since a query has n possible starting frames and m possible durations (neglecting boundary effects). Accounting for the slope limits used when computing time alignments, a $\tilde{m}$-frame query has $\left(W - \frac{1}{W}\right) \tilde{m}n$ candidate matches, and since the time needed to build a time alignment between

an r-frame motion and an s-frame motion is $O(rs)$, $O(\tilde{m}^3 n)$ work is needed to evaluate all of these candidate matches. The total amount of time needed to process all possible queries is therefore $O(m^4 n^2)$.

This brute force approach suffers from several problems. First, the computational cost is prohibitive, even for small data sets and short query clips. For example, using a data set with just 10 seconds of motion sampled at 60Hz and a maximum query length of 1.5s, a brute force comparison took over 16.5 hours on a 1.3GHz Athlon processor. Second, the results are redundant: if a particular motion segment is numerically similar to the query, then small perturbations of its start and end times will probably yield additional "similar" motion segments (indeed, this redundancy plagues existing efficient search algorithms [22, 92, 16, 44]). Finally, searches at run time are needlessly restricted to a maximum query length and a fixed distance threshold.

Abstractly, a brute force comparison of all possible motion segments is an analysis of the grid of distances formed by computing D for every pair of frames in the data set. Specifically, comparing any two motion segments amounts to solving a dynamic programming problem on a subsection of this grid. The problems mentioned in the previous paragraph stem from the highly redundant nature of this analysis: "nearby" pairs of motion segments are processed independently, even through the corresponding regions of the distance grid may overlap considerably. Through a more careful analysis of the distance grid, we will show that one can efficiently build a compact representation of all possibly similar motion segments, without placing any artificial restrictions on run-time queries.

Without loss of generality, we limit the discussion to finding segments in a single motion $\mathbf{M}_1$ that are numerically similar to another (possibly identical) motion $\mathbf{M}_2$; if a data set has many motions, then they are compared pairwise. Let $\mathbf{M}_i[q, r]$ denote the motion segment $\mathbf{M}_i(t_q), \ldots, \mathbf{M}_i(t_r)$. Two motion segments $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ are *potentially similar* if there is some distance threshold for which they are numerically similar under the criteria of Section 6.2.1. For this to hold, there must exist at least one time alignment starting at cell (a, c) and ending at cell (b, d) that is everywhere inside the valid region and obeys the continuity, monotonicity, and non-degeneracy restrictions. If there is no such time alignment, then $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ cannot be numerically

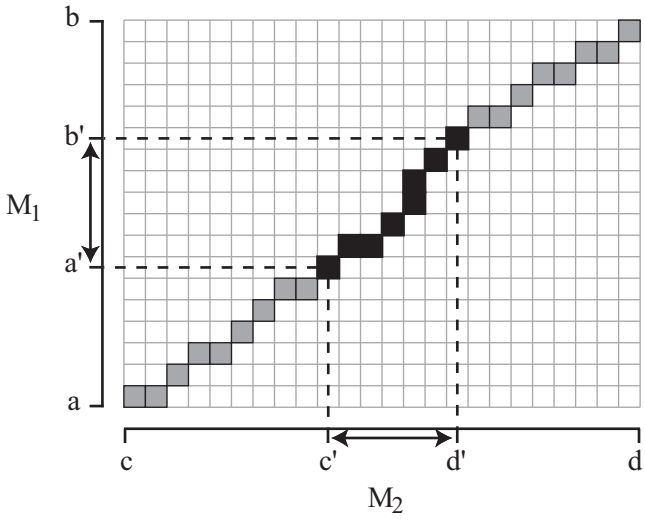

Figure 6.5: Any subregion of an optimal time alignment for $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ is an optimal time alignment for two shorter motion segments $\mathbf{M}_1[a', b']$ and $\mathbf{M}_2[c', d']$.

similar. Otherwise, dynamic programming provides the optimal time alignment between $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$.

Two observations can now be made that allow us to compactly represent *all* potentially similar motion segments. First, if cells (a, c) and (b, d) can be connected with a valid time alignment, then it is likely that nearby pairs $(a \pm \delta, c \pm \delta)$, $(b \pm \delta, d \pm \delta)$ can also be connected. In other words, the boundaries of $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ can be perturbed to find other potentially similar motion segments. It therefore makes sense to identify locally optimal pairs of motion segments where the time alignment has a locally minimal average cell value. Second, any subsection of the optimal time alignment for $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ is itself an optimal time alignment for motion segments inside $\mathbf{M}_1[a, b]$ and $\mathbf{M}_2[c, d]$ (Figure 6.5). An optimal time alignment therefore represents an entire family of potentially similar motion segments, not just a single pair.

In light of these observations, our algorithm searches for long paths on the distance grid that correspond to locally optimal time alignments. It starts by looking for chains of 1D minima that satisfy the continuity, monotonicity, and non-degeneracy restrictions. This is done by first locating all minima that have no neighbors to the left, bottom, or bottom-left, since these cannot be in the interior of a chain. Then for each of these minima a chain is formed by iteratively searching the

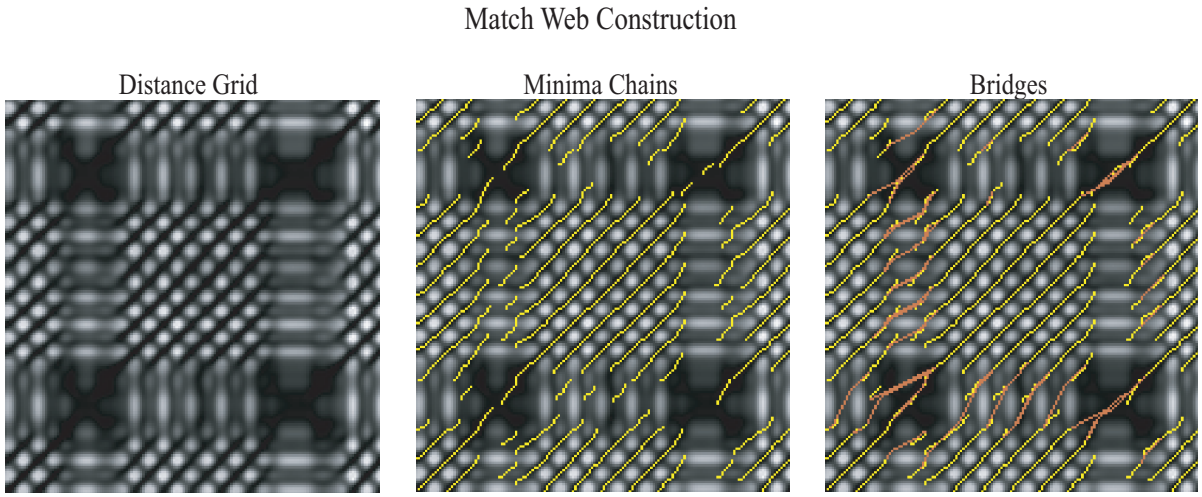

Figure 6.6: A match web is built by computing the distance between every pair of frames, finding chains of local 1D minima (yellow), and adding bridges that connect nearby chains (orange).

top, right, and top-right cells for other minima, making sure along the way that the non-degeneracy condition is not violated. Since some local minima are spurious — for example, walk cycles that are 180 degrees out of phase have local minima at frames where the legs are closest together — some of these chains will not be meaningful. As a heuristic, our algorithm simply removes chains below a threshold length ($0.25s$ in our implementation). Each remaining minima chain is a locally optimal time alignment since moving any cell increases the average distance.

Since the precise location of an individual minimum is somewhat arbitrary (Section 6.2.1), nearby minima chains that ought to be connected may be separate. To be conservative, our algorithm considers connecting any two chains as long as the connecting path is inside the valid region of the grid and has a length (measured via Manhattan distance) less than a threshold L, which was $2s$ in our implementation. For each chain $\mathbf{C}$, we identify other chains $\mathbf{C}'$ in the vicinity. We then compute for each cell C_i on $\mathbf{C}$ the optimal path to each cell on $\mathbf{C}'$ whose Manhattan distance to C_i is less than L. As usual, this path must be within the valid region and obey the other restrictions on time alignments. The path with the smallest average distance is retained as the final connection, or *bridge*, between $\mathbf{C}$ and $\mathbf{C}'$. Note that because bridges must be inside the valid region, nearby chains may not be connectable.

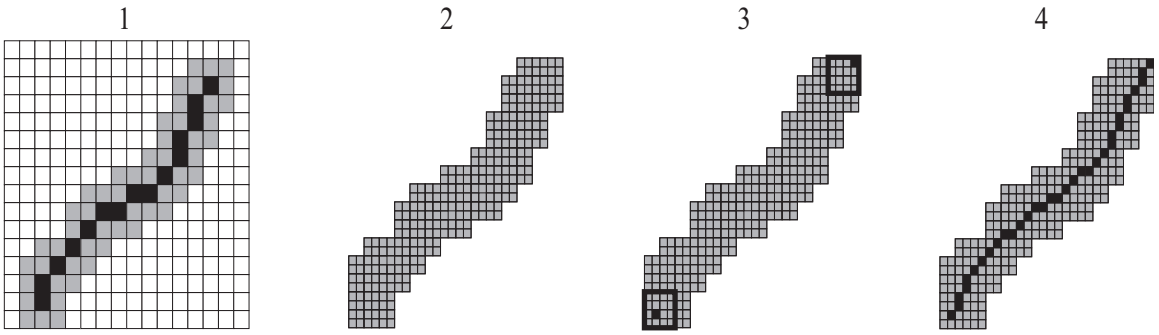

Figure 6.7: To refine a chain, it is first padded (1) and upsampled (2) to form a search region. New endpoints are then found via local search (3) and connected with an optimal path (4).

The result of this procedure is a network of paths on the distance grid, some of them minima chains and others bridges between chains (Figure 6.6). This network represents all potentially similar pairs of motion segments, and we refer to it as a match web. Starting from any cell on any path of the match web, a time alignment can be generated by travelling further down the path and possibly branching off onto connecting paths. While this time alignment is not necessarily optimal, it is close to optimal since every cell is either a local minimum or has a distance value close to a nearby minimum. Also, match webs can be stored compactly since all that must be retained are the grid position and value of each cell on each path.

Match webs can be constructed more efficiently by building them at a low resolution and then refining them at higher resolutions. The lowest resolution match web is built by downsampling $\mathbf{M_1}$ and $\mathbf{M_2}$ and running the algorithm described above. Each minima chain is then refined as shown in Figure 6.7: first, the chain is padded and upsampled to form a search region, then new endpoints are placed at the smallest-valued cells in the vicinity of the old endpoints, and finally these points are joined with an optimal path. Each bridge is handled similarly, except the endpoints are restricted to be on the refined versions of the chains it connects. Figure 6.8 shows an example of refining a low-resolution match web.

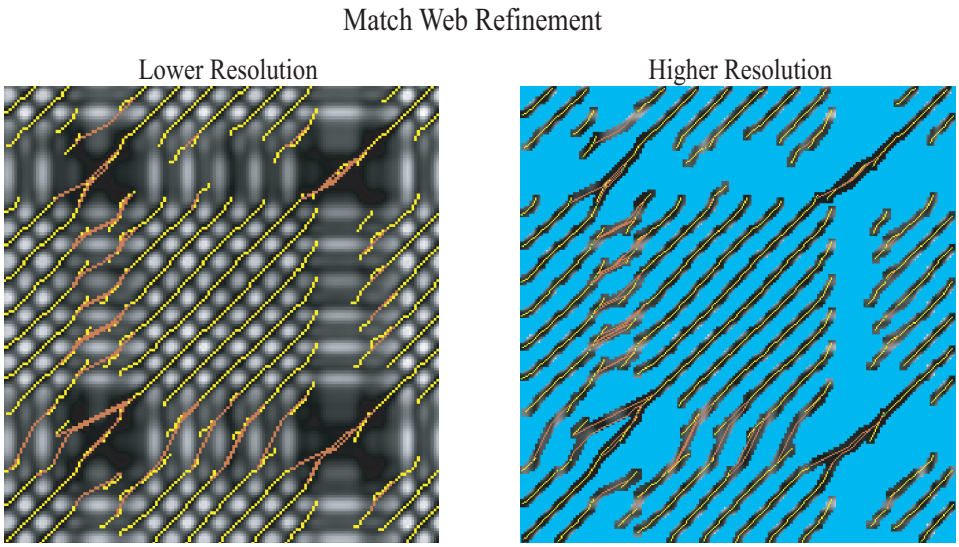

Figure 6.8: A match web may be built at a low resolution (left) and refined at a higher resolution (right). Cells shown in blue are not processed during refinement.

6.2.3 Searching With Match Webs

We now explain how to a use a match web for $\mathbf{M_1}$ and $\mathbf{M_2}$ (where possibly $\mathbf{M_1}$ and $\mathbf{M_2}$ are the same motion) to search for matches. We assume the user has provided a query motion segment $\mathbf{M_1}[a, b]$ and a distance threshold ϵ, and without loss of generality we assume that frames of $\mathbf{M_1}$ correspond to rows of the match web. The case where the query is in $\mathbf{M_2}$ (i.e., it spans columns of the match web rather than rows) is handled similarly. The search algorithm essentially intersects the match web with the rectangle defined by the region from row a to row b, as shown in Figure 6.9. Let a *match sequence* be a sequence of cells formed by selecting any cell of any path on the match web and then travelling down that path, possibly branching off onto connecting paths. The goal is to find all match sequences that span from row a to row b. The algorithm starts by finding every path (minima chain or bridge) that contains a cell in row a. For each such path, the leftmost (i.e., earliest) cell in row a serves as the initial cell of a match sequence. It then checks whether the path contains cells in row b, and if so, a new match sequence is formed by appending all cells up to the last one in row b. The algorithm next considers branching off onto connecting paths to search for additional match sequences. This is done by walking down the path, progressively adding cells

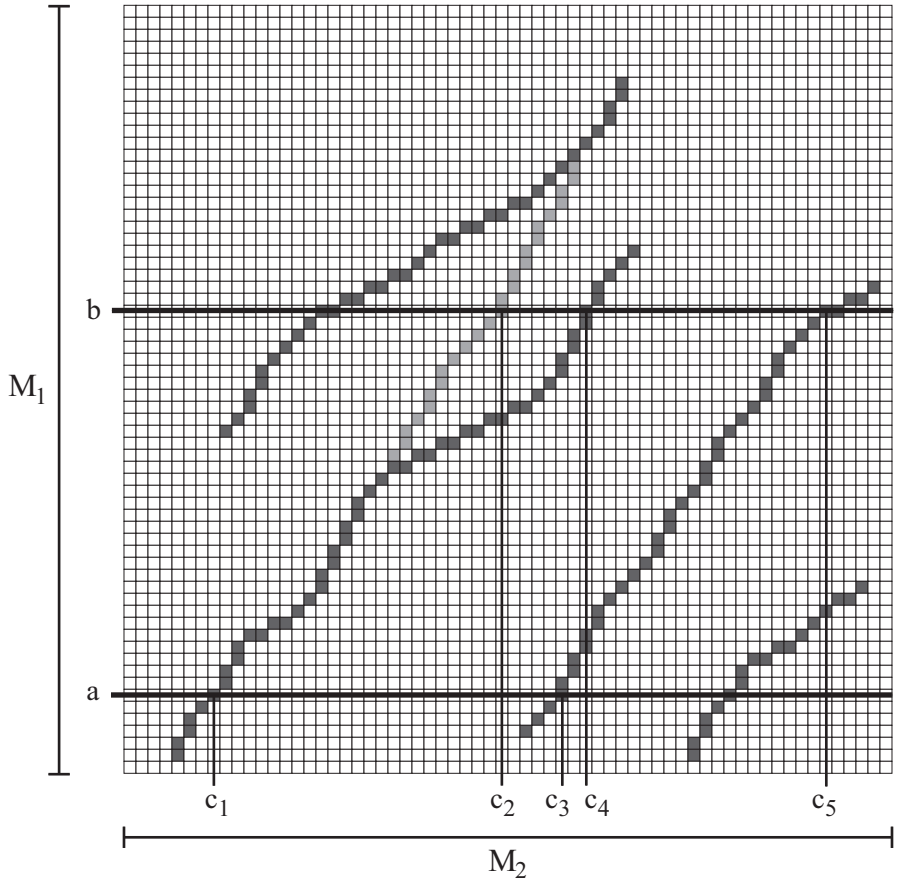

Figure 6.9: A simple search example for the query $\mathbf{M}_1[a, b]$. There are three potential matches: $\mathbf{M}_2[c_1, c_2]$, $\mathbf{M}_2[c_1, c_4]$, and $\mathbf{M}_2[c_3, c_5]$.

to the match sequence and recursively processing each connecting path, until either the end of the path is reached or a cell in row b is encountered.

Each match sequence is a time alignment between the query and a motion segment in $\mathbf{M_2}$ defined by the match sequence's first and last columns. Any match sequence whose average cell value is greater than ϵ is discarded. While each remaining match sequence may be viewed as a match to the query, some of these matches overlap significantly and hence are redundant. In Figure 6.9, for example, this is true of $\mathbf{M_2}[c_1, c_2]$ and $\mathbf{M_2}[c_1, c_4]$. To remove these redundancies, the matches are sorted in order of increasing average distance and placed in an array. The first element of this array is then returned as a match, and every other element that overlaps with it by more than a threshold percentage σ_0 is discarded. This procedure iterates until no matches are left.

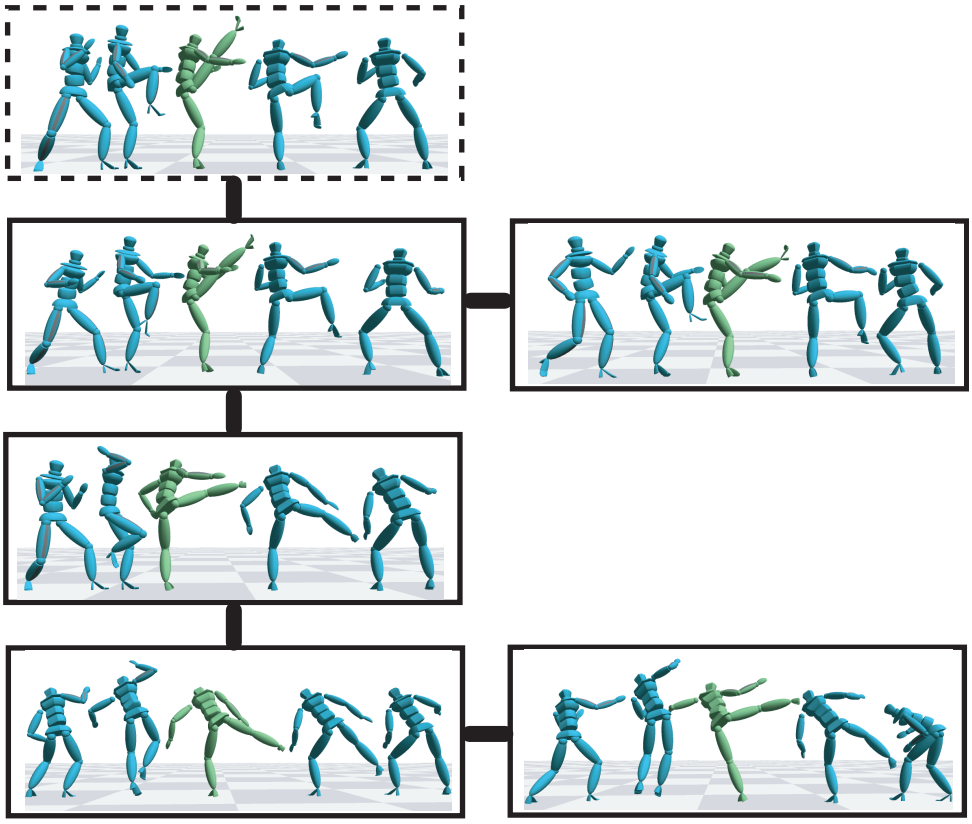

Figure 6.10: An example match graph. The query is in the dotted box and the other nodes are matches. Edges indicate numerical similarity.

The overlap $\sigma(\mathbf{M_2}[a,b], \mathbf{M_2}[c,d])$ of two candidate matches $\mathbf{M_2}[a,b]$ and $\mathbf{M_2}[c,d]$ is defined as

$$\sigma(\mathbf{M_2}[a,b], \mathbf{M_2}[c,d]) = \frac{\|[a,b] \cap [c,d]\|}{\min\left(\|a,b\|, \|c,d\|\right)}, \tag{6.1}$$

where $\|[t_1,t_2]\|$ is the size of the interval $[t_1,t_2]$.

So far we have described how to find the matches that are closest to the query, which we call first-tier matches. To find more distant matches, the search algorithm is repeated for each first-tier match, yielding second-tier matches. This process continues until no new matches are found. In this manner a graph is built where the nodes are motion segments and edges between nodes indicate that the segments are numerically similar. We call this data structure a *match graph* (Figure 6.10). Each edge in the match graph is associated with a time alignment and is assigned a cost equal to the average value of this time alignment's cells. The distance between two nodes is defined as the

cost of the minimal-cost connecting path on the match graph. The match graph can by itself be of interest since it depicts numerical similarity relationships in the matches. To illustrate, Figure 6.10 shows a match graph where the query was a front kick; note the progression from front kick to standing side kick to leaping side kick.

When processing queries other than the initial query, it must be determined whether each "new" match is a heretofore unseen motion or a duplicate of an existing match. When doing this, one must account for the fact that, in practice, duplicates will overlap a great deal but not span identical frame intervals. Let $\mathbf{M}$ be the current query and $\mathbf{M}'$ be a newly identified match. Our system starts by comparing $\mathbf{M}'$ against each node and recording the one with the greatest degree of overlap, $\mathbf{M_{max}}$, where the overlap is defined as in Equation 6.1. If this maximum overlap is less than a tolerance $\overline{\sigma}_0$, then $\mathbf{M}'$ is added to the match graph as a new node along with an edge connecting it to $\mathbf{M}$. If the maximum overlap is greater than σ_0, then the frame interval of $\mathbf{M_{max}}$ is averaged with that of $\mathbf{M}'$ and an edge is added between $\mathbf{M}$ and $\mathbf{M_{max}}$. Otherwise, if the maximum overlap is between $\overline{\sigma}_0$ and σ_0, then $\mathbf{M}'$ is discarded. Intuitively, this is done because an intermediate maximum overlap indicates that $\mathbf{M}'$ is neither a truly new match nor sufficiently similar to any existing match that it can be merged with it, and hence the safest procedure is to eliminate it. In our experience, this situation arises primarily when the distance threshold is sufficiently large that motion segments with only very rough similarity to the query are considered matches.

Appropriate values for σ_0 and $\overline{\sigma}_0$ depend on the query. In general, we have found $\sigma_0 \approx 80\%$ and $\overline{\sigma}_0 \approx 20\%$ to work well, but queries involving multiple periods of a cyclic motion (e.g, walking) require larger values of $\overline{\sigma}_0$ to allow greater overlap in distinct matches.

6.2.4 Experimental Results

We tested our match web implementation on a data set containing 37,000 frames, or a little over 10 minutes of motion sampled at 60Hz. The data was divided into thirty files ranging in length from 3s to 75s, and it included both motions where the actor performed a scripted sequence of specific moves and motions consisting of random variations of the same action. The former class of motions included picking up and putting back objects at predefined locations, walking/jogging in a spiral

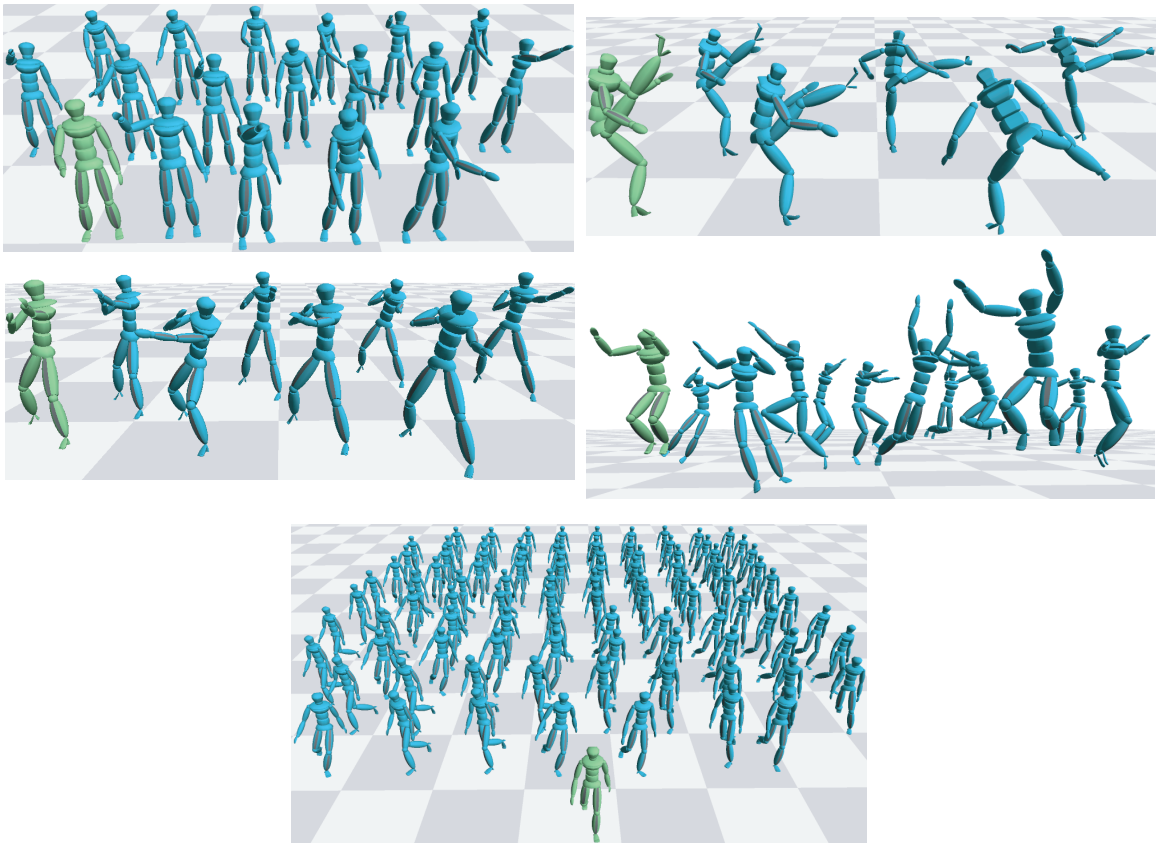

Figure 6.11: Search results for reaching, kicking, punching, jumping, and walking. Query motions are green and matches are blue.

at different speeds, stepping onto/off of platforms of various heights, and sitting down/standing up using chairs of various heights. The latter class of motions consisted of kicks, punches, cartwheels, jumping, and hopping on one foot. All experiments were ran on a machine with 1GB of memory and a 1.3GHz Athlon processor.

Figure 6.11 shows some query motions and the sets of the matches returned by our system. Videos are available at http://www.cs.wisc.edu/graphics/Gallery/Kovar/ParamMotion. In each case, the entire search process took less than half a second. Manually cropping matches from the data set with similar precision would be quite tedious, especially in the case of the walking query, where 95 matches were identified.

The remainder of this section provides details on the cost of building match webs and executing searches (in terms of both time and storage), presents results on the accuracy of the search, and briefly discusses some advantages and limitations of our approach.

6.2.4.1 Time and Storage Requirements

We initially constructed a match web for the entire data set by building a low resolution version at 10Hz and then refining it in two stages, first to 20Hz and then to 60Hz. The total computation time was 50.2 minutes. Without compression, the size of the match web on disk was 76.2MB. While this is three times the size of the original data, it nonetheless fit comfortably into main memory. Applying a standard compression algorithm (Lempel-Ziv encoding, as implemented in gzip) reduced the size by a factor of three to 24.6MB. Finding matches for a 1.5s query took on average 0.024s. Since each match is used as a new query, the time needed for a full search depends on how many matches are found — for example, finding 100 matches would take about 2.4s.

Using just the names of the data files, we next divided the data set into six categories: cartwheels, fighting, reaching, locomotion, jumping/hopping, and miscellaneous. Most individual data files, however, still contained multiple actions; for example, one "reaching" file contained six reaching motions, many walk cycles, and some motion of the actor readying himself. Separate match webs were built for each category, which took a total of 3.4 minutes and consumed 45MB of disk space without compression. Searching the reaching and locomotion match webs, which each comprised about a quarter of the data, took on average 0.006s for a 1.5s query, or about a quarter of the time needed in the unlabelled case. These results may be interpreted as follows. Building a match web requires $O(n^2)$ time for a data set of n frames, since the distance between every pair of frames must be computed (see Section 6.4 for more discussion on scalability). Hence, dividing the data set into six pieces should reduce computation time by roughly an order of magnitude. On the other hand, motions in different categories tend to be dissimilar and hence have sparse match webs, so dividing up the data set produces a more modest savings in storage than in computation time. The reduced search time stems from our search algorithm being approximately linear in the size of the data set, so smaller data sets yield proportionately faster searches.

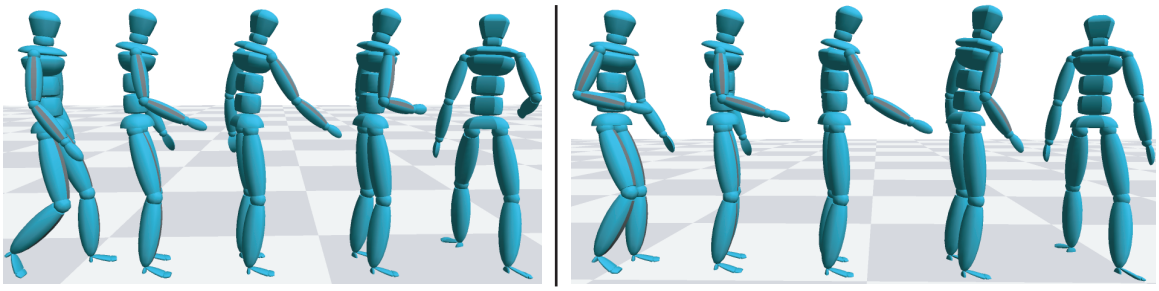

Figure 6.12: Due to their strong numerical similarity, our system confuses picking up an object (left) with putting it back to the same location (right).

6.2.4.2 Accuracy

To test the accuracy of the search results, we first restricted searches to data files in the same semantic category as the query. We entered queries of walking, jogging, jumping, hopping, punching, cartwheeling, sitting down/standing up, stepping onto/off of platforms, kicking, and picking up an object. For each query we attempted to find a distance threshold that would find all logically similar motion segments (identified manually) and nothing else. This was possible for all but the last two queries. For the kick query, we were able to find all kicks involving the same leg except for a spinning back kick. This kick was sufficiently different that it did not have strong time correspondences with any other kick (criterion 2 in Section 6.2.1). For the query of a character picking up an object from a shelf, the system returned every motion involving reaching to the shelf, but in half of the cases the character was putting the object back. A human can distinguish these motions because the reaching arm is initially hanging downwards when picking the object up and bent when putting it back (see Figure 6.12). However, this difference is sufficiently subtle relative to the rest of the motion that our system could not discern it.

We next ran the same experiments using the original, unlabelled data set. The results were identical, with two exceptions. First, more matches were returned for the walking query, since some motions not labelled as locomotion contained short segments of walking. Second, for the reaching query we found that any distance threshold large enough to return all of the data set's 17 reaching motions also included spurious matches. This is because some reaching motions were sufficiently different than the others that, in terms of our numerical similarity metrics, they

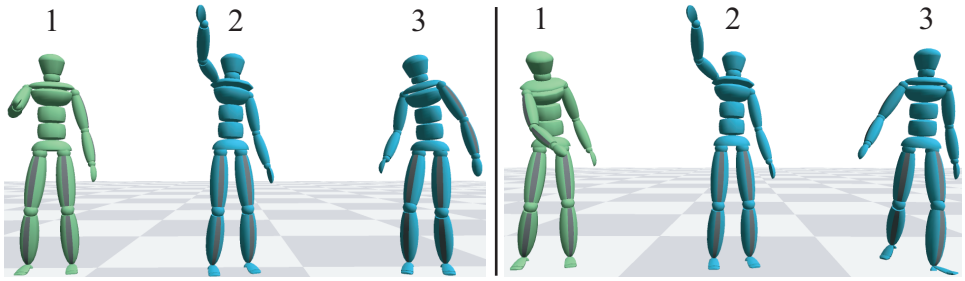

Figure 6.13: Left. Relative to a middle reach (1), a high reach's (2) numerical distance is comparable to looking over one's shoulder (3), but it is closer in terms of graph distance. **Right.** For a query of a lower-left reach (1), any threshold that returns an upper-right reach (2) as a first-tier match also returns spurious matches, such as walking (3).

were comparable to logically unrelated motions such as looking over one's shoulder (Figure 6.13). However, when sorted in order of increasing distance to the query, the first 17 matches were the true reaching motions.

6.2.4.3 Discussion

Our search algorithm defines matches as motion segments that are either close to the query or connected to it via a sequence of close intermediary matches. This allows one to find distant matches while using smaller, more reliable distance thresholds that prune unrelated motion segments. For example, given a query of someone reaching to the lower left, any distance threshold large enough for an upper-right reach to be considered close (i.e., a first-tier match) also produced many spurious matches, such as walking motions (Figure 6.13). On the other hand, using a lower threshold and multi-step search correctly identified the upper-right reach as closer to the query than any other non-reaching motion.

Nonetheless, some matches may be sufficiently far from the others that to find them one *must* use a threshold which also returns spurious matches. Since these spurious matches are used as new queries, they can lead to additional spurious matches. Two possible solutions are to incorporate semantic information, which prunes the space of candidate matches, and to sort the matches based on shortest-path distance to the query, on the assumption that spurious matches have a greater total distance than true matches. In our experiments, both of these solutions were successful.

If one logical match is very far from the others, then it may not be possible to generate clear time correspondences with any other match (that is, the second criterion of Section 6.2.1 would fail). In this case, that match will not be part of the match web and cannot be found by our system. We believe this is reasonable because our ultimate goal is to blend the matches to create parameterized motions, and a match that is this different from the others is unlikely to yield successful blends.

6.3 Building Parameterizations

Once example motions have been collected, they can be interpolated to create new motions using the methods of Chapter 5. While the synthesized motion can be controlled simply by varying the interpolation weights, in general these weights have no simple relationship to motion features. To provide more intuitive control, we parameterize the space of interpolations according to a user-supplied parameterization function $\mathbf{f}$ that computes relevant properties of the initial query motion $\mathbf{M_q}$. For the following discussion, we assume that the inputs to $\mathbf{f}$ are joint positions and orientations. This allows a wide range of properties to serve as the basis for the parameterization, including the location of an end effector at a point in time; the average, minimum, or maximum angular velocity of a joint; and features of aggregate quantities such as the center of mass. We assume that the parameterization function accounts for all meaningful differences between the input motions, and in particular we assume that motions with identical parameters could be used interchangeably. For example, if the examples are reaching motions and $\mathbf{f}$ provides the wrist location at a reach's apex, then it is implicitly assumed that the character performs each individual reach in a similar manner (e.g., it does not sometimes snatch objects and at other times casually pick them up).

Abstractly, given a set of input motions, $\mathbf{f}$ maps interpolation weights $\mathbf{w}$ to a parameter vector $\mathbf{p}$. Our goal is to invert this function: given a set of parameters, we want interpolation weights that produce the corresponding motion. Unfortunately, in general $\mathbf{f^{-1}}$ has no closed form representation. Moreover, since the number of examples is almost always greater than the dimensionality of the parameter space, $\mathbf{f}$ is a many-to-one function, and thus computing $\mathbf{f^{-1}}$ is an ill-posed

problem in the sense that it has no unique solution. In light of this, we use scattered data interpolation to construct a well-defined approximate representation of $\mathbf{f^{-1}}$ from a set of discrete samples $\{(\mathbf{p_1}, \mathbf{w_1}), \ldots, (\mathbf{p_m}, \mathbf{w_m})\}$. If there are n example motions, then initially there are n of these samples: the i^{th} example $\mathbf{M_i}$ is associated with a sample $(\mathbf{f}(\mathbf{M_i}), \delta_{\mathbf{ij}})$, where the i^{th} component of $\delta_{\mathbf{ij}}$ is 1 and the other components are 0. Given a set of target parameters $\widetilde{\mathbf{p}}$, the scattered data interpolation algorithm identifies nearby parameter samples $\{\mathbf{p_{i_1}} \ldots \mathbf{p_{i_k}}\}$ and averages the weights associated with each $\mathbf{p_{i_j}}$ according to the distance between $\mathbf{p_{i_j}}$ and $\widetilde{\mathbf{p}}$. Note that this makes $\mathbf{f^{-1}}$ well-posed by restricting the space of possible weights based on the samples $(\mathbf{p_i}, \mathbf{w_i})$.

The approximation of $\mathbf{f^{-1}}$ becomes better as the parameter space is sampled more densely, in the sense that the generated interpolation weights will produce motions that more accurately possess the desired parameters. Indeed, in the limit of arbitrarily many samples we effectively have a lookup table that directly maps motion parameters to interpolation weights. Uniform sampling also leads to better parameterizations, because clusters of samples can skew the approximation. In the extreme case where one set of sampled parameters $\mathbf{p_i}$ is duplicated several times, it will influence the final set of weights as if it were much closer to the target parameters $\widetilde{\mathbf{p}}$ than it truly is. Unfortunately, the sampling of parameter space provided by the example motions is not guaranteed to be either dense or uniform, and hence the approximation of $\mathbf{f^{-1}}$ may be inaccurate (Figure 6.2). To correct this, we synthesize interpolations of the examples to create additional samples of $\mathbf{f^{-1}}$, ensuring that the parameter space is sampled densely and uniformly.

The remainder of this section provides details on building parameterizations. The following issues are covered:

1. **Motion registration.** The methods of Chapter 5 are generalized so registration curves can be built from match graphs.

2. **Sampling strategy.** To sample interpolation weights when there are many example motions, our algorithm finds subsets of these examples that are nearby in parameter space. Weights are then randomly generated for this subset in a manner that explicitly limits the degree to which the data can be extrapolated.

3. **Fast interpolation that preserves constraints.** We use a k-nearest-neighbors technique that is efficient for large example sets and respects constraints on interpolation weights.

After discussing these matters, we conclude with some example results.

6.3.1 Registration

A registration curve is built for the example motions in a manner similar to what was used in Chapter 5: using timewarp and alignment curves between pairs of motions, "global" timewarp and alignment curves are built that encompass all of the motions. The procedure for determining constraint matches is then the same as before. The algorithm starts by using Dijkstra's algorithm to identify the shortest path from the query $\mathbf{M_q}$ to every other motion in the match graph. Any edge that is not on one of these shortest paths is discarded. For each remaining edge, the methods of Chapter 5 are used to build timewarp and alignment curves between the motions at the incident nodes. The timewarp curves are constructed so the initial and final frames of both motions are exactly interpolated. Now, given a frame of $\mathbf{M_q}$, the corresponding frame in any other motion can be found by walking down the shortest path on the match graph, using the timewarp curve at each edge to convert the frame of the current motion to the corresponding frame of the next motion. In this manner for any frame $\mathbf{M}_q(t_q)$ one can generate a frame correspondence $(\mathbf{M_q}(t_q), \mathbf{M_2}(t_2), \ldots, \mathbf{M_n}(t_n))$, where without loss of generality we have identified $\mathbf{M_q}$ with $\mathbf{M_1}$. To generate a timewarp curve for the entire match graph, frames of $\mathbf{M_q}$ are sampled to generate a dense set of these frame correspondences, and an n-dimensional, strictly increasing, endpoint interpolating quadratic B-spline is fit to them as in Chapter 5. The procedure for building the alignment curve is analogous: for any frame of $\mathbf{M_q}$, a transformation that aligns it with the corresponding frame of $\mathbf{M_i}$ can be found from the match graph, and so sets of these transformations are sampled and fit with a B-spline.

If any part of the parameterization function involves the skeletal configuration on a specific frame $\mathbf{M_q}(t_0)$, then this frame index t_0 is converted to the corresponding index u_0 on the timewarp curve. This allows the parameterization function to be computed for any other example motion and for any interpolation of the examples.

6.3.2 Sampling

While the goal is to produce a dense sampling of parameter space, parameter samples cannot be created directly; instead, they must be generated indirectly by sampling interpolation weights. Densely sampling these weights is infeasible for large example sets, because the dimensionality of the interpolation weight space is proportional to the number of examples, and so the number of samples needed to achieve a given sampling density grows exponentially with the number of examples. Moreover, such a sampling would be redundant because $\mathbf{f}$ is a many-to-one function — the same region of parameter space would be covered repeatedly by different sets of interpolation weights.

These difficulties can be avoided by limiting interpolations to subsets of examples that are nearby in parameter space. Intuitively, the goal of sampling is to fill in the gaps in parameter space, and the most natural strategy for filling any given gap is to combine the closest example motions. For example, imagine the examples are various reaching motions and we want to sample interpolated motions that reach near chest height. This could theoretically be done by combining motions that reach to the ground with ones that involve standing on tiptoes, but it is more sensible to instead combine example motions that already reach to points near the desired region.

This intuition can be turned into an algorithm as follows. First, the parameters of each example motion are computed. To approximate the accessible region of parameter space, a bounding box is generated for these parameters and expanded in each dimension by a fixed percentage about the central value (20% in our implementation). Points within this region are then randomly sampled, and for each point the algorithm locates the $d + 1$ example motions with the closest parameters, where d is the dimensionality of the parameter space. This number of neighbors is used because it is the minimum necessary to form a volume in parameter space. The weight of every other motion is set to zero, and a random set of weights is generated for the neighbors under the restriction that these weights must be nearly convex, or *valid*. Mathematically, this is defined as

$$-\delta \leq w_i \leq 1 + \delta \tag{6.2}$$

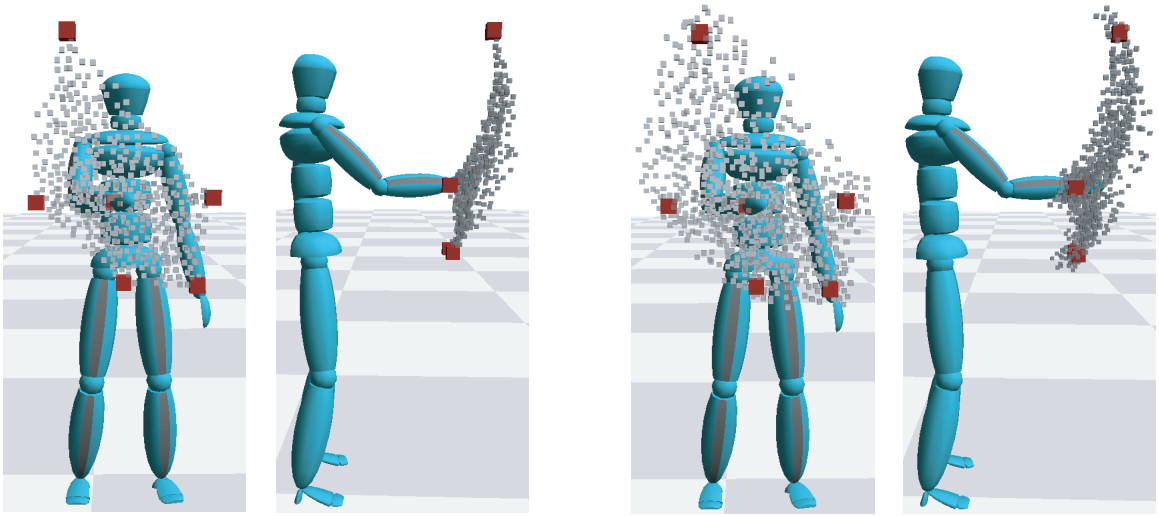

Figure 6.14: Samplings of parameter space for a set of reaching motions, using different values of δ.

and

$$\sum_i w_i = 1, \tag{6.3}$$

where δ controls the allowable degree of extrapolation. In particular, when $\delta = 0$ there is no extrapolation in the sense that the interpolation weights are strictly convex. A random set of valid interpolation weights can be calculated as follows. Let S be the sum of all weights that have been assigned values; initially $S = 0$. While unassigned weights exists, randomly select one of these weights w_i. If w_i is the last unassigned weight, set it to $1 - S$. Otherwise randomly assign it a value from the interval $[\max(-\delta, -\delta - S), \min(1 + \delta, 1 + \delta - S)]$. This ensures that both w_i and $S + w_i$ have a value in the range $[-\delta, 1 + \delta]$. The latter condition is important because it guarantees that the final weight is valid.

To prevent parameter samples from being too close, for each new sample the closest existing sample is found. If these samples are within a threshold distance of each other, then the new sample is discarded. In our experiments, which focused on joint positions, the threshold was half an inch.

By selecting different values for δ, a user can control the tradeoff between flexible synthesis (motions are generated from a larger space of possible interpolation weights) and quality guarantees (synthesized motions stay "near" the original data). Figure 6.14 shows samplings of parameter

space generated with different values for δ. In general, we have found that δ can be set to .1–.15 without much sacrifice of motion quality.

6.3.3 Interpolation

Given a new set of parameters $\widetilde{\mathbf{p}}$, a k-nearest-neighbors algorithm is used to find interpolation weights $\widetilde{\mathbf{w}}$ that produce those parameters. Let the k nearest neighbors be $\mathbf{p_1}, \ldots, \mathbf{p_k}$, in order of increasing distance, and let $\mathbf{w_i}$ be the interpolation weights associated with $\mathbf{p_i}$. $\widetilde{\mathbf{w}}$ is approximated as

$$\widetilde{\mathbf{w}} = \sum_{i=1}^{k} \alpha_i \mathbf{w_i}. \tag{6.4}$$

Following Allen et al [3], each α_i is initially assigned the value

$$\alpha_i = \frac{1}{D(\widetilde{\mathbf{p}}, \mathbf{p_i})} - \frac{1}{D(\widetilde{\mathbf{p}}, \mathbf{p_k})}, \tag{6.5}$$

where D computes the distance between two parameters (Euclidean distance in our implementation). These weights are then normalized so they sum to 1. Since the α_i are nonnegative and sum to 1, $\widetilde{\mathbf{w}}$ is inside the convex hull of the $\mathbf{w_i}$, and therefore $\widetilde{\mathbf{w}}$ satisfies the conditions in Equations 6.2 and 6.3. This ensures that any parameters specified by the user will produce a motion within the space of valid interpolations. In particular, parameters that are not attainable are effectively projected onto the accessible region of parameter space.

As a result of our sampling procedure, at most $d + 1$ elements of each $\mathbf{w_i}$ are nonzero. This implies that $\widetilde{\mathbf{w}}$ will in the worst case have $k(d+1)$ nonzero weights, and in practice there are fewer because nearby parameter samples tend to have the same set of nonzero weights. Since motions with zero weight can be ignored when creating a blend, the asymptotic run time of our algorithm is independent of the number of examples. This analysis neglects the cost of finding the k nearest neighbors, but we have found that even a brute force nearest neighbor calculation is negligible relative to the cost of interpolating motions.

Motion	# Examples	Time To Build	Size of Samples
reach	6	6.7s	4.5%
walk	96	4.6s	2.9%
kick	4	1.8s	1.4%
sit	2	3.9s	0.4%
step up	4	2.5s	0.4%
punch	7	1.7s	4.9%
hop	4	3.3s	1.8%
cartwheel	11	6.5s	8.2%

Table 6.1: Data about the parameterized motions built in our experiments. The storage needed for parameter samples is given as a percentage of the motion data's size.

6.3.4 Results and Applications

We have implemented the above algorithms and used them to create a variety of parameterized motions; see Table 6.1 for a summary. In each case we generated a thousand parameter samples using the method described in Section 6.3.2. The time needed to generate these samples varied from 1.7s to 6.7s, and after eliminating redundant samples the storage cost was 8.2% of the example motion data in the worst case and 3.2% on average. In all of our experiments new motions could be synthesized in real time, and these motions matched the user's target parameters to within visual tolerance as long as they were within the accessible region of parameter space. We set $k = 12$ when performing nearest neighbor interpolation. Videos are available at http://www.cs.wisc.edu/graphics/Gallery/Kovar/ParamMotion.

In addition to providing more accurate parameterizations, the sampled parameters can be used to visualize the range of synthesizable motions. Figure 6.15 shows some cases where this can be accomplished simply by drawing markers at the location of each parameter sample. Another application of our methods is automatic removal of redundant example motions, which can reduce the parameterized motion's memory footprint. For each example motion, we compute its parameters and see if they can be reproduced within a user-specified tolerance by interpolating nearby

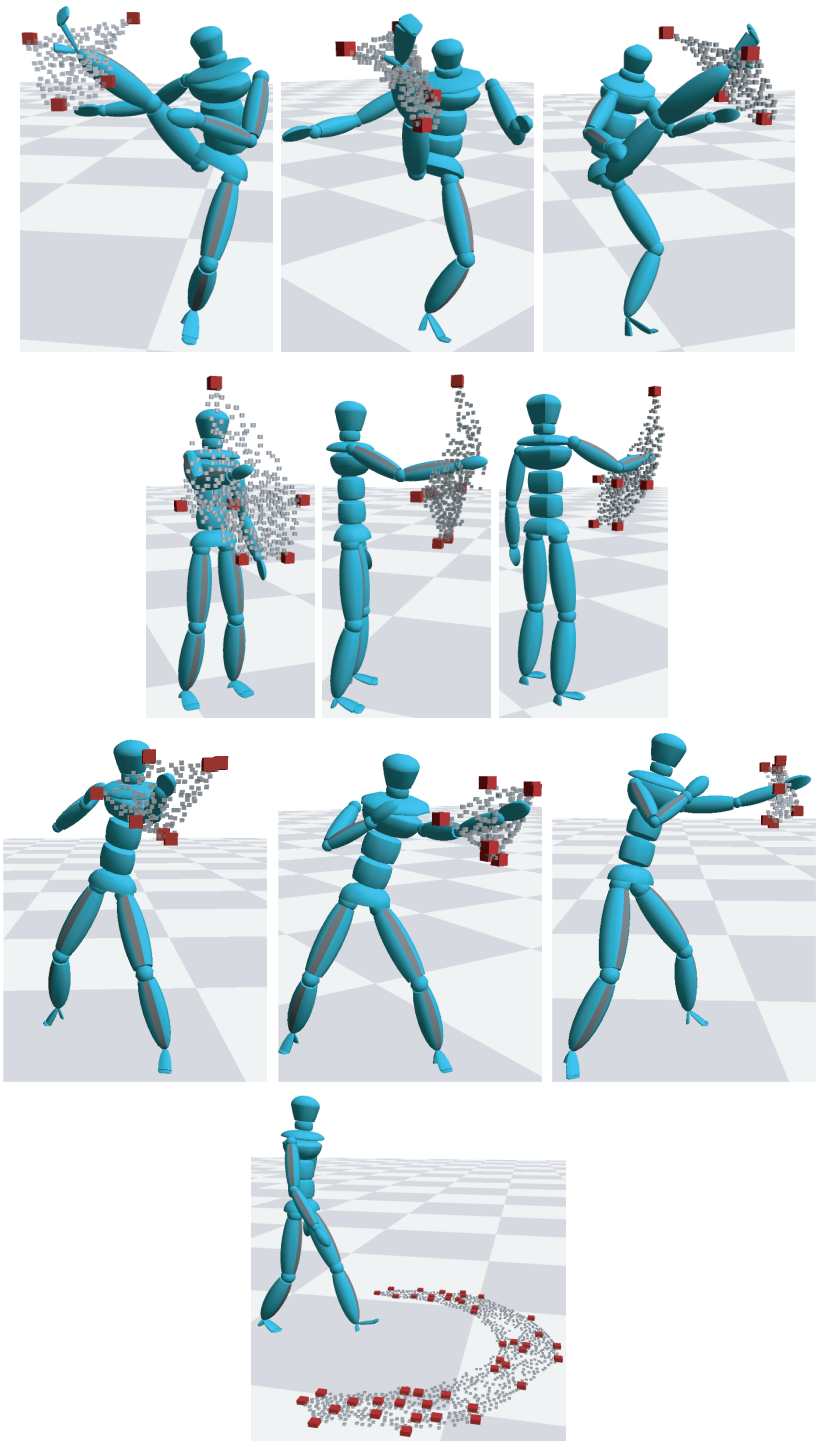

Figure 6.15: In order from top to bottom: visualization of the accessible locations for the ankle of a kick, the wrist of a reach, the wrist of a punch, and the final position of a walk cycle. Large red cubes show parameters of example motions; small grey cubes are sampled parameters.

examples. If so, it is discarded. For our parameterized walk, we removed all example motions where the final root location could be reproduced to within a quarter inch, reducing the number of motions from 96 to 46. The parameterized walk also shows the scalability of our scattered data interpolation method: including all 46 example motions in motion interpolation takes an order of magnitude longer than using our algorithm.

The automation provided by our system makes it feasible to experiment with unusual parameterized motions. Starting with 34s of cartwheel data, we built a parameterized motion where the user could control the final position of a sequence of cartwheels. The total amount of time needed to build the match web, identify a particular cartwheel, execute a search for other cartwheels, and generate parameter samples was less than thirty seconds.

6.4 Discussion

This chapter has presented automated methods for extracting logically related motions from a data set and converting them into an intuitively parameterized space of motions. One contribution of this work is a novel search method that uses numerically similar matches as intermediaries to find more distant matches, together with a precomputed representation of all possibly similar motion segments that makes this approach efficient. A second contribution is an automatic procedure for parameterizing a space of interpolations according to user-specified motion features. This algorithm samples interpolations to build an accurate approximation of the map from motion parameters to interpolation weights, and it incorporates a scalable scattered data interpolation method that ensures interpolation weights have reasonable values.

We conclude with a brief discussion of the scalability and generality of our methods.

6.4.1 Scalability

For our test data set, which is large relative to what is commonly used in current research, the time and space costs for building and storing a match web were quite manageable. However, a match web for a data set of n frames takes $O(n^2)$ time to construct and $O(n^2)$ space to store. The

time costs can be mitigated by using a multi-resolution construction method (as discussed in Section 6.2.2) and by computing match webs for different pairs of motions in parallel, and storage can be reduced through standard compression algorithms. Nonetheless, for massive databases it will not be feasible to build match webs for every pair of motions. A simple solution is to partition the database into independent modules based on semantic content and compute match webs separately for these modules. Since a very large database will contain many unrelated motions, such a division would be natural and is likely to already be reflected in the organization of the data files. The development of alternatives to match webs that are asymptotically more efficient yet have similar performance characteristics is left for future work.

Because our search algorithm tests every path in the match web that is contained within the rows spanned by the query, in general the time needed to execute a search is linear in the size of the database. However, we believe the paths in the match web may be organized into a hierarchical data structure that allows more efficient pruning of paths that exceed the distance threshold. Developing such a data structure is also left for future work.

While our parameterization algorithms apply in theory to parameter spaces of arbitrary dimensionality, in practice they are limited by the quantity of available data. Roughly speaking, we need at least enough examples to cover every combination of minimum/maximum values for individual parameters (the "corners" of the space), and more examples typically provide higher quality results. The number of necessary example motions is hence exponential in the number of parameters. This is a fundamental limitation of scattered data interpolation. In future work we intend to explore methods for easing the data requirements by identifying motion properties that are, for the purposes of sampling, uncorrelated.

6.4.2 Generality

While we have focused on parameterization functions involving joint positions and orientations, our methods allow parameterizations based on abstract properties like mood. For qualitative features like these where accuracy is less meaningful, we can simply skip the sampling step of Section 6.3.2 and apply scattered data interpolation directly to the example motions.

Motion sets found by our search engine are not guaranteed to be blendable. For example, one of our query motions consisted of a character stepping toward a shelf, picking up an object, and walking away. While our system correctly identified the eight other picking-up actions in the database, in three of these the initial step was with the wrong foot, and so these motions had to be discarded. More generally, the only reliable way of determining whether motions can be successfully blended is to create and look at specific blends. In light of this, one of the primary advantages of our system is that it greatly speeds and simplifies the process of experimentation.