DOI: 10.1111/j.1467-8659.2009.01626.x

EUROGRAPHICS 2010 / T. Akenine-Möller and M. Zwicker
(Guest Editors)

Volume 29 (2010), Number 2

Bidirectional Search for Interactive Motion Synthesis

Wan-Yen Lo[†1] and Matthias Zwicker[1,2]

[1]University of California, San Diego [2]Universität Bern

Abstract

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

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

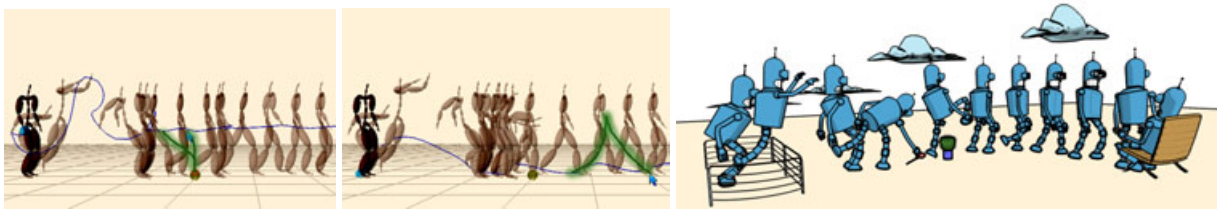
1. Introduction

Realistic motion synthesis is a core topic in computer animation. Many of the most successful techniques are based on recombining motion fragments using motion graphs. By reusing and concatenating motion capture data, these methods can generate complex and natural-looking motions. Most systems allow a user to provide a number of constraints to specify the desired motion. Such constraints are formulated as a cost function. Motion synthesis is cast as a search problem for a path through the motion graph that minimizes the total cost. The search complexity for an optimal or near-optimal solution, however, is exponential to the connectivity of the graph and the length of the desired motion sequence. Applying these techniques for interactive applications is thus challenging.

In this paper we present a novel approach to apply bidirectional search for motion synthesis using motion graphs. Our technique improves the search efficiency while preserving the search quality. The key idea is to expand two search trees simultaneously, one from the beginning and one from the end of the motion sequence. A core component of our bidirectional search algorithm is a novel technique to efficiently merge the two search trees to obtain one continuous motion sequence. We dynamically divide the search space to ensure that the two trees have similar height. We cache partial paths using hash tables so that we can merge nearby nodes efficiently. This approach allows us to reduce the required search depth by a factor of almost two, which leads to a significant performance improvement. More applications can thus be improved to achieve interactive performance. Since A* search is optimally efficient and considered the state-of-the-art technique for near-optimal motion synthesis, we demonstrate our approach using bidirectional A* search in this paper.

To demonstrate the efficiency of our approach, we present an intuitive sketch interface for interactive motion synthesis. Today's video games include characters with a rich set of behaviors. It becomes increasingly difficult to control varied motions with conventional interfaces, such as joysticks, game pads, mice, or keyboards, since they often require memorizing awkward keystroke combinations or gestures. With existing sketching systems, users need to memorize a list of mappings between stroke patterns and motions. Our system, on the other hand, does not require any memorization but allows users to control the motion intuitively by sketching a trajectory on a specified body part. Given a stroke input, we search and compose a sequence of motions whose projected trajectory best matches the input, as shown in Figure 1. With our bidirectional A* search, we achieve a significant speedup over traditional near-optimal techniques, and can synthesize motions in seconds with a

† walo@ucsd.edu



Edit hand trajectory to grasp object | Edit foot trajectory to step over | Final motion

Figure 1: *An editing session with our sketch interface. The user first edits the trajectory of the character's hand to pick up an object. The original trajectory is marked in blue, and the user edit in green. Our bidirectional search algorithm provides immediate feedback to visualize the edited motion. The user next selects the character's foot and edits its trajectory to step over an obstacle. Our system synthesizes the final motion, shown on the right, satisfying all the constraints in a near-optimal fashion.*

moderate-sized graph. In summary, this paper makes the following contributions:

- We describe a bidirectional search algorithm to improve the search efficiency for near-optimal motion synthesis. We present a novel strategy to dynamically divide the search space evenly and to merge two search trees efficiently.
- We develop a sketching interface for interactive motion synthesis. It allows a user to intuitively generate complex motions in an interactive environment.

2. Related Work

Motion Synthesis with Motion graphs. Motion graphs and related approaches have proven to be quite successful as a tool to synthesize natural and complex motions [KGP02, AF02, AFO03, CLS03]. These techniques organize fragments of captured motion in a graph structure, and generate new motions by building walks on the graph. Over the past few years, a number of approaches have been developed to combine interpolation techniques with motion graphs to provide smoother transitions and more accurate control [SO06, HG07, SH07, BCvdPP08].

Given user constraints, Kovar et al. [KGP02] use depth-first search to obtain graph walks. They improve the efficiency of naïve depth-first search using a branch-and-bound strategy and incremental search. Lee et al. use greedy best-first search and traverse only a fixed number of frames to maintain a constant rate of motion [LCR*02]. Arikan et al. [AF02] developed a hierarchical, randomized search strategy. In their follow up work, Arikan et al. [AFO03] presented a method that allows users to specify constraints as intuitive annotations. This method uses a dynamic programming approach and coarse-to-fine refinement to search for the motion sequence that satisfies the user constraints. Choi et al. [CLS03] adapt probabilistic roadmaps (PRMs) to search in a motion graph, which is a popular algorithm in path planning. However, the quality of the synthesized motion is not guaranteed to be near-optimal with any of these methods.

Safonova and Hodgins [SH07] use interpolation between motion graphs to increase the range and accuracy of output motions. They emphasize the benefits of performing optimal search and employ anytime A* to achieve this goal. Their approach, however, is not well-suited for interactive applications, unless the optimality bound is increased significantly. Lau and Kuffner [LK05] also use A* search. They build a finite state machine of character behaviors that reduces the search space and the computational cost of global search. They also show how pre-computation can be leveraged to further increase runtime performance [LK06]. Their state machines, however, are limited to contain at most a few dozen states to achieve interactivity.

Bidirectional search. Bidirectional search has been explored in artificial intelligence and path planning [RN03, LaV06]. The critical step is to design a mechanism to merge the two partial searches, which needs to be custom tailored for the given search problem. Without proper designs, bidirectional search might have worse performance than unidirectional search [Poh71, Kwa89], because it is necessary to prevent two search frontiers from passing each other. Kandl and Kainz [KK97] presented one of the first successful approaches to bidirectional search and demonstrated that it can be more efficient than unidirectional search. They run an A* algorithm and change the search direction only once, so the two searches might not be well balanced. They hash the frontier nodes in the first search to cut off some branches in the reverse search meeting the opposite frontier. In our work, we exploit properties of the state space in motion synthesis to define a cut that divides the search. This allows us to dynamically balance the two searches to gain more speedup.

The Rapidly-Exploring Random Tree (RRT) planner is popular in motion planning [LK00, LaV06]. The basis of the RRT method is the incremental construction of search trees that attempt to rapidly and uniformly explore the state space. This can be considered as a Monte-Carlo way of bias-

ing the search into the largest Voronoi regions. Kuffner and LaValle [KL00] described a bidirectional extension of RRT. The objective of RRT related search, however, is to find a feasible path quickly. It does not attempt to find an optimal or even close to optimal path.

In computer animation, Sung et al. [SKG05] developed a bidirectional variation of probabilistic roadmaps (PRM) for efficient searching in motion graphs. Their PRM search strategy is based on randomized greedy search from both the beginning and the end of the motion sequence. They construct a continuous output motion by merging the two paths at their closest point. This strategy does not decrease search depth. Shapiro et al. [SKF07] use the bidirectional RRT algorithm to modify motions to obey constraints such as being collision-free. Their search algorithm returns only a feasible result, while we focus on finding a near-optimal solution efficiently.

User interfaces for motion control/synthesis. Intuitive user interfaces are essential for effective motion synthesis applications. Shiratori and Hodgins [SH08] provided an extensive overview of recent techniques that employ a variety of input devices. Our approach is most similar to previous techniques that use sketching. In some early works [LCR*02, KGP02], the character is able to follow the path sketched by the user. Other sketch-based approaches allow a user to generate a wider variety of motions by drawing simple strokes [TBvdP04, Osh05]. These techniques, however, depend on predefined dictionaries that map different stroke patterns to motions. Our system is more intuitive in that the users are not required to memorize any control pattern, but can sketch the trajectory in an intuitive way.

3. Search in Motion Graphs

The optimal path through a motion graph that satisfies user constraints often leads to the most intuitive results [SH07]. However, finding such a path can be computationally expensive. This makes it challenging to perform optimal or near-optimal search in interactive applications. The main idea of our approach is to run two searches simultaneously from the start state and from the end state of the final motion to boost the search performance. The main benefit of this approach is that it reduces the maximum search depth almost by a factor of two. This reduction in search depth causes drastic gain not only in computing but also memory efficiency, for we reduce the number of nodes that are expanded and stored. We demonstrate the benefits of the bidirectional strategy by applying it to existing search algorithms. We focus on A* search, since it is provably the most efficient among all the optimal search algorithms [RN03].

Before going into the details of the search algorithm we provide some notations. Each node in the motion graph is defined as $G = (I)$, where I is an index of the pose in the motion capture database. The links in the motion graph are short motion clips that transition between the poses defined on both sides. Each node in the *search tree* is a tuple $N = (X, f(X), N')$, consisting of the current state $X \in \mathcal{S}$ of the character, a cost function $f(X)$ associated with the state, and the parent N' of this node in the search tree. The *state space*, $\mathcal{S}$, is defined as the range of all possible states. Whenever a node in the motion graph is unrolled into the environment, new nodes are added to the search tree and their states X are determined based on the motion clips (i.e., the links in the motion graph) that are traversed. The state may include any information that can be derived from unrolling the motion graph, such as global position and orientation. We refer readers to Lau and Kuffner's work [LK05] for more details about the implementation of unrolling a motion graph into the environment to form a search tree.

3.1. Standard A* Search

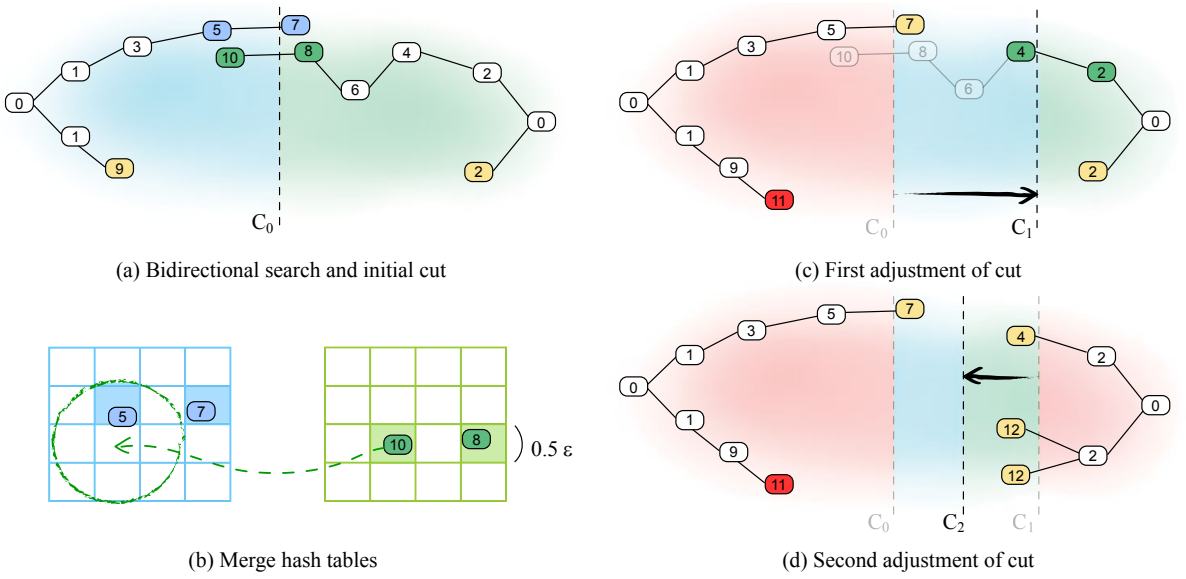
A* search is a best-first graph search algorithm that finds the lowest-cost path from a given initial node to a goal node [RN03]. The main feature of A* search is that it uses a cost function to determine the order in which the search visits nodes in the graph. The algorithm evaluates the cost of a node $N = (X, f(X), N')$ by combining the actual cost to reach the node, $g(X)$, and a *heuristic function* $h(X)$ that estimates the cost to get from the node to the goal. Hence, $f(X) = g(X) + h(X)$. We can interpret $f(X)$ as an estimated cost of the cheapest solution that passes through N.

Starting with an initial node, A* search maintains a priority queue of nodes to be traversed, where the node N with the lowest estimated cost $f(X)$ has the highest priority. At each step, the node with the highest priority is expanded and removed from the queue, the costs of its neighbors are updated accordingly, and the neighbors are inserted into the priority queue. The algorithm continues until the lowest f value in the priority queue exceeds the cost of the best solution so far, or until the queue is empty.

The solution of A* search is guaranteed to be optimal if the heuristic function h is *admissible*, that is, if $h(X)$ never overestimates the cost to reach the goal. Further, A* search is appealing because it is optimally efficient: given a graph and a heuristic function, no other optimal algorithm is guaranteed to expand fewer nodes than A* [RN03]. In motion synthesis, A* search has been successfully adopted to search for optimal or near-optimal motions in motion graphs or finite state machines [LK05, SH07].

3.2. Bidirectional A* Search for Motion Synthesis

We propose a bidirectional algorithm that can make existing search algorithms, especially A* and its variants, more efficient. If unidirectional search has a branching factor, i.e., an average number of successors of any node, of b, then the search space for finding a path of length d is on the order of $O(b^d)$. A bidirectional algorithm, however, can reduce the



(a) Bidirectional search and initial cut

(c) First adjustment of cut

(b) Merge hash tables

(d) Second adjustment of cut

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

size of search space to $O(b^{\frac{d}{2}})$. In our application, this translates into significant speedups as we will show in Section 5.1 and Section 5.2.

The performance of bidirectional search strongly depends on two factors: a **stopping criteria** that determines when to stop the searches from each direction, and a **merge query** that efficiently evaluates potential merge points between the two searches. The performance may be even worse than unidirectional A* if they are not properly designed. The core component of our bidirectional search algorithm is a novel strategy to design these factors for motion synthesis. We address the first issue by defining and dynamically adjusting a global *cut*, which determines where the search is stopped. To solve the second issue, we introduce an efficient data structure that we call a *merge hash table* (MHT).

The Cut. The search starts by alternately expanding two search trees from both ends by maintaining two priority queues. We expand both trees until they reach the *cut*. The idea is that the cut halves the *search space*, the set of all possible solutions, so the work required to expand both trees is well balanced. When a node passes the cut, we stop its further expansion, since this side will be explored by the other search tree. At the beginning of the search, without sufficient information about how the search will proceed, it is difficult to know where to place the cut. We thus place an initial cut between the start and end states as a hyperplane that splits the state space $\mathcal{S}$ into two equal halves, as shown in Figure 2a.

The performance of bidirectional A* largely depends on where the cut is placed, since this determines how the search space is split. The initial cut, estimated in the state space, may not divide the search space evenly, because search depth is not directly proportional to distances in state space. For example, some arcs in the search tree, such as the jumping motion, span a longer distance in state space than others, such as a small step. Hence, one tree may surpass the cut and terminate much earlier than the other. It is also possible that one search converges more quickly to the optimal solution than the other. Unfortunately, after one tree's termination the search behavior resembles unidirectional A*, so we would like to keep both trees growing for as long as possible.

We propose an algorithm to adjust the cut dynamically so that it eventually converges to the halfway cut in the search space. If one of the trees terminates before the other, we

move the cut toward the other tree so that it splits the remaining, not yet fully explored space into two equal halves. This procedure is iterated every time one tree stops growing (all the frontier nodes either pass the cut or have costs exceeding the current minimal cost). We repeat this process until both trees terminate, or until there is not enough space remaining to make the adjustment. Two iterations of dynamic cut adjustment are illustrated in Figure 2c and Figure 2d.

Merging two Search Trees. When a node passes the cut, we need to detect whether the current path can be merged with any existing paths from the opposite search. To this end we record the following state attributes with each node in the search tree: an index G to the corresponding node in the motion graph, and the character's position x, z, and orientation θ in a global coordinate system. We say that two paths p_1 and p_2 are *mergeable* if they contain any pair of nodes N_1 and N_2 that are close enough in state space. More precisely, they are mergeable if $\exists N_1 \in p_1$, $N_2 \in p_2$ such that $G_1 = G_2$, $|x_1 - x_2| < \varepsilon_x$, $|z_1 - z_2| < \varepsilon_z$, and $|\theta_1 - \theta_2| < \varepsilon_\theta$, where $\varepsilon = (\varepsilon_x, \varepsilon_z, \varepsilon_\theta)$ is the tolerable deviation in (x, z, θ) between two states. Instead of evaluating all pairs of nodes for merging, which takes linear-time computation, we would like to perform the query efficiently.

To achieve efficient merging, we maintain *merge hash tables* (MHT) individually for each search tree. Each MHT records a set of candidate nodes for merging from each side. The main idea is to *quantize* the state space into a grid of cells, and to use the quantized state of each node to compute the key for hashing. Pairs of nodes that can be merged will be guaranteed to fall into neighboring cells. The hash table allows us to retrieve nodes in neighboring cells in constant time while storing the grid efficiently. This approach is a variation of spatial hashing [GG98], which has been used in graphics for example for collision detection [THM*03, LH06].

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

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

Every time we adjust the cut, we must update the MHTs along with the search queues. To do the update efficiently, we maintain a Fibonacci heap for each search tree respectively to order the nodes according to their distance to the search root in state space. Given a cut, this allows us to extract all nodes beyond the cut efficiently (the amortized running time is $O(m \log n)$, where m is the number of nodes beyond the cut, and n is the number of nodes in the tree). More specifically, we update the search queues and MHTs as follows: First, insert every pair of successive nodes crossing the new cut into the MHT; second, mark all nodes beyond these pairs as obsolete; third, insert all the other frontiers into the search queue if they are not already in it and if their costs are smaller than the current minimal total cost. We illustrate the updated status after cut adjustments in Figure 2c and Figure 2d.

4. Intuitive Motion Control with Strokes

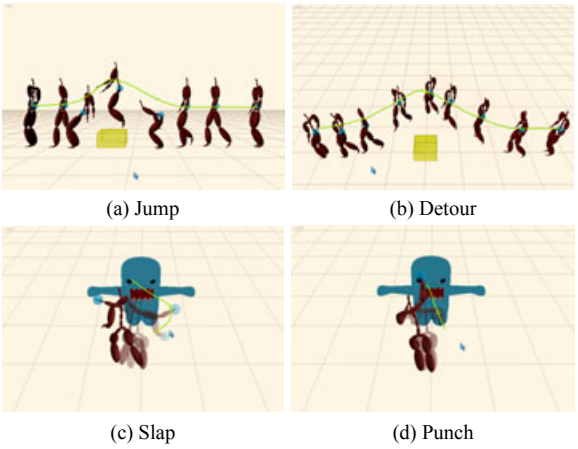
The basic idea of our system is to allow the user to specify a motion by sketching a desired trajectory for any controllable body part. Then we generate a motion whose trajectory of the specified body part best resembles the input stroke under the current viewing perspective. Our system is intuitive to use in that similar strokes may represent different motions under different viewing perspectives (Figure 3a and b), and different stroke trajectories can predictably generate different motions (Figure 3c and d) even with the same start and end conditions. These features are infeasible with gesture-based sketching systems [TBvdP04, Osh05].

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

4.1. Input Stroke Analysis

The input strokes convey the user's intentions for a motion in several ways. First, the shape of the stroke represents the desired motion trajectory under the current viewing perspective. Second, the user can draw strokes at different speeds to control the speed of the final motion. At last, the intersection of the stroke and the objects in the environment represents the character's contact with the environment. We next describe different cost functions that capture these constraints.

Distance Cost. The distance cost measures the deviation of the synthesized motion from the trajectory specified by



(a) Jump (b) Detour

(c) Slap (d) Punch

Figure 3: *Our system allows users to intuitively generate motions using a simple stroke interface. Similar strokes under different situations can lead to different motions.*

the input stroke. We use the technique proposed by Hoff et al. [HKL*99] to generate a *distance function* to the input stroke. The distance is measured in the image plane where the user draws the strokes. We sample the input stroke uniformly in time, and for each pixel c on the image plane we store its distance to the closest sample point. We denote the distance function by $D(c) : \mathbf{R}^2 \rightarrow \mathbf{R}$, which enables constant time look-up of distances. Along with the distance, we also use $T(c) : \mathbf{R}^2 \rightarrow \mathbf{R}$ to keep track of the identity of the closest sample point for each pixel on the image plane. $T(c)$ provides information about the sketching speed along with the chronological flow of the stroke.

We compute the distance cost for each new node in the search tree that we explore. To obtain a robust cost that is also efficient to compute, we evaluate the distance to the stroke only at a few key frames [ACCO05]. Let us denote the state of a parent node in the search tree by X' and the state of one of its children by X. To measure the distance error of X with respect to the stroke, we project the controlled body part to the image plane at each key frame. We denote the 3D positions at the key frames by $(p_1, p_2, \ldots, p_n)$ and their projected positions on image the plane by $(c_1, c_2, \ldots, c_n)$. Our distance cost is then

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

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

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

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

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

Complete Cost. The cost function also considers the smoothness of the transition between the poses G' and G, and the physical energy required to perform the in between motion. We adopt the point cloud metric [KGP02] with weights on different joints [WB04] to compute the transition cost. The energy cost is the sum of the squared torques via inverse dynamics [SH07]. There may be more than one sequence of motions fulfilling the input stroke, but the one with the least energy usually looks more natural. Hence, the complete cost function is defined as

$$g(X') = g(X) + \gamma_\Phi \Phi(X, X') + \gamma_\Psi \Psi(X, X') + \gamma_\Delta \Delta(G, G'),$$

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

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

4.2. Bidirectional A* Search

In this section, we provide more details about applying bidirectional A* search with the stroke interface. Our objective is to find two mergeable state sequences with minimal cost, $(X_1^f, X_2^f, \ldots, X_m^f)$ and $(X_n^b, X_{n-1}^b, \ldots, X_1^b)$, from forward and backward searches respectively. X_m^f and X_n^b are mergeable states, as illustrated in Figure 4a. The state in our system is defined as $X = (G, x, z, \theta, t)$, where G, x, z, and θ have been introduced in Section 3.2, and t indicates the chronological order of the current state along the user stroke. We find this

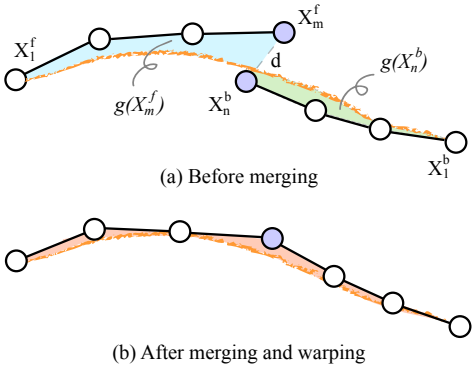


Figure 4: *(a) A possible solution of bidirectional search. The two mergeable nodes are colored in purple. (b) We warp the whole sequence while fixing the start and end states so that the final pose can fulfill the constraint more precisely.*

value by projecting the controlled body part (pelvis for example) of the character onto the image plane and looking up the identity of the closest sample point on the input stroke using the T function.

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

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

The mergeable states X_m^f and X_n^b in the result sequences may not match exactly because the motion space is discrete and the environment is quantized. If we simply concatenate the two sequences, the final state of the merged sequence will deviate from the user-specified state. Although the deviation is within the error tolerance, the result motion will be unnatural in some cases. For example in the case of grasping or kicking the character will act to the air. Therefore, we include a warping step [WP95, Gle01] so that the final states match more precisely. We illustrate the solution before and after warping in Figure 4a and Figure 4b respectively. To avoid foot sliding due to warping we apply analytical IK at run time to plant the feet [TB96, LS99]. Warping should be reduced to a minimum since it often degrades the motion quality. An advantage of near-optimal solutions is that they require less warping than motions with higher costs, which deviate further from user constraints. In the appendix, we show that we can control the optimality of the final solution by adjusting the size of quantization, ε.

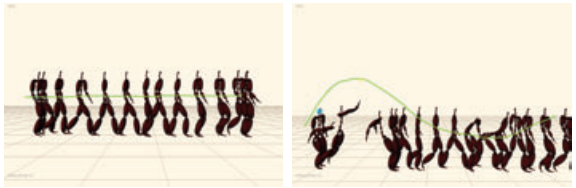
The heuristic function in our search is used to estimate the cost of getting to the goal. Our heuristic function is a product of the shortest distance from the current position to the goal, multiplied by the minimum cost function value required to travel one unit distance. We run a number of tests to empirically compute this minimum value from the motion graph data, similar to the heuristic function on the character location in [SH07].

5. Results

To produce our results, we adopt the technique proposed by Zhao and Safonova [ZS08] to construct well-connected graphs. During the construction, we interpolated the motions in a close to physically correct way [SH05]. The resulting graph is larger than a standard motion graph since it includes many interpolated poses that do not belong to the original data to achieve better quality in the synthesized motion. We compress the graph by retaining only the nodes where contact changes happen, and the links among them [SH07]. We first use the technique from Lee and his colleagues [LCR*02] to identify contacts and then modify them manually. There may be more than one path connecting each pair of contact change nodes, and we use Dijkstra's shortest path algorithm to compute and retain the optimal one. The culling step does not affect the functionality of the graph as long as users are not allowed to control the details of the motion during a period of time when the contacts are not changing [SH07].

5.1. Search Performance Comparison

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



(a) Simple walk (b) Jump and duck

(c) Walk and pick (d) Long walk

Figure 5: *Motion sequences obtained from our algorithm (without the final warping step). The character is controlled at its pelvis (a), right hand (b), right hand (c), and pelvis (d) respectively. In (d), the character jumps and lifts his leg up high once because under this viewing perspective the input stroke only provides constraint on the moving path, and these actions happen to fulfill the trajectory better.*

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

Figure 6: *Performance comparison between standard A* search (*A**), bidirectional A* search without cut adjustment (*bidirectional A**), and our proposed bidirectional A* search with cut adjustment (*bidirectional A*+*). With each example we indicate the time it takes to execute the motion. We report the computation times to find a near-optimal motion, the number of nodes expanded in the forward and backward search trees, and the speedup.*

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

We compare the search performance of standard A*, bidirectional A*, and bidirectional A* with dynamic cut adjustment. In all the test cases presented in this section, we use the same graph, cost function, and heuristic function. The synthesized results and performance comparisons are shown in Figure 5 and Figure 6 respectively. Since the synthesized results of the three approaches are indistinguishable, we only show those of bidirectional A* with cut adjustment in Figure 5.

In the first test case ("simple walk") the user draws a straight line in a side view to guide the character to walk from left to right, as shown in Figure 5a. This case is simple and the two search trees are almost balanced, so adjusting the cut provides no further speed up. The second test case ("jump and duck") is harder because the constraint is abstract and strays away from the actual trajectory. A larger search space is required to better fulfill the constraints. Figure 6 shows that both versions of bidirectional A* outperform standard A*. Bidirectional A* without dynamic cut adjustment, however, does not fully utilize the advantage of search space splitting. Since the link of the jumping motion is longer than average, the forward search tree terminates much earlier, and dynamic cut adjustment can improve performance by moving the cut towards the other end. In the next test ("walk and pick"), shown in Figure 5c, the character is asked to detour in an S route to pick up a ball. Both bidirectional versions outperform standard A* search by a large margin. Our approach is more than seventy times faster than unidirectional search. The main reason is that this case is more favorable to backward search, so bidirectional A* with cut adjustment gains an advantage by moving the cut toward the start state several times. In the last experiment ("long walk") the character is asked to walk a long way before making a 180-degree turn in the end, as shown in Figure 5d. This example favors forward search a bit more than backward search. Without cut adjustment, the forward search waits while backward search expands ten times more nodes, so the performance is even slightly worse than A*. With our cut adjustment the cut is moved backward twice to balance the search, and we achieve interactive performance even for this motion that takes 30 seconds to execute.

5.2. Scalability

Our bidirectional strategy can be applied on many graph-like structures to improve the performance of motion synthesis, e.g. state machines, motion graphs, or even interpolated motion graphs [SH07]. Also, the bidirectional strategy can be applied on other A* variants (truncated A*, inflated A*, or

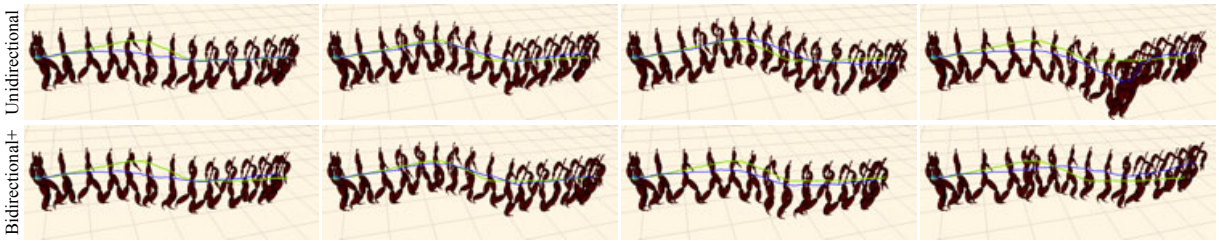


Figure 7: *Synthesized results from unidirectional and bidirectional search. The result motions are about 7 second long. The green line indicates the input stroke, while the blue line represents the trajectory of the synthesized motion.*

anytime A*), which would promise further performance improvement at the cost of reduced quality of the solution.

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

First, we compare unidirectional and bidirectional search (with dynamic cut adjustment) using a motion graph and an interpolated motion graph on the same input. The synthesized motions and performance results are shown in Figure 7 and Figure 8 respectively. With the interpolated motion graph, the synthesized motion can fulfill the constraint more precisely, but the running time increases accordingly. With depth reduction, however, our bidirectional search suffers less from the increased graph size and achieves a speedup of more than 10. We also implemented inflated A* by multiplying the estimated cost with an error tolerance $\delta > 1$, so that the cost of the solution is bounded from above by δ times the cost of an optimal solution. From Figure 8, we can see that even with $\delta = 10$, unidirectional search still requires 5 seconds, which is 3.7 times slower than our bidirectional search with no inflation ($\delta = 1$, 1.36 seconds). This shows that our algorithm is able to find a better solution in a considerably shorter amount of time (the quality of the synthesized motions can be compared in Figure 7).

Secondly, we make the same comparison as above but with a longer and more complicated input. The synthesized results and performance comparisons are shown in Figure 9 and Figure 10 respectively. In this case, without inflation, both unidirectional and bidirectional search cannot find a solution within two minutes, and more than ten million search nodes are expanded. Setting $\delta = 2$, we are able to find a solution with bidirectional inflated A*, but fail again with standard inflated A*. With standard inflated A*, we need to set a much larger tolerance $\delta = 10$ to find a solution in a comparable amount of time. Since the cost of the solution is only guaranteed to be bound within 10 times the optimal cost, its quality is reduced significantly as shown in Figure 9.

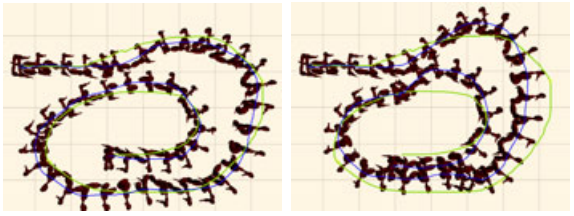
			Unidirectional	Bidirectional+
MG	δ = 1	time	0.297 s	0.078 s
		exp	47,821	4,786 + 6,310
		speedup	1.0x	**3.80x**
IMG	δ = 1	time	14.171 s	1.36 s
		exp	2,450,768	78,278 + 83,434
		speedup	1.0x	**10.41x**
IMG	δ = 3	time	10.547 s	0.078 s
		exp	1,235,655	1,447 + 3,119
		speedup	1.0x	**135.21x**
IMG	δ = 10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

Figure 8: *Performance comparison between unidirectional and bidirectional search. We applied both search algorithms on a motion graph (*MG*) and an interpolated motion graph (*IMG*) with the same input. We also made comparisons with both standard A* ($\delta = 1$) and inflated A* ($\delta > 1$).*

Although we demonstrated the bidirectional strategy only on basic A* and inflated A*, we can apply it to other variants too. We would expect similar results for example for anytime A*, which is a variant of inflated A* that dynamically adjusts the optimality bound δ at run time. We can also adopt the approach of pre-expanding A* search trees [LK06] to further enhance the search performance.

6. Conclusions

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



Bidirectional+, $\delta = 2$ Unidirectional, $\delta = 10$

Figure 9: *Synthesized results from unidirectional and bidirectional inflated A*. Note the deviation from the synthesized motion (blue line) to the input (green line) in the right image due to large inflation. Bidirectional search produces a higher quality result ($\delta = 2$) in a shorter amount of time (6.6 sec.) than unidirectional search ($\delta = 10$, 10.18 sec).*

search type	time	exp
Unidirectional, $\delta = 2$	> 2 mins	> 10 million
Bidirectional+, $\delta = 2$	6.60 s	354,641 + 459,431
Unidirectional, $\delta = 10$	10.18 s	2,385,957

Figure 10: *Performance comparison between unidirectional and bidirectional search with interpolated motion graph using a longer and more complicated input.*

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

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

Appendix

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

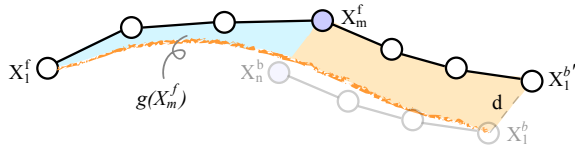


Figure 11: *Direct merging result of the two partial paths in Figure 4a found with bidirectional search.*

Because of discretization, most solutions will not meet the goal exactly, but it is important that the motion should fulfill the end constraint precisely. Thus, to evaluate the cost of a solution, we warp its tail to the goal position. This warping step will change the cost evaluated during the search framework. The cost difference between the original and warped solutions, however, is bounded above by a linear function, $\sigma(\varepsilon, l)$, where l is the length of solution, and ε is a threshold used to discard solutions with large deviation. The reason why the bound depends on ε and l can be visualized intuitively as the difference of the shaded area in Figure 11 and Figure 4b.

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

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

Acknowledgments

We are grateful to Liming Zhao and Alla Safonova for generously providing their motion capture data. We also thank Kuei-Chun Hsu for help with motion capture and character rigging. Thanks to Peng Du, Will Chang, Toshiya Hachisuka for the invaluable discussion and feedback, and to the anonymous reviewers for their helpful comments.

References

[ACCO05] ASSA J., CASPI Y., COHEN-OR D.: Action synopsis: pose selection and illustration. *ACM Trans. Graph. 24*, 3 (2005), 667–676. 6

[AF02] ARIKAN O., FORSYTH D. A.: Interactive motion generation from examples. *ACM Trans. Graph. 21*, 3 (2002), 483–490. 2

[AFO03] ARIKAN O., FORSYTH D. A., O'BRIEN J. F.: Motion synthesis from annotations. *ACM Trans. Graph. 22*, 3 (2003), 402–408. 2

[BCvdPP08] BEAUDOIN P., COROS S., VAN DE PANNE M., POULIN P.: Motion-motif graphs. In *SCA'08: Proc. ACM/Eurographics Symposium on Computer Animation* (2008). 2

[CLS03] CHOI M. G., LEE J., SHIN S. Y.: Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Trans. Graph. 22*, 2 (2003), 182–203. 2

[GG98] GAEDE V., GÜNTHER O.: Multidimensional access methods. *ACM Computing Surveys 30*, 2 (1998), 170–231. 5

[Gle01] GLEICHER M.: Motion path editing. In *I3D '01: Proc. Symposium on Interactive 3D graphics* (2001), pp. 195–202. 7

[HG07] HECK R., GLEICHER M.: Parametric motion graphs. In *SI3D* (2007), pp. 129–136. 2

[HKL*99] HOFF III K. E., KEYSER J., LIN M., MANOCHA D., CULVER T.: Fast computation of generalized voronoi diagrams using graphics hardware. In *Proc. ACM SIGGRAPH 99* (1999), pp. 277–286. 6

[KGP02] KOVAR L., GLEICHER M., PIGHIN F. H.: Motion graphs. *ACM Trans. Graph. 21*, 3 (2002), 473–482. 2, 3, 6

[KK97] KAINDL H., KAINZ G.: Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research 7* (1997), 283–317. 2

[KL00] KUFFNER J. J., LAVALLE S. M.: Rrt-connect: An efficient approach to single-query path planning. In *Proc. IEEE ICRA* (2000), pp. 995–1001. 3

[Kwa89] KWA J. B.: Bs*: an admissible bidirectional staged heuristic search algorithm. *Artif. Intell. 38*, 1 (1989), 95–109. 2

[LaV06] LAVALLE S. M.: *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006. 2

[LCR*02] LEE J., CHAI J., REITSMA P. S. A., HODGINS J. K., POLLARD N. S.: Interactive control of avatars animated with human motion data. *ACM Trans. Graph. 21*, 3 (2002), 491–500. 2, 3, 7

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. *ACM Trans. Graph. 25*, 3 (2006), 579–588. 5

[LK00] LAVALLE S. M., KUFFNER J. J.: Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions* (2000), pp. 293–308. 2

[LK05] LAU M., KUFFNER J. J.: Behavior planning for character animation. In *SCA'05: Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2005), pp. 271–280. 2, 3

[LK06] LAU M., KUFFNER J. J.: Precomputed search trees: Planning for interactive goal-driven animation. In *SCA'06: Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Sept. 2006), pp. 299–308. 2, 9

[LS99] LEE J., SHIN S. Y.: A hierarchical approach to interactive motion editing for human-like figures. In *Proc. ACM SIGGRAPH 99* (1999), pp. 39–48. 7

[Osh05] OSHITA M.: Motion control with strokes. *Journal of Visualization and Computer Animation 16*, 3-4 (2005), 237–244. 3, 5

[Poh71] POHL I.: Bi-directional search. *Machine Intelligence 6* (1971), 127–140. 2

[RN03] RUSSELL S., NORVIG P.: *Artificial Intelligence: A Modern Approach*, second ed. Prentice Hall, 2003. 2, 3

[SH05] SAFONOVA A., HODGINS J. K.: Analyzing the physical correctness of interpolated human motion. In *SCA '05: Proc. 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2005), pp. 171–180. 7

[SH07] SAFONOVA A., HODGINS J. K.: Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph. 26*, 3 (2007), 106. 2, 3, 6, 7, 8

[SH08] SHIRATORI T., HODGINS J. K.: Accelerometer-based user interfaces for the control of a physically simulated character. *ACM Trans. Graph. 27*, 5 (2008), 1–9. 3

[SKF07] SHAPIRO A., KALLMANN M., FALOUTSOS P.: Interactive motion correction and object manipulation. In *I3D '07: Proc. Symposium on Interactive 3D Graphics and Games* (2007), pp. 137–144. 3

[SKG05] SUNG M., KOVAR L., GLEICHER M.: Fast and accurate goal-directed motion synthesis for crowds. In *SCA '05: Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2005), pp. 291–300. 3

[SO06] SHIN H. J., OH H. S.: Fat graphs: constructing an interactive character with continuous controls. In *SCA'06: Proc. ACM/Eurographics Symposium on Computer Animation* (2006). 2

[TB96] TOLANI D., BADLER N. I.: Real-time inverse kinematics of the human arm. *Presence 5*, 4 (1996), 393–401. 7

[TBvdP04] THORNE M., BURKE D., VAN DE PANNE M.: Motion doodles: an interface for sketching character motion. *ACM Trans. Graph. 23*, 3 (2004), 424–431. 3, 5

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANETS D., GROSS M.: Optimized spatial hashing for collision detection of deformable objects. In *Proc. VMV* (2003), pp. 47–54. 5

[WB04] WANG J., BODENHEIMER B.: Computing the duration of motion transitions: an empirical approach. In *SCA '04: Proc. ACM/Eurographics Symposium on Computer Animation* (2004). 6

[WP95] WITKIN A. P., POPOVIC Z.: Motion warping. In *Proc. ACM SIGGRAPH 95* (1995), pp. 105–108. 7

[ZS08] ZHAO L., SAFONOVA A.: Achieving good connectivity in motion graphs. In *SCA'08: Proc. ACM/Eurographics Symposium on Computer Animation* (2008). 7