Evaluating Motion Graphs for Character Animation

Paul S. A. Reitsma and Nancy S. Pollard
Carnegie Mellon University

Realistic and directable humanlike characters are an ongoing goal in animation. Motion graph data structures hold much promise for achieving this goal; however, the quality of the results obtainable from a motion graph may not be easy to predict from its input motion clips. This paper describes a method for using task-based metrics to evaluate the capability of a motion graph to create the set of animations required by a particular application. We examine this capability for typical motion graphs across a range of tasks and environments. We find that motion graph capability degrades rapidly with increases in the complexity of the target environment or required tasks, and that addressing deficiencies in a brute-force manner tends to lead to large, unwieldy motion graphs. The results of this method can be used to evaluate the extent to which a motion graph will fulfill the requirements of a particular application, lessening the risk of the data structure performing poorly at an inopportune moment. The method can also be used to characterize the deficiencies of motion graphs whose performance will not be sufficient, and to evaluate the relative effectiveness of different options for improving those motion graphs.

Categories and Subject Descriptors: Computer Graphics [**Three-Dimensional Graphics and Realism**]: Animation

General Terms: Experimentation, Measurement, Reliability

Additional Key Words and Phrases: motion capability, capability metrics, motion capture, human motion, motion graphs, motion graph embedding, editing model

1. INTRODUCTION

Character animation has a prominent role in many applications, from entertainment to education. Automated methods of generating character animation, such as motion graphs, have a wide range of benefits for such applications, especially for interactive applications. When coupled with motion capture and motion editing, motion graphs have the potential to allow automatic and efficient construction of realistic character animations in response to user or animator control.

Unfortunately, many of these potential benefits are difficult to realize with motion graphs. One major stumbling block is a lack of understanding of the effects of the motion graph data structure; it is not known with any confidence what a character can or cannot do when animated by a particular motion graph in a particular environment. Due to this lack of knowledge of a motion graph's capabilities, they are not reliable enough for many applications, especially interactive applications where the character must always be in a flexible and controllable state regardless of the control decisions of the user. Even if a particular motion graph does happen to fulfill all our requirements, that reliability will go unknown and – for a risk-averse application – unused without a method to evaluate and certify the capability of the motion graph.

Author's address: Paul Reitsma, Computer Science Department, O'Reilly Institute, Trinity College Dublin, Dublin 2, Ireland.

This paper addresses the problem of analyzing a motion graph's capability to create required animations. To do so, we propose a definition of a motion graph's capability, describe a method to quantitatively evaluate that capability, and present an analysis of some representative motion graphs. The method identifies motion graphs with unacceptable capability and also identifies the nature of their deficiencies, shedding light on possible remedies. One important finding is that the capability of a motion graph depends heavily on the environment in which it is to be used. Accordingly, it is necessary for an evaluation method to explicitly take into account the environment in which a motion graph is to be used, and we introduce an efficient algorithm for this purpose. In addition, the information obtained from this evaluation approach allows well-informed tradeoffs to be made along a number of axes, such as trading off motion graph capability for visual quality of the resulting animations or for motion graph size (i.e., answering "how much motion capture data is enough?"). As well, the evaluation approach can offer insight into some strengths and weaknesses of the motion graph data structure in general based on the analysis of a range of particular motion graphs. We discuss briefly how our method for analyzing individual motion graphs, as well as our conclusions about the motion graph data structure in general, might be useful for the related problem of motion synthesis. Finally, we conduct experiments to examine the space/time scaling behavior of the evaluation method, as well as its overall stability and validity over various scenarios.

1.1 Contributions

The primary contributions of this paper are:

—**A statement of the problem of evaluating global properties of motion generation algorithms.** We propose that it should be possible to quantitatively evaluate a character's ability to perform a suite of tasks and to use these results to compare motion graphs or other algorithms for motion generation.

—**A concrete method for evaluating motion graphs for applications involving a broad class of motion types.** We offer metrics to capture several measures of a character's ability to navigate effectively within an environment while performing localized tasks at arbitrary locations.

—**An efficient algorithm for embedding a motion graph into a given environment for analysis.** The algorithm embeds the entire motion graph, allowing analysis of the full variability inherent in the available motion clips, and is efficient enough to scale to large environments or motion graphs.

—**Benchmarking results for a sample dataset over various environments.** We show that the ability to navigate a simple environment is easy to achieve, but that increased complexity in either the task or the environment substantially increases the difficulty of effectively using motion graphs.

While benchmarking example motion graphs, we obtained two major insights about motion graphs as they are commonly used today:

—Simple tasks in simple environments are easy for most motion graphs to perform, but capability degrades rapidly with increasing complexity.

—Reactivity (e.g., to user control) is poor for motion graphs in general. Furthermore, techniques such as adding data, allowing motions to be edited more, or using hub-based motion graphs did not raise reactivity to acceptable levels in our experiments.

Portions of this research have appeared previously in [Reitsma and Pollard 2004]. This paper incorporates substantial improvements, including: *(1)* a new embedding algorithm with asymptotically lower memory requirements which allows evaluation of scenarios orders of magnitude larger; *(2)* several times as many evaluation metrics covering a much wider range of practical considerations; *(3)* the ability to evaluate a wider class of motion graphs, including those with user-triggered actions such as picking up an object or ducking; *(4)* more thorough evaluation of the scaling behavior of the system; and *(5)* more thorough evaluation of

some of the underlying assumptions of the system. The majority of techniques and results in this paper are new.

2. BACKGROUND

It is necessary to choose a particular automatic animation generation technique in order to demonstrate concrete examples of our measurements in practice. We used motion graphs, due to a number of benefits the algorithm offers (e.g., realistic motion, amenability to planning, similarity to in-production systems), and due to the significant body of supporting work on the technique (e.g., [Molina-Tanco and Hilton 2000] [Lee et al. 2002] [Kovar et al. 2002] [Arikan and Forsyth 2002] [Li et al. 2002] [hoon Kim et al. 2003] [Arikan et al. 2003] [Kwon and Shin 2005] [Lee et al. 2006]).

Some previous research has looked at the problem of altering motion graphs to overcome their disadvantages. Mizuguchi et al. [2001] examines a common approach used in the production of interactive games, which is to build a motion-graph-like data structure by hand and rely heavily on the skills and intuition of the animators to tune the results. Gleicher et al. [2003] developed tools for editing motion data in a similar manner, in particular the creation of transition-rich "hub" frames. Lau and Kuffner [2005] use a hand-tuned hub-based motion graph to allow rapid planning of motions through an environment. Sung et al. [2005] make use of a probabilistic roadmap to efficiently speed up path searches using a motion graph. Ikemoto et al. [2006] use multi-way blending to improve the transitions available within a set of motion capture data. Our analysis approach is complementary, as it can be used to evaluate and provide information on motion graphs generated in any fashion. Shin and Oh [2006] and Heck and Gleicher [2007] introduced methods to associate a single edge (or node) of a motion graph with multiple related motion clips, allowing any path using that edge (or node) to use a blended combination of those motion clips.

In order to make embedding large motion graphs into large environments a tractable task, we use a grid-based approach inspired by grid-based algorithms used in robotic path-planning (e.g., [Latombe 1991] [Lozano-Pérez and O'Donnell 1991] [Donald et al. 1993]). We note that these algorithms are designed for a different purpose than ours, namely that of finding a single good path for a robot to move from start to goal positions, and hence cannot be easily adapted to our analysis problem. Instead of finding a single path, our embedding approach represents the set of all possible paths through the environment for the purpose of evaluating character capabilities.

Four other projects have considered placing a motion graph into an environment in order to capture its interaction with objects. Research into this area was pioneered by Lee et al. [2002]. Their first approach captured character interactions with objects inside a restricted area of the environment. A second approach ([Choi et al. 2003]) demonstrates how to incorporate a portion of a motion graph into an environment by unrolling it onto a roadmap ([Kavraki and Latombe 1998]) constructed in that environment. Reitsma and Pollard [2004] presented an earlier version of this work, which differed as noted in Section 1.1. Suthankar et al. [2004] applied a similar embedding process to model the physical capabilities of a synthetic agent, and Lee et al. [2006] applied a similar embedding process to incorporate motion data into repeatable tiles. Our work is complementary to these projects, with the information gained via analysis being useful for improving motion-generation algorithms. For example, analysis could drive the creation of a sparse-but-sufficient roadmap.

While some work has been done on the evaluation of character animation, most such work has examined only local quantities corresponding to a small set of test animations. For example, several researchers have examined user perceptions of animated human motion (e.g., [Hodgins et al. 1998] [Oesker et al. 2000] [Reitsma and Pollard 2003] [Wang and Bodenheimer 2003] [Wang and Bodenheimer 2004] [Harrison et al. 2004] [Ren et al. 2005], [Safonova and Hodgins 2005]). By contrast, the goal of our approach is to evaluate more global properties of the motions available to animate the character, such as the expected efficiency of optimal

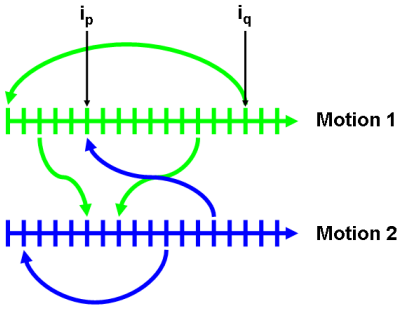


Fig. 1. A motion graph is formed by adding additional transitions between frames of motion capture data. In this diagram, each vertical hashmark is a frame of motion; connecting arrows represent these additional transitions. The frames of Motion 1 from i_p to i_q represent one clip (node) in the motion graph, and the edges of the motion graph are the connections between clips. The clip represented by the frames from i_p to i_q, for example, has two incoming edges: one from Motion 2 (blue arrow) and another from frame i_{p-1} in Motion 1.

point-to-point paths through the environment. These local and global measurements are complementary, combining to form a more complete understanding of the animations that can be created.

3. SYSTEM OVERVIEW

Our system takes as input *(1)* a set of motion capture data, *(2)* visual quality requirements for the animations (represented as a model of acceptable motion editing), *(3)* a set of motion capability requirements (i.e., what tasks the character must be able to accomplish), and *(4)* an environment in which the character is to perform those tasks. These inputs define the *scenario* under analysis.

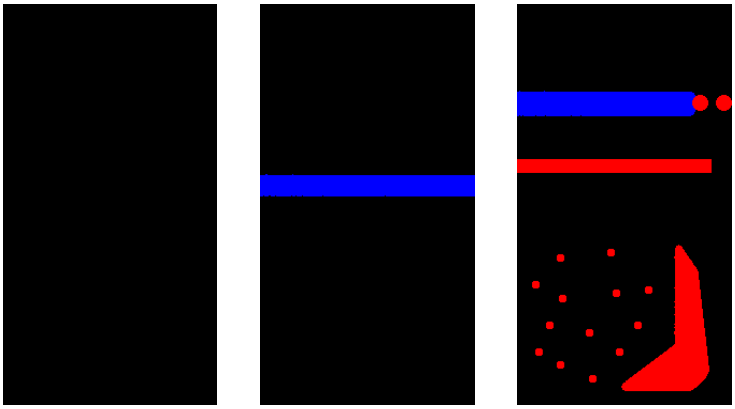
The motion capture data is processed to form a motion graph (see Section 3.1). We further process this motion graph to capture interactions with the target environment (Section 3.2 and Section 4), based on a model of motion editing (Section 3.3). The resulting data structure is used to measure the animated character's ability to successfully complete the required tasks in the given environment (Section 3.4 and Section 5).

3.1 Introduction to Motion Graphs

The following is a brief introduction to the idea of a motion graph; much more detail can be found in the references, for example [Kovar et al. 2002].

Normally, motion capture data is played sequentially; to replay the motion of clip i, the animation transitions from frame i_k to frame i_{k+1}. The intention of a motion graph is to allow additional flexibility when using the motion capture data. The data is examined to find additional transitions of the form i_k to j_m (i.e., clip i, frame k to clip j, frame m) that will result in acceptable motions. One common criterion for selection of such transitions is that frames i_k and j_{m-1} are sufficiently similar.

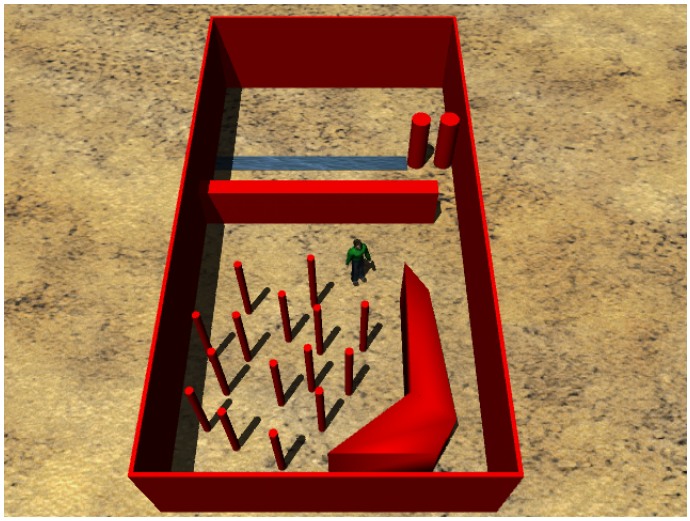
A motion graph is formed by considering these additional transitions to be the edges of a directed graph (see Figure 1). Nodes in the graph are sequences of frames extending between transitions (e.g., frame i_p to frame i_{q-1}), and are referred to in this paper as motion *clips*. Traversing this graph consists of playing a sequence of these clips, with all clip-to-clip boundaries being at one of these additional transitions. To allow continuous motion generation, only the largest strongly connected component (SCC) of the motion graph is used.



(a) Empty environment

(b) Environ with trench

(c) More complex environ



(d) Complex environ (3D)

Fig. 2. Example environments. The upper obstacle (blue) in environments (b) and (c) is annotated as a chasm or similar obstacle, meaning it can be jumped over. The environment shown in (c) and (d) is our baseline reference environment.

3.2 Capturing Motion Graph/Environment Interaction

The details of a particular environment can have a very significant impact on the capabilities imparted to an animated character by a motion graph. The simplest example of this is to consider two environments (Figure 2(a) and 2(b)), one of which has a deep trench bisecting it that must be jumped over; for a motion graph with no jumping motions, the two environments will induce a very different ratio between the total space of the environment and the space which can be reached by a character starting in the lower portion. A more complex example is shown in Figure 2(c) and 2(d), where small changes to the sizes and positions of obstacles—changes that may appear superficial—produce very large differences in the ability of the character to navigate through the environment. Due to this strong environmental influence on the capabilities of a motion graph, the only way to understand how a motion-graph-driven animated character will actually perform in a given environment is to analyze the motion graph in the context of that environment.

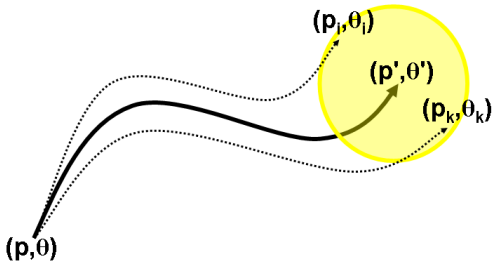


Fig. 3. A motion clip can be edited to place its endpoint anywhere within its footprint (yellow region). Dotted arrows show the edited paths corresponding to the possible endpoints (p_i, θ_i) and (p_k, θ_k).

For any given scenario to be evaluated, we need a way to capture the influence of the environment on the capabilities of the motion graph. There are a number of possible approaches to capturing the interaction between an environment and the motion graph; our approach is to *embed* the motion graph into the environment—unrolling it into the environment in the manner described in Section 4.

3.3 Visual Quality Requirements

Embedding a motion graph into an environment so that its capabilities can be measured requires making a choice of editing model. Generally, the visual quality of motions created in a motion graph will start at a high base due to the verisimilitude of the underlying motion capture data, and will be degraded by any editing done to the motion, either due to the loss of subtle details or to the introduction of artifacts such as foot sliding. Such editing is necessary to transition between the different motion clips of the motion graph and to precisely target the character to achieve certain goals (such as picking up an object or walking through a narrow doorway). Additionally, allowing motions to be edited increases the range of motions available to the animation system, increasing the capabilities of the available animations while decreasing their visual quality.

We assume that current and future research on motion editing and on perceptual magnitude of editing errors will allow animators to determine the extent to which a motion clip can be edited while maintaining sufficient visual quality to meet the given requirements. Given those bounds on acceptable editing, a motion clip starting at a point p and character facing θ which would normally terminate at a point $p\prime$ and facing $\theta\prime$ can be edited to terminate at a family of points and facings $\{p_i, \theta_i\}$ containing $\{p\prime, \theta\prime\}$. In general, the better the editing techniques and the lower the visual quality requirements, the larger this family of valid endpoints for a given clip. This family of points is known as the *footprint* of the motion clip; given a specific starting point and direction, the footprint of a clip is all of the endpoints and ending directions which it can be edited to achieve without breaching the visual quality requirements (see Figure 3).

Design of motion editing models is an open problem; we use a simple linear model, meaning the amount of acceptable editing grows linearly with both distance traveled and motion clip duration (see Appendix A), similar to the approach of Sung et al. [2005] and Lee et al. [2006].

3.4 Motion Capability Requirements

We define the *capability* of a motion graph within an environment as its ability to create motions that fulfill our requirements within that environment. The appropriate metric for evaluating character capabilities depends on the tasks the character is expected to perform. For localized tasks such as punching a target or

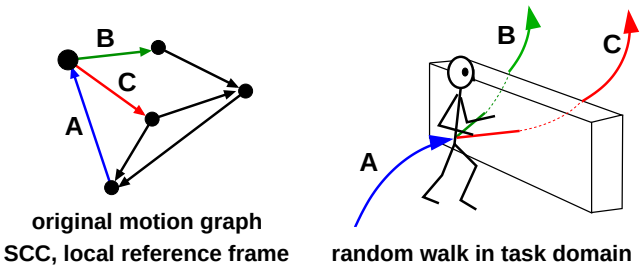


Fig. 4. (Left) A motion graph may be constructed using a reference frame local to the character to preserve flexibility in the character's motion. (Right) Using this motion graph to drive the character through an environment with obstacles can result in dead ends. Because obstacle information is not available in the local reference frame (left), extracting a strongly connected component (SCC) from the original motion graph does not guarantee that the character can move through a given environment indefinitely.

kicking an object, the task may be to efficiently navigate to the target and contact it with an appropriate velocity profile, while for some dance forms the task may be to string together appropriate dance sequences while navigating according to the rules of the dance being performed. This paper focuses on navigation and localized actions (such as punching, ducking, or picking up an object). We focus on the following requirements, chosen so as to approximate the tasks one might expect an animated character to be required to perform in a dynamic scenario, such as a dangerous-environment training simulator or an adventure game:

—Character must be able to move between all major regions of the environment.
—Character must be able to perform its task suite in any appropriate region of the environment; for example, picking up an object from the ground regardless of that object's location within the environment.
—Character must take a reasonably efficient path from its current position to any specified valid target position.
—The previous two items should not conflict; i.e., a character tasked with performing a specific action at a specific location should still do so efficiently.
—Character must respond quickly, effectively, and visibly to user commands.

4. EMBEDDING INTO THE ENVIRONMENT

In this section, we describe the details of our embedding algorithm. To illustrate the design considerations for an embedding approach, consider attempting to compute the value of a sample metric: Simple Coverage. The value of this metric is just the fraction of the environment through which the character can navigate freely. One immediate challenge is to form a compact description of the space of character trajectories – simply expanding the motion graph in a breadth-first fashion from an arbitrary starting state, for example, will rapidly lead to an exponential explosion in the number of paths being considered. A second key challenge is to consider only paths which do not unnecessarily constrain the ability of the character to navigate. For example, avoiding dead ends is necessary for an autonomous character with changing goals, or for a character subject to interactive control. Even though the original motion graph has no dead ends, obstacles can cause dead ends in the environment (see Figure 4)

In order to meet these challenges, we discretize the environment, approximating it with a regular grid of cells. At each cell, we embed all valid movement options available to the character. This embedding forms a directed graph, of which we only use the largest strongly connected component (SCC). Note that this SCC is in the *embedded* graph, not within the original motion graph. For example, in Figure 4, link A is not valid at that position within the environment, and so would be discarded *for that position only*; link A may not

be blocked in other parts of the environment, and hence may be a part of the (embedded) SCC in some positions and not in others.

While computing the embedded SCC has a high one-time cost, that cost is amortized over the many tests run to compute the metrics of interest, each of which is made more efficient by having the information embodied in the SCC available. This savings can reach many orders of magnitude for metrics such as testing for availability of efficient paths through the environment. Consider, for example, the problem of searching for an optimal path in an environment filled with obstacles. With the embedded graph, paths that lead inexorably to collisions (of which there are many) are not part of the SCC and never need to be considered. Computing this SCC in advance, then, allows efficient computations of the capabilities of the motion graph under investigation.

4.1 Discretization

We discretize the environment along the following dimensions, representing the state of the character:

X The x-axis groundplane position of the character's root.

Z The z-axis groundplane position of the character's root.

Θ The facing direction of the character.

C The clip which brought the character to this (X, Z, Θ) point (i.e., the character's pose).

C is inherently discrete, but the groundplane position indices (X, Z) are determined by discretizing the environment into a grid with fixed spacing distance between adjacent X or Z bins. Similarly, the facing of the character is discretized into a fixed number of angular bins.

Typically, a ground-plane position specified by only X and Z is referred to as *grid location*; adding the facing angle Θ denotes a *grid point*; finally, specifying the pose of the character by including the motion clip that brought the character to that point specifies a *grid node* in the 4D state space.

Each grid node can be considered a node of a directed graph, where $[x, z, \theta, c]$ has an edge to $[x\prime, z\prime, \theta\prime, c\prime]$ if and only if:

—Clip c can transition to clip $c\prime$ in the original motion graph

—Given a character with root position (x, z) and facing θ, animating the character with clip $c\prime$ places $(x\prime, z\prime, \theta\prime)$ within the footprint of $c\prime$.

—The character follows a valid path through the environment (i.e., avoids obstacles and respects annotations) when being animated from (x, z, θ) to $(x\prime, z\prime, \theta\prime)$ by the edited clip $c\prime$.

A pair of nodes (u, v) is known as an *edge candidate* if the first two criteria hold (i.e., the edge will exist unless the environment renders it invalid). Note that collision detection is a part of the third criterion (that the character follows a valid path through the environment), and hence is performed every time an edge candidate is examined.

For example, suppose clip A can be followed by either clip B or clip C in the original motion graph. In the environment represented in Figure 5, the starting point $([1, 1, \frac{\pi}{2}, A]$ is marked with a blue star, endpoints of valid edges with green circles, and edge candidates which are not valid edges are marked with red squares. The edge $([1, 1, \frac{\pi}{2}, A], [5, 1, \frac{\pi}{2}, C])$ is in the embedded graph, since its endpoint is within C's footprint and the path to it is valid. By contrast, there is no edge from $([1, 1, \frac{\pi}{2}, A], [5, 0, \frac{\pi}{2}, C])$, since the edited path of clip C (dotted arrow) is not valid in the environment (it intersects an obstacle).

Note that this discretization approach offers a tradeoff between precision and computation. Our metrics generally give broadly similar results across a range of grid sizes (see Section 6.5), suggesting that the values computed in a discretized setting are reasonable approximations of those values in the continuous case.

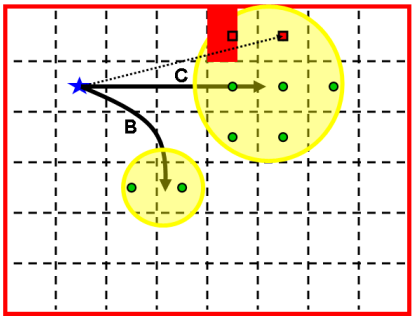
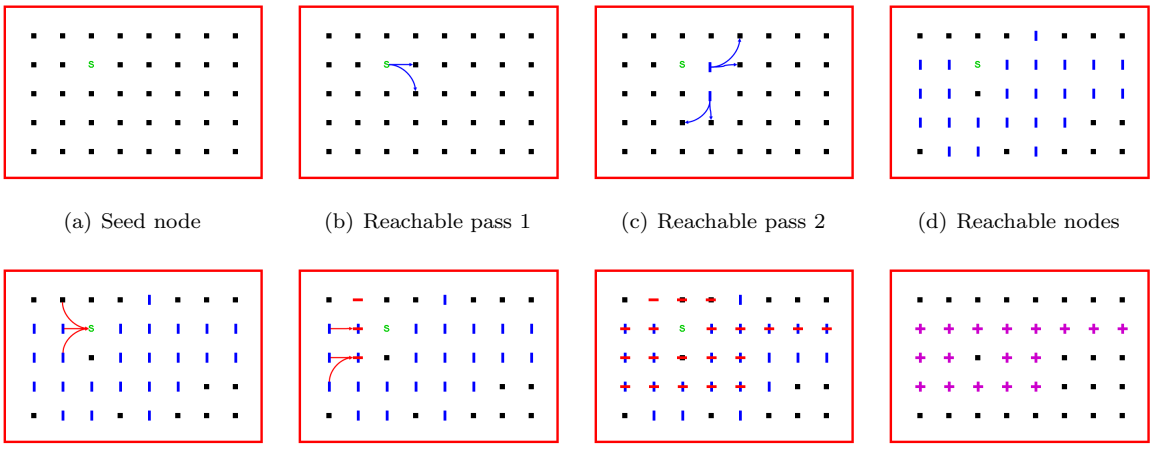


Fig. 5. Computing edges from a single node; the endpoints of valid edges are marked as green circles. Note that not all nodes within a clip's footprint are the endpoints of edges, as obstacles can obstruct the edited path from the source to the target node (dotted path).



(a) Seed node (b) Reachable pass 1 (c) Reachable pass 2 (d) Reachable nodes

(e) Reaching pass 1 (f) Reaching pass 2 (g) Reaching nodes (h) SCC

Fig. 6. Steps of the embedding algorithm. (a) A seed node (green "S") is chosen. (b) First pass of the reachable flood marks nodes reachable in one step from the seed node. (c) Second pass marks additional nodes reachable in two steps. (d) All reachable nodes are marked (blue vertical hashes). (e) First pass of the reaches flood marks nodes which reach the seed node in one step. (f) Second pass marks additional nodes reaching in two steps. (g) All reaching nodes are marked (red horizontal hashes). (h) Intersection of set of reachable nodes and set of reaching nodes (purple "+") is the SCC of the embedded graph.

4.2 Space-Efficient Unrolling

Reitsma and Pollard [2004] present an embedding algorithm that works by computing all edges and then using Depth-First Search to find the largest strongly connected component (SCC). The main limitation of this algorithm is that it requires explicitly storing all edges needed to compute the embedded graph. While this is an efficient and sensible approach for smaller environments or coarser grids, due to memory considerations it significantly limits the size of environments and the resolution at which they can be processed.

As an alternative that requires much less memory, we propose a flood-based algorithm which takes advantage of the fact that edges outgoing from or incoming to a node can be computed only as needed. For

grid nodes u and v, an edge candidate (u, v) will be in the SCC if and only if nodes u and v are both in the SCC, and (u, v) is an edge of the embedded graph (i.e., its associated path is valid within the environment), so storing only the nodes in the SCC allows computation of the correct edges as needed.

Finding the SCC for use with this on-the-fly approach can be done efficiently in the discretized environment (see Figure 6):

—Choose a source node s (green "S").

—Flood out from s, tagging all reachable nodes (blue vertical hashes).

—Flood into s, tagging all nodes reaching it (red horizontal hashes).

—Intersection of the two tagged sets is the SCC (purple "+").

In practice, motion graphs embedded into environments typically result in an embedded graph with a single SCC whose number of nodes is linear in the total number of nodes in the environment which are not obstructed by obstacles[1]; 10-25% of the total number of nodes in the environment is a typical size for an SCC. Accordingly, s can be chosen uniformly at random until a node inside the SCC is found without an asymptotic increase in time[2].

Flooding is done using a multi-pass Breadth-First Search. For finding the set of Reachable nodes, bit arrays are used to tag which nodes are Active or Reached. Initially only s has its Active bit set, and no nodes have their Reached bits set. At each pass, the algorithm iterates through each node k in the environment. If Active(k) is set, then k is expanded. Expanding a node consists of setting Reached(k) to true, setting Active(k) to false, and finding the set of edges for k[3]. For each of those edges j, Active(j) is set if and only if Active(j) and Reached(j) are both false (i.e., j has not been tagged or expanded yet). The algorithm ends when there is no node k such that Active(k) is set.

Since any node's Active bit is set to true at most once, the algorithm terminates. Since all nodes reachable from s will have their Active bit set, the algorithm correctly computes the set of nodes reachable from s. Each pass of the algorithm may touch every node in the environment, but each edge is expanded only once per flood direction. Accordingly, the runtime of the algorithm is the cost of computing whether each edge is valid in the environment plus the cost of reading each node's tags for each iteration until until the search is complete; i.e., $\Theta(e) + \Theta(D * N)$, for e the number of edges in the SCC, N the number of nodes in the environment, and D the diameter of the embedded graph (i.e., the maximum depth of the Breadth-First Search). In practice, the $\Theta(e)$ term dominates, making the algorithm approximately as efficient as regular Breadth-First Search.

The set of nodes which can reach s is computed analogously; however, since the set of reachable nodes in already known, only those nodes need to be expanded, as no other nodes can be in the SCC. This offers substantial computational savings.

Note that storing the Reachable and Reaching sets requires two bits of storage per node, and that another bit is required as scratch memory for the Active set, for a total memory requirement of three bits per node.

---

[1]There is some evidence this should be expected; see, for example, [Muthukrishnan and Pandurangan 2005] regarding this property in random geometric graphs.

[2]In practice, the actual overhead is minimal if fast-fail tests are used. Testing that source nodes have at least 5,000 reaching and reachable nodes efficiently rejects most nodes outside the SCC; in our experiments, testing that the source node can reach itself was the only other fast-fail test required.

[3]Recall from Section 4.1, an edge from node $[x, z, \theta, c]$ to node $[x\prime, z\prime, \theta\prime, c\prime]$ exists if and only if (1) clip c can transition to clip $c\prime$ in the original motion graph, (2) starting the character at position (x, z, θ) and animating it with clip $c\prime$ will place $(x\prime, z\prime, \theta\prime)$ within that clip's footprint, and (3) the resulting path is valid in the environment (i.e., avoids obstacles and respects annotations).

4.2.1 *Space Complexity.* We can estimate the theoretical space complexity of the embedded graph with respect to the parameters affecting it. For a given environment, the number of nodes in the embedded graph is

$$O(n) = O(AxzaC) \tag{1}$$

and the number of edges in the embedded graph is

$$O(e) = O(Ax^2z^2a^2Cb) = O(nxzab) \tag{2}$$

where:

A = the accessible (i.e., character would not collide with an obstacle) area of the environment, in m^2.
x = the discretization of the X axis, in grid cells per m.
z = the discretization of the Z axis, in grid cells per m.
a = the discretization of the character's facing angle, in grid cells per 2π radians.
C = the number of motion clips used (i.e., the number of nodes in the original motion graph).
b = the mean branching factor of the original motion graph.

Equation 1 simply notes that the number of nodes in the SCC is linear in the total number of nodes into which the environment is discretized. Equation 2 demonstrates how the average number of edges per node is constant with increasing environment size, but increases linearly with increasing resolution (in each dimension), due to more grid points being packed under each clip's editing footprint. Note that a clip's editing footprint increases in size with the length and duration of the clip, but that enforcing a minimum and maximum clip duration when creating the motion graph ensures two clips' footprints will differ in area by at most a constant factor.

The one-step unrolling algorithm presented in [Reitsma and Pollard 2004] stores the edges of this embedded graph explicitly, and so has space complexity $O(nxzab)$. By contrast, the flood-based algorithm presented here has space complexity $O(n)$; i.e., linear in the number of nodes in the embedded graph, rather than in the number of edges, meaning the amount of memory required is reduced by a factor of $\Theta(xzab)$. In practice, the big-Theta notation hides an additional large constant factor of improvement, as only a few bits are required to compute and store each node.

The empirical scaling behavior of these two algorithms with respect to environments of increasing size is examined in Section 6.4.3.

4.2.2 *Time Complexity.* While the space-efficient algorithm requires asymptotically less space than the one-step unrolling algorithm, it typically does not require asymptotically more time.

The algorithm of Reitsma and Pollard [2004] computes each edge candidate in the environment once, stores valid edges, and accesses each cached edge twice when using Depth-First Search to find the SCC. The runtime for this algorithm is:

$$T_{rp04} = \Theta(oNbf + 2E) = \Theta(oE + 2E) = \Theta(oE) \tag{3}$$

where N is the number of unobstructed nodes in the environment, f is the mean number of nodes under the footprint of a clip, E is the number of edges in the environment, and o is the mean number of obstacle-intersection tests required when computing each edge.

When expanding a node, the algorithm of Section 4.2 computes either incoming edge candidates or outgoing edge candidates. As a node will be expanded only once in each direction (Reachable/Reaching), each outgoing edge candidate from that node and each incoming edge candidate to that node will be computed only once when finding the SCC. Accordingly, the runtime of our algorithm is:

$$T_{flood} = \Theta(on_rbf + onbf\prime) = O(oE) \tag{4}$$

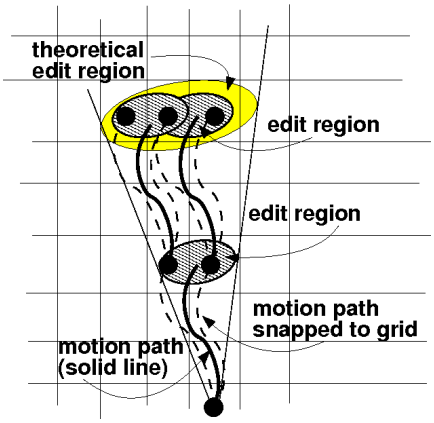


Fig. 7. In our grid-based algorithm, the end of each motion segment is snapped to the centers of grid points inside the editing footprint for that motion segment. This figure compares how growth in this theoretical edit region compares to the regions actually used in the grid-based algorithm.

where n_r is the number of nodes reachable from the SCC, and $f\prime$ is the mean number of nodes under the *reverse footprint* of a clip (i.e., the set of start points $\{p, \theta\}$ such that the footprint of clip c played from that point will contain the end point $(p\prime, \theta\prime)$).

In the typical case, where $n = \Theta(N)$ (i.e., the number of nodes in the SCC is linear in the number of unobstructed nodes in the environment), the runtimes for T_{rp04} and T_{flood} are asymptotically equivalent. Substantial runtime differences can occur from two sources: first, $f\prime$ will typically be larger than f, due to the manner in which angular edits are made to clips; second, n_r and especially n will typically be smaller than N. Empirical runtimes are examined in Section 6.4.3.

4.3 Correctness of the Embedding Algorithm

Our embedding algorithm is resolution complete in the sense that a sufficiently fine grid will capture all significant variations in character paths that are possible given our motion graph, task domain, and motion editing model. However, at practical grid resolution, some possible paths will be lost. Figure 7 shows an example. Consider a straight walking motion that can repeat. Figure 7 shows a sketch of edit regions grown using our approach, which requires forcing motion clips to terminate at cell centers. The theoretical edit region that would result from playing the same motion clip twice in succession is also shown. Eventually, (after playing the motion clip three or four times in succession), the theoretical edit region will contain grid centers that are not captured by our algorithm. As the grid is made finer, the practical effect of this approximation will decrease, and we discuss the results of experiments testing the stability to changes in grid resolution of our analysis approach in Section 6.5.

The embedding algorithms are made conservative by making connections only to grid centers within the edit region associated with a motion segment. When the algorithms are implemented in this way, no paths can be generated that are not possible given the motion graph, task domain, and motion editing model.

4.4 Obstacles and Annotation Constraints

Annotations can be used to define general constraints on the motions usable in the environment; annotations we used (with examples of use) included:

—Obstacle cannot be traversed (high wall).

—Obstacle can be traversed by jumping motions (chasm).

—Obstacle can be traversed by jumping or stepping motions (low wall).

—Region can be traversed only by sneaking motions (area near guard).

—Region cannot be traversed by jumping motions (normal locomotion should not be assumed to include jumping for no reason).

—Picking-up motions should not be used unless the user selects the action.

Most of these annotations—either defining different obstacle types which interact differently with different motion types, or defining regions where the character must act in a certain way—are self-explanatory, and are handled automatically during collision-detection; any path which violates an annotation constraint will not be expanded, and hence cannot contribute to the embedded graph. Annotations such as the last one are more complicated, as some motions need to be designated as strictly-optional (and hence cannot be used to connect parts of the embedded graph) but readily-available. We refer to these as *selective actions*.

4.5 Selective Actions

Selective actions are those actions which must be strictly optional (i.e., selectable by the user at will, but otherwise never occurring in unconstrained navigation). A user may choose to have the character perform a picking-up action when there are no nearby objects on the floor, for example, but it would look very strange for such a spurious pick-up action to show up in automatically-generated navigation.

An embedded graph created from all motions will not respect this requirement; in particular, parts of the environment may be reachable only via paths which require selective actions, rather than by regular locomotion, and hence those parts of the environment should not be considered reachable.

Selective actions are handled by a slight modification to the embedding algorithm. First, the embedding algorithm described previously is run with only those actions deemed "locomotion" permitted; all other actions are deemed "selective", and are not expanded, although any selective-action nodes reached are marked as such. An exception is made for selective actions which are necessary for traversing an annotated obstacle; e.g., jumping motions are permitted only insofar as they are required to traverse obstacles such as the jumpable-chasm obstacle in Figure 2(b). This creates a backbone embedded graph composed of locomotions.

Next, for each reachable selective action, nodes from all valid paths of reasonable length are added to the embedded graph. The result may not be a strongly connected component, since some of the added paths may not rejoin the SCC; accordingly, the embedding algorithm is rerun (as illustrated in Figure 6) to find the new strongly connected component. This time, however, the resulting SCC must be a subset of the previous embedded graph (i.e., the original SCC plus the added paths); hence, that information can be used to substantially reduce the number of nodes expanded by the embedding algorithm. The resulting embedded graph, known as the *augmented SCC* will automatically include all reachable selective actions which can in turn reach the backbone SCC by paths of reasonable length, and will include all such rejoining paths.

The cost for creating this augmented SCC is the cost for creating the initial SCC, plus the cost of adding in paths from each selective action out to a fixed depth, plus the cost of re-running the flooding algorithm on this subset of nodes. The total cost is:

$$T_{aug} = T_{flood} + T_{paths} + T_{flood\prime} = O(oE) + O(onbfh) + O(oE) = O(oE) \tag{5}$$

where h is the mean depth in edges of the added paths. Comparing to Equation 4, computing the augmented embedded graph is asymptotically equivalent to computing the regular motion graph with all motions treated equally, and in practice requires about twice as much computation time. In addition, storing the "selective" actions during computation of the initial embedded graph and storing that initial SCC during computation

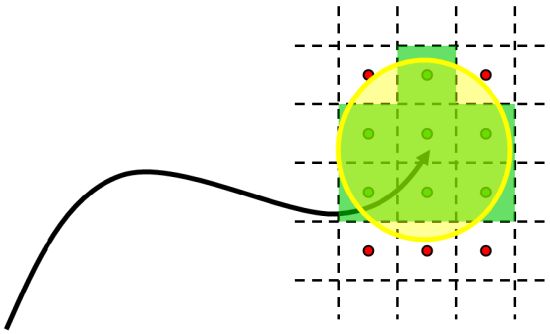


Fig. 8. A grid cell is covered by a clip if that clip's footprint includes the cell's center. (Covered grid cells are in green.)

of the final, augmented SCC requires an extra bit of storage per node, for a total cost of four bits per node in the environment.

We used this type of embedded graph for all of our experiments.

5. MOTION GRAPH CAPABILITY METRICS

To measure the capability of a motion graph, we define metrics which evaluate the motion graph's ability to fulfill the requirements identified in Section 3.4.

5.1 Environment Coverage

The most basic problem a motion graph can have in an environment is simply that it is unable to navigate the character effectively enough to access large regions of that environment. Accordingly, the environment coverage metric is designed to capture the ability of the character to reach every portion of its workspace without becoming irretrievably stuck, similar to the viability domain from viability theory (see, for example, Aubin [1990]). For navigation, this workspace is represented by discretized grid points $\{X, Z, \Theta\}$ (see Section 4.1).

We define grid point (X, Z, Θ) as *covered* by clip c if the center of (X, Z, Θ) is within the footprint of c (see Figure 8).[4] From this occupancy information we compute Environment Coverage as:

$$C_{XZA} = \frac{\sum_i covered(i)}{\sum_i collisionFree(i)} \tag{6}$$

where the numerator contains a count of all 3D grid points which are covered by at least one clip in the embedded graph, and the denominator contains a count of all grid positions which are theoretically valid for character navigation (i.e., some motion type exists for which the character would not collide with an obstacle or violate an annotation constraint; see Figure 9). C_{XZA} is a discretized estimate of the fraction of viable (x, z, θ) workspace through which the character can navigate under arbitrary control. C_{XZ}, the 2D equivalent, can be defined analogously.

Regions with low or no coverage typically indicate that the entrances to those regions from the rest of the environment are highly constricted, and there are few ways to place the motions required to enter that region without colliding with obstacles or violating annotation constraints.

[4]An alternative definition of coverage is that any grid point containing any root position of any frame of any clip is covered. In practice, these definitions give essentially identical results, but the former definition is significantly faster to compute.

Fig. 9. Coverage over a simple environment. Obstacles are in red; covered areas are in grey. Brighter grey means more coverage (i.e., the grid location is covered by more clips).

5.2 Action Coverage

The coverage for action k is defined analogously to Environment Coverage. The 2D version is:

$$C_{XZ,k} = \frac{\sum_i covered_k(i)}{\sum_i collisionFree(i)} \tag{7}$$

where $covered_k(i)$ is true if and only if there exists an edge in the embedded graph such that animating the character with that edge causes the central frame of action k to occur in grid location i.

5.3 Path Efficiency

The path efficiency metric is designed to evaluate the ability of the motion graph to allow the character to navigate efficiently within the accessible portion of the environment. Each path will have a particular value for path efficiency, and hence the goal is to estimate the distribution of path efficiencies over the set of all possible paths through the environment. To estimate this distribution, we must make an assumption about the manner in which paths will be specified. For this paper, we examine the problem of point-to-point navigation, where the (X, Z) values of start and end points are specified (such as with a mouse) and a path must be found for the character to travel between these two points[5]. Ideally, a near-minimal-length path would exist for all pairs of start and end positions. The metric for relative path length in point-to-point navigation is:

$$E_P = \frac{pathLength}{minPathLength} \tag{8}$$

where $pathLength$ is the length of the shortest path available to the character from the start point to the end point and $minPathLength$ is the length of an ideal reference path between those two points (see Figure 10).

When evaluating the path efficiency ratio of a $(start, end)$ pair, we work with the embedded graph described in Section 4. Using this graph ensures that only paths which do not result in dead ends are considered, and also significantly improves the efficiency of shortest path calculations.

Given this embedded graph, we use Monte Carlo sampling to estimate the distribution of path efficiency ratios within the 4D space defined by all valid $(start, end)$ pairs. Start and end positions are selected uniformly at random from the set of grid positions which have non-zero coverage (see Section 5.1). Value $pathLength$, the shortest path available to the character, is computed using A* search through the embedded

[5]Several similar definitions of this metric are possible, such as accepting paths that pass through the target point, rather than only ones ending there. In practical embedded graphs, the results will tend to be very similar, but the definition used here can be computed more quickly, and is closer to the "go to the click" interaction paradigm which motivates the metric.

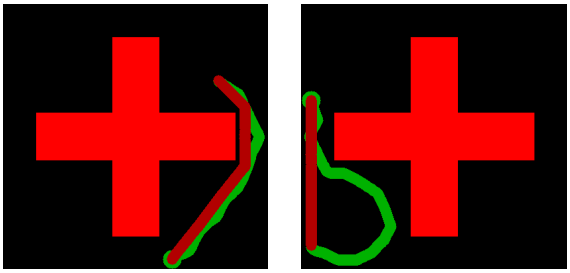


Fig. 10. Paths through a simple environment. Obstacles are in bright red, the *minPathLength* path is in deep red, and the *pathLength* path is in green (starting point is marked with a wider green dot). (Left) A typical path. (Right) The theoretical *minPathLength* path can head directly into the wall, while the *pathLength* path, which relies on motions in the SCC, must end in a state that allows for further movement.

graph. Value *minPathLength*, the shortest path irrespective of available motions, is estimated using A* search over the discretization grid imposed on the environment, with each 2D (X, Z) grid location being connected to all neighbors within five hops in either X or Z (i.e., an 11x11 box). Path efficiency is then computed as in Equation 8.

Due to the highly discrete nature of motion graphs, extremely short paths may have unrepresentative path efficiency ratios according to the presence or absence of a clip of particular length in the motion graph; to reduce this erratic influence and to consider paths more representative of typical user-controlled navigational tasks, we throw away $(start, end)$ pairs whose linear distance is less than 4m and re-select start and end candidates. Since the space complexity of $A*$ grows exponentially with path depth and we expect more sophisticated planning algorithms will be used in practice, we similarly throw out $(start, end)$ pairs whose *minPathLength* is greater than 7m. We believe such longer paths will not have significantly different path efficiency distributions from paths in the $4 - 7$m range, due to the ability to chain shorter paths together to create long ones. Note also that extremely slow-moving motions, such as idling in place, can skew the results. We treat each clip as having a minimum effective speed of 0.5m/s for the purposes of minimum path computation in order to help filter out spurious paths such as ones which make all turns by stopping and turning in place. Note that this speed is virtual only; while a path which includes two seconds of idling would have an additional 1m added to its length during the search for the shortest path, if selected its true length would be used to calculate the path efficiency ratio.

A poor score on the Path Efficiency metric usually results from the most direct route between the start and end locations passing through areas of very limited coverage (and, hence, very limited path options).

5.4 Action Efficiency

This metric measures the efficiency overhead required to execute a desired action, such as picking up an object lying on the floor across the room. The metric is measured in a Monte Carlo fashion exactly analogous to that described for Path Efficiency:

$$AE_a = \frac{pathLength_a}{minPathLength} \tag{9}$$

where a is the action type in question, *minPathLength* is the length of the reference path computed exactly as per the path efficiency metric, and $pathLength_A$ is the length of the shortest path through the embedded graph that ends in a clip which executes an instance of action a in the end location.

High values of this metric as compared to the Path Efficiency value for the same $(start, end)$ pair typically represent actions which are inflexibly linked into the motion graph, such as a ducking motion which can only be accessed after running straight for several meters, and which are coupled with or consist of motions which

do not fit well into regions of the environment (such as a region requiring twisty paths between obstacles).

5.4.1 *A* Planner.* Our A* planner uses the following as a conservative distance estimate:

$$D_{est} = D_{S \to X} + ||X - E|| \tag{10}$$

where D_{est} is the heuristic estimate of distance from the start point S to the end point E, $D_{S \to X}$ is the length of the path currently being examined, and $||X - E||$ is the linear distance between X, the end of the current path, and E.

Note that our A* path planner returns the optimal path between the specified start and end locations; applications which use a different style of planner which sometimes returns sub-optimal paths will have commensurately-worse path-based metric results. In the more demanding environments, some of the optimal paths initially move a substantial distance away from the end location, so a path planner with a limited horizon would generate paths that were longer or even substantially longer than optimal, which would result in a worse score on both the Path Efficiency and the Action Efficiency metrics.

5.5 Local Maneuverability

Inspired by the idea of local controllability [Luenberger 1979], the local maneuverability metric is designed to evaluate the responsiveness of the character to interactive user control. In many interactive applications, a highly responsive character is critical; in a hazardous-environment training scenario, for example, a significant lag in character response would not only be immensely frustrating and interfere with training, but could actually teach bad habits; for example, if a character is unable to transition rapidly into evasive behaviors, users may simply learn to avoid using such behaviors, even in situations where they would be appropriate.

The instantaneous local maneuverability of a character is simply the mean amount of time required for that character to perform any other action which is currently valid. At a particular moment, the instantaneous local maneuverability of the character with regard to action k is:

$$LM_k(t) = (1 - \alpha(t)) * D_{c_0} + MDP_k \tag{11}$$

where c_0 is the currently-playing clip, $\alpha(t)$ is the fraction of that clip already played at time t, D_c is the duration in seconds of clip c, and MDP_k is the shortest possible time to reach an instance of motion type k from the end of the current path while following paths from the motion graph. For example, if the character is 0.3s from the end of a running clip and the minimum-time path from the end of that clip to an instance of punching is 1s, then the character's Local Maneuverability with respect to punching is 1.3s.

The character's overall Local Maneuverability is the weighted sum of its per-action Local Maneuverabilities:

$$LM(t) = \frac{1}{||K||} \sum_{k \in K, k \neq T_{c_0}} (w_k * LM_k) \tag{12}$$

where K is the set of actions which are in the motion graph and currently valid, and w_k is the weight for action type k. This gives an overall measure of the character's ability to respond to external control, taking into account the different reactivity needs of the different motion types (i.e., evasive actions such as ducking may need to be available much more rapidly than actions such as picking up an object).

There are two ways to measure expected overall local maneuverability. The first is *Theoretical* Local Maneuverability, which is measured in the original motion graph:

$$LM_{T,k} = \frac{1}{||C||} \sum_{c \in C} (0.5 * D_c + MDMP_{c,k}) \tag{13}$$

where C is the set of all clips in the motion graph and $MDMP_{c,k}$ is the duration in seconds of the minimum duration path through the motion graph from the end of clip c to any instance of motion type k. This gives

a baseline for what instantaneous local maneuverability a character can be expected to have at any time under ideal conditions (i.e., the environment does not invalidate any of the minimum-duration paths). Poor theoretical local maneuverability values typically indicate poor connectivity between motion types in the motion graph.

By contrast, in-practice or *Practical* Local Maneuverability computes the expected instantaneous local maneuverability from each node of the *embedded* graph:

$$LM_{P,k} = \frac{1}{||G||} \sum_{n \in G} (0.5 * D_{c_n} + MDEP_{c_n,k}) \tag{14}$$

where G is the set of nodes in the embedded graph, c_n is the clip associated with embedded graph node n, and $MDEP_{c,k}$ is the duration in seconds of the minimum duration path through the embedded graph from the end of clip c to an instance of motion type k. This gives an expectation for what instantaneous local maneuverability a character can be expected to have at any time when navigating through the environment in question. Comparing this to its theoretical equivalent can provide information about the restrictions the environment places on responsiveness to user control.

Note that computing local maneuverability for every node is computationally expensive; in practice, we use a subset of nodes selected uniformly at random (i.e., 1% sampling means any particular node was used in the calculation with probability 0.01). The stability of the result at different sampling densities is examined in Section 6.5.

Finally, note that local maneuverability can be computed for many slices of the data; of note are:

—Computing the local maneuverability for a single action which needs to be rapidly accessible at all times (such as evasive ducking).

—Computing the local maneuverability from one subset to another, such as from locomotions to all evasive maneuvers.

—Computing the local maneuverability from locomotions to same-type locomotions with the character's facing turned more than N degrees clockwise from the starting facing, in order to gain a local measure of the character's ability to respond rapidly to a user's navigational directions (such as "run into that alcove to the left to avoid the boulder rolling at you").

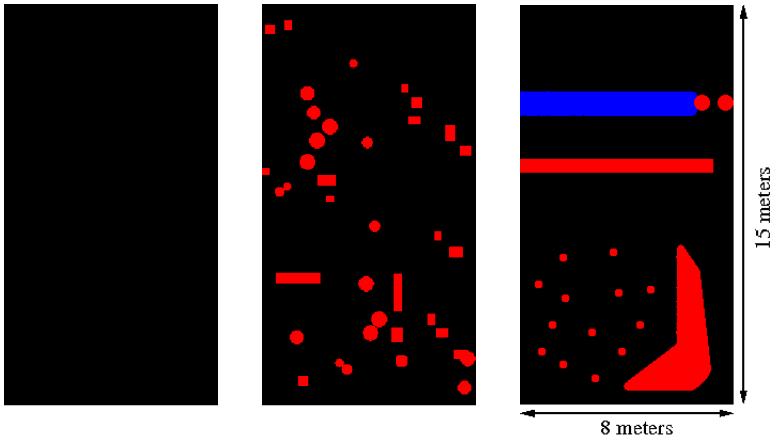
—Computing statistical information on any of the above by examining the distribution of local maneuverability values over all nodes. Information about extremal values, such as how often it would take greater than 3 seconds to reach an evasive motion, can be particularly informative.

6. RESULTS

6.1 Example Scenarios

The two components of a test scenario are the motion graph used and the environment in which it is embedded. Our primary test environment was a large room with varying levels of clutter (Figure 11(c)). This room measures 15m by 8m, and our primary testing resolution used a grid spacing of 20cm by 20cm for the character's position and 20 degrees for the character's facing angle. We include results from other environments, including tests on randomly-generated environments of varying sizes and configurations, and other discretization grid resolutions. To take into account obstacles, we do collision detection with a cylinder of radius 0.25m around the character's root, although more exact collision detection algorithms (e.g., [Gottschalk et al. 1996]) could be plugged in seamlessly; hence, all obstacles can be described as 2D shapes with annotations.

All motion graphs used were computed using motions downloaded from the CMU motion capture database (mocap.cs.cmu.edu). Our dataset consisted of 50 motion clips comprising slightly over seven minutes of cap-



(a) Empty (b) Random (c) Baseline

Fig. 11. Evaluation environments of increasing complexity. The upper obstacle (blue) in environment (c) is annotated as a chasm or similar obstacle, meaning it can be jumped over.

tured motion, with significant amounts of unstructured locomotion (running, sneaking, and idling), several examples each of several actions (jumping, ducking, picking up an object from the floor, punching, and kicking an object on the floor), and transitions between the locomotions and some of the (locomotion,action) pairs (sneak+duck, run+duck, run+punch, run+kick, sneak+pickup, etc.). Each source motion was labeled with the types of motions it contained, as well as the start, center, and end times of actions.

The per-pair values of adding a transition between any pair of frames were computed using the technique of Lee et al. [2002]. The resulting matrix of transition costs was processed in a globally-greedy manner to identify local maxima (see Appendix B) and to enforce user-specified minimum and maximum clip lengths. Our primary motion graph consisted of 98 nodes (clips) connected by 288 edges, representing 1.5 minutes of motion capture data with eight distinct types of motion. We also include results from larger motion graphs containing additional examples of the same types of behavior.

Our embedding algorithm was implemented in Java and used a simple hash table of fixed size to cache nodes' edge lists during metric evaluation, with hash table collisions being resolved by a simple hit-vs-miss heuristic. Running on a 3.2GHz Xeon computer, finding the final (augmented) embedded graph required about 6.7 minutes and produced an embedded graph with 218K nodes and 1.88M edges.

When playing back motion through the embedded graph, transitioning from the end of motion clip A to the start of clip B was done by warping the first frame of clip B to match the last frame of clip A, and then using quaternion splines to smoothly remove that warping over the following 0.5s. Motion editing to warp clips to grid centers was done using displacement splines on the root translation and facing angle. This editing technique of course creates footsliding artifacts, which should be cleaned up in post-processing.

6.2 Baseline

Table I shows evaluation results from our basic test scenarios. XZ Coverage is the fraction of collision-free (X,Z) grid cells in the environment which contain at least one node of the embedded graph; XZA Coverage is the analogous quantity taking into account character facing (see Section 5.1). Local Maneuverability (Section 5.5) is the minimum-duration path from a running motion to an instance of the "pick" action in either the original motion graph (Theoretical LM) or the embedded graph (Practical LM). Path Efficiency is the ratio of the distances of the shortest point-to-point path in the embedded graph vs. an ideal reference

path (Section 5.3), and Action Efficiency is the ratio of the distances of the shortest path ending in a "pick" motion vs. the same ideal reference path (Section 5.4). All of these paths are optimal for their particular start and end locations, so the Median Path Efficiency, for example, is the efficiency of the *optimal* path for a typical pair of start and end points. Accordingly, any scenario with poor Path Efficiency has a poor value for that metric in the optimal case, so a path at "90% Path Efficiency" means the *optimal* path for that particular start and end location was less efficient than the optimal paths for 90% of the other $(start, end)$ pairs chosen.

Results for the environment with no obstacles are excellent, suggesting that the process of discretization and analysis does not unduly affect the capabilities of the motion graph. Note, however, how Practical Local Maneuverability and Action Efficiency show mobility is slightly affected, largely by the walls surrounding the environment removing some path options.

The environment with randomly-placed obstacles is relatively open, but represents a much more realistic environment, with more interesting evaluation results. XZ Coverage is still high (94.3%), but XZA Coverage is much lower (66.5%), reflecting congested regions which the character can only traverse along a single axis. Median Path Efficiency is still very close to the optimum, but the mean path is over 20% longer than the reference path, meaning that some regions of the environment are difficult for navigation and require very long paths. This is reflected more strongly in the sharply higher values for Practical Local Maneuverability and especially for Action Efficiency. The latter is especially interesting; even the median paths were much (85%) longer than the reference paths, suggesting that the motion graph requires a sizeable open region to efficiently set up and use motions such as "pick"s.

The performance of the default motion graph in the Baseline Environment – the most obstacle-dense of the three – is quite poor, especially with respect to having the character use specific actions in specific parts of the environment. The high XZ Coverage value (95.7%) indicates that the character can reach almost any point in the environment; however, the lower XZA Coverage value (60.6%) – reflecting restrictions on the character's facing at many points in the environment – indicates the character may have a restricted ability to maneuver at many points. We see that most point-to-point paths through the embedded graph are still relatively efficient (median was 11% longer than reference), although interactions with highly congested regions of the environment cause a significant number of exceptions (Figure 12).

By contrast, the mean time from any frame to the nearest "pick" action in the embedded graph is almost doubled (3.6s to 6.6s) from its already-high value in the original motion graph, and is substantially worse than the equivalent measurement in the relatively-cluttered Random Environment. Accordingly, we observe substantial difficulty in creating efficient point-to-point paths which end in a specific action, such as a "pick" motion; even typical paths (Figure 13) are about 160% longer than the ideal reference paths (see Table I, Action Efficiency (Median)).

Of the discrete actions available in our motion graph (i.e., pick, duck, kick, punch), the range of areas in which each could be used varied widely (see Figure 14). In addition, a slight modification to the Baseline Environment radically changed the coverage pattern (see Figure 15).

Section 6.3 examines several potential approaches for improving the performance of motion graphs in this environment. Section 6.4 explores how this evaluation approach scales with increasing size and complexity of scenarios, as well as the effects of our on-demand edge computation algorithm. Finally, specifying a set of metrics required making several assumptions, the validity of which are examined in Section 6.5.

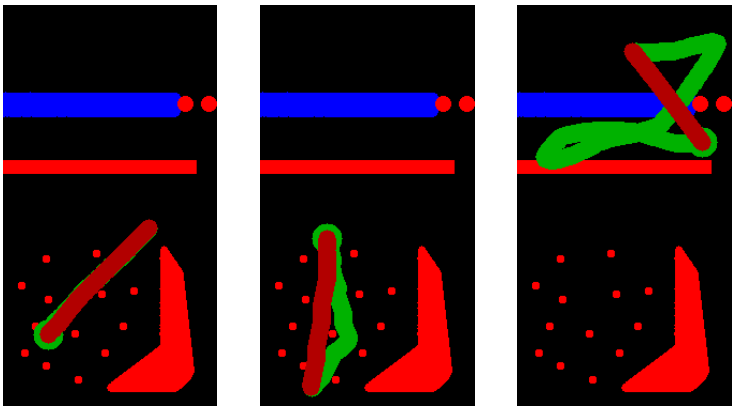
6.3 Improving Motion Graphs

We examine the effects of three common techniques which can be used to improve the flexibility of a character's performance when using a motion graph:

(1) Adding more motion data

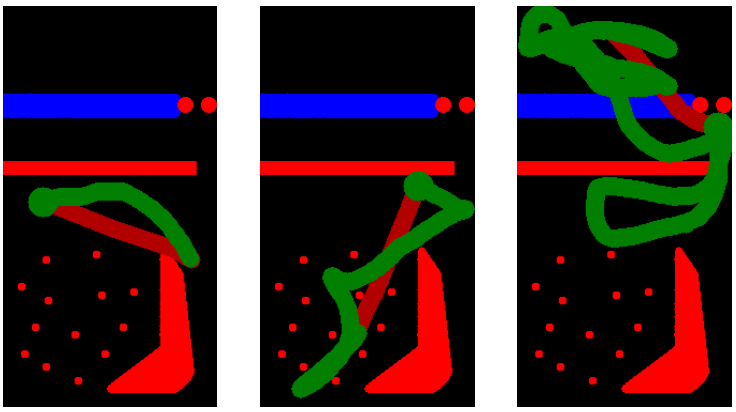
Environ	Coverage(%)		Local Maneuv		Path Efficiency		Action Efficiency	
	XZ	XZA	Theory	Prac	Mean	Median	Mean	Median
Empty	99.8	94.1	3.6s	4.4s	1.00	1.00	1.25	1.11
Random	94.3	66.5	3.6s	5.6s	1.21	1.03	1.93	1.85
Baseline	95.7	60.6	3.6s	6.6s	1.87	1.11	2.85	2.59

Table I. Evaluation results for the three basic test scenarios. Capability steadily worsens as the environments become more congested with obstacles, as shown by the across-the-board increase of all metric values.



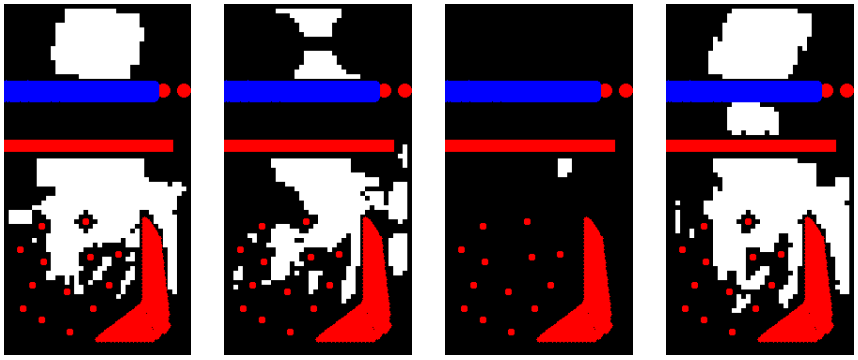
(a) Good path (10th percentile) (b) Median path (50th percentile) (c) Bad path (90th percentile)

Fig. 12. Representative character navigation paths in the baseline scenario. The best paths actually available to the character are in green, ideal reference paths are in dark red. The starting point of the actual path is drawn slightly widened. A 10^{th} percentile path means that, of all the (start,end) pairs tested, only 10% resulted in a better Path Efficiency metric score (i.e., had a ratio between optimal path actually available to the character and ideal reference path that was lower).



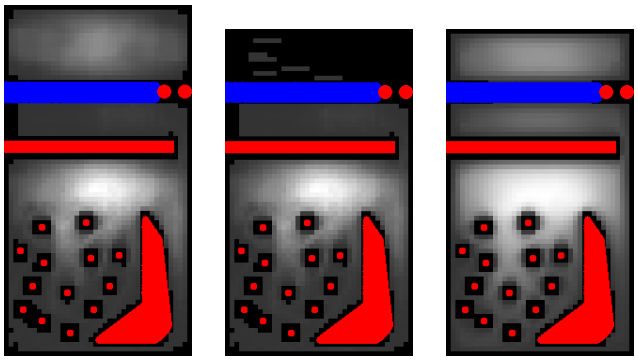
(a) Good path (10th percentile) (b) Median path (50th percentile) (c) Bad path (90th percentile)

Fig. 13. Representative character pick paths in the baseline scenario. Best-available paths ending in a pick action are in dark green and ideal reference paths are in dark red. The starting point of the actual path is drawn slightly widened. A 10^{th} percentile path means that, of all the (start,end) pairs tested, only 10% resulted in a better Path Efficiency metric score (i.e., had a ratio between optimal path actually available to the character and ideal reference path that was lower).



(a) Duck coverage (b) Kick coverage (c) Punch coverage (d) Pick coverage

Fig. 14. XZ Coverage locations for ducking, kicking, punching, and picking-up in the baseline scenario. Areas where the action can be performed are marked in white. Section 7.2 examines the poor coverage of the punching action.



(a) Baseline (b) Shortened (c) Larger motion graph

Fig. 15. (a) XZ Coverage for the baseline scenario. (b) XZ Coverage for a version of the baseline scenario with the environment shortened from 15m to 14m. (c) XZ Coverage for the shortened environment with a larger motion graph. Areas through which the character can navigate are marked in grey, with brighter grey corresponding to more ways to navigate through the area.

(2) Allowing more motion editing

(3) Increasing inter-action connectivity

6.3.1 *Larger Motion Graphs.* Table II shows the evaluation results for motion graphs of approximately 200%, 300%, and 400% the size of the baseline motion graph (first row). Larger motion graphs were created from the original dataset by allowing progressively lower-quality transitions to be included in the motion graph. Larger motion graphs allow more flexibility in creating motions, and improve the metrics across the board; however, no substantial improvement to the efficiency of paths ending in a "pick" action was noted until the size of the motion graph had tripled from the original, and adequate performance was not obtained until a motion graph of quadruple size was used.

In addition, even the motion graph of 200% baseline size allowed sufficient flexibility to navigate efficiently within the Baseline Environment, and to restore the capability lost in the shortened environment (see Figure 15).

Unfortunately, our primary method of generating larger motion graphs – by lowering the threshold for acceptable transitions when creating the motion graph – tended to lower the overall quality of motions

	Coverage(%)		Local Maneuv		Path Efficiency		Action Efficiency	
Clips	XZ	XZA	Theory	Prac	Mean	Median	Mean	Median
98	95.7	60.6	3.6s	6.6s	1.87	1.11	2.85	2.59
190	98.3	90.3	3.3s	4.6s	1.11	1.01	2.36	2.10
282	98.3	91.5	3.6s	3.9s	1.06	1.01	1.47	1.37
389	98.3	95.5	2.6s	3.0s	1.02	1.00	1.20	1.11

Table II. Evaluation results by size of motion graph (see Section 6.3.1). All metric values show substantial improvement, although different tasks improve at different rates.

Edit	Coverage(%)		Local Maneuv		Path Efficiency		Action Efficiency	
Size	XZ	XZA	Theory	Prac	Mean	Median	Mean	Median
75%	40.5	17.2	3.6s	10.3s	2.67	2.67	5.12	5.39
88%	51.5	28.9	3.6s	9.0s	1.54	1.24	3.29	3.31
100%	95.7	60.6	3.6s	6.6s	1.87	1.11	2.85	2.59
112%	96.8	69.4	3.6s	5.9s	1.55	1.05	2.34	1.89
125%	97.2	75.9	3.6s	5.7s	1.25	1.02	1.98	1.48
150%	98.3	84.0	3.6s	4.9s	1.18	1.02	1.61	1.19

Table III. Evaluation results for the baseline scenario with different sizes of editing footprint (see Section 6.3.2). All metric values improve, but not necessarily to acceptable levels (e.g., Local Maneuverability).

generated from that motion graph, due to the larger amount of editing needed to smooth over the lower-quality transitions. In practice, some of the paths generated from the largest motion graph were very efficient and looked good at a global level, but contained poor-quality motion and looked poor at a more local level, suggesting that increased capability from lower transition thresholds should be balanced against this potential quality loss.

6.3.2 *Increasing Allowed Editing.* Table III shows the evaluation results for the baseline motion graph with varying assumptions about the maximum allowable amount of motion editing. The results show that a minimum level of editing is necessary to form a well-connected embedded graph. After that point, however, capability improves at a decreasing rate. Even the highest level of editing does not adequately resolve the problems with poor Practical Local Maneuverability and Action Efficiency, with transitioning from running to a "pick" action taking a mean of almost 5 seconds, and paths ending in a "pick" being over 60% longer than the ideal reference paths.

While allowing more motion editing permits greater coverage of the environment, in practice there are limits on the extent to which a motion can be edited while remaining of high enough quality to meet the requirements of the application. When editing size is larger than 125% of baseline, we observe unacceptable footsliding artifacts in the resulting motion.

6.3.3 *User-Modified Motion Graphs.* Table IV compares the baseline motion graph with a small but densely-connected hub-based motion graph which was carefully edited to have highly-interconnected motion types in a manner similar to that of Lau and Kuffner [2005].

While the hub-based motion graph has superior Local Maneuverability, its Path Efficiencies are only slightly better than that of the baseline motion graph. In general, the hub-based motion graph did not offer as much improvement as expected, some reasons for which are examined in Section 7.2.

Motion Graph	Coverage(%)		Local Maneuv		Path Efficiency		Action Efficiency	
	XZ	XZA	Theory	Prac	Mean	Median	Mean	Median
Baseline	95.7	60.6	3.5s	6.6s	1.87	1.11	2.85	2.59
Tuned	93.0	40.0	2.9s	5.2s	1.51	1.17	2.48	2.39

Table IV. Evaluation results for the Baseline Environment with either the baseline motion graph or a smaller, hub-based one (see Section 6.3.3). The hub-based motion graph has better capability by most measures, but by a surprisingly small amount.

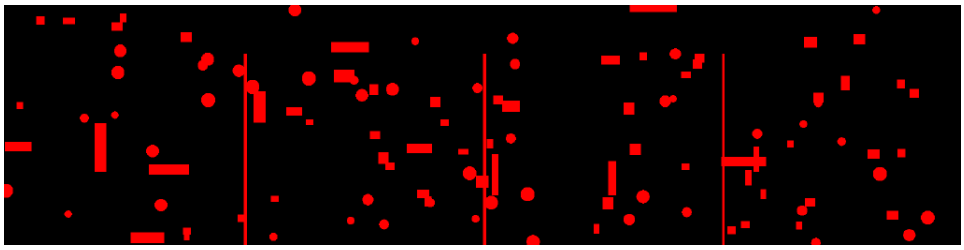


Fig. 16. 40m by 10m environment partitioned into rooms by 8m-long walls and populated with random obstacles. 10m by 10m, 20m by 10m, and 80m by 10m environments of the same format were also used to evaluate scaling behavior of the algorithm in equivalently-dense environments.

	Time(s)			Edges		Memory Used	
Area(m^2)	Total	Embedding	Metrics	/sec	$/m^2$	Base	Peak
56	612	163	470	3.5k	10k	182MB	204MB
120	1,769	742	1,010	5.8k	36k	182MB	283MB
240	4,709	2,137	2,550	8.2k	73k	183MB	382MB
1,000	24,337	15,667	8,653	8.3k	130k	190MB	361MB
5,000	417,075	274,052	142,955	10.2k	560k	222MB	542MB

Table V. Time and memory requirements for evaluation of environments of different sizes. "Edges/sec" is the number of edges in the embedded graph divided by the total embedding time. For comparative purposes, an NBA basketball court is approximately $430m^2$, and an NFL football field is approximately $5,300m^2$; evaluation of either would fit comfortably within the memory of a typical desktop PC.

6.4 Scaling Behavior of the Methods

Realistic scenarios often include large environments with complex motion graphs, and a practical method for evaluating motion graphs must be able to scale to meet these demands. We examined the scaling behavior of the evaluation method in terms of increasing demands of several different types.

6.4.1 *Scaling with Increasing Area.* Table V shows the time and memory requirements for evaluating environments of different sizes. Embedding Time is the total computation time required to obtain the strongly connected component of the final embedded graph. Metrics time is the computation time required to compute all metrics. Base memory is the memory use reported by the Java Virtual Machine (JVM) after the graph embedding. Peak memory is the maximum memory use reported by the JVM; path search is the main contributor to the difference between peak and base memory, with some effect from dynamic edge caching. Each environment was populated with randomly-selected and randomly-placed obstacles in the manner of the Random Environment (Figure 11(b)) of a size appropriate to the environment (i.e., larger environments tended to have both more and larger obstacles). Graph embedding dominated the running time of larger environments, increasing from 28% of processing time for the smallest environment to 66% for

Size(m)	Embedding Time(s)	Time(s) $/m^2$	Edges /sec	Edges $/m^2$
10x10	1,007	10.5	3.7k	38k
20x10	2,190	11.0	3.5k	39k
40x10	4,076	10.2	3.8k	39k
80x10	9,571	12.0	3.7k	44k

Table VI. Time to compute the embedded graph is approximately linear with area across environments with equivalent obstacle densities. (Additional "fake obstacles" (which had no effect on edge validity) were added to the smaller environments to equalize collision-detection overhead between environments.)

the largest.

In this experiment, runtime scaled approximately quadratically with area, but actually decreased on a per-edge basis ("Edges/sec" column in Table V). The apparent reason for this quadratic scaling with area is that the size of each obstacle increased with the size of the environment, making successive environments closer and closer approximations of an unconstrained environment; accordingly, the size of the embedded graph (and hence the computation time) increased faster than the size of the environment ("Edges/m^2" column).

To factor out the effects of different obstacle densities, we examined a set of environments designed to have approximately equal obstacle density (see Figure 16). All environments in the set were 10m long, with widths of 10m, 20m, 40m, and 80m. Each environment was partitioned into 10m by 10m sections by a series of narrow 8m-long walls (so movement between sections was possible only through the 2m gaps beside the walls); these partitions made the larger environments approximate a series of the smallest 10m by 10m environment, rather than having large regions away from the constraining influence of side walls. Each environment was populated with an equal density of random obstacles drawn from the same population (i.e., comparable sizes). Finally, each environment contained enough "fake" obstacles (requiring collision tests but not affecting edge validity) to bring the total obstacle count up to the number of obstacles present in the largest environment. These fake obstacles made per-edge collision detection costs comparable across the environments.

By contrast with the largely unconstrained environments seen previously, embedding time for these environments divided into "rooms" scaled close to linearly with area (Table VI). The partitioned environments maintain approximately the same embedded graph density at all sizes, and hence their computational requirements scaled linearly with area, as the theoretical analysis suggested. We note, however, that there was a small upwards trend in embedded graph density and per-m^2 embedding time (about 15% over an 8x size increase), suggesting that even the small 2m openings between adjacent rooms may have slightly changed the obstacle density of the environments. We conclude that different configurations of obstacles – such as long, thin walls partitioning the interior of a building vs. small, scattered bushes or tree trunks in a field – may have qualitatively different effects on the capability of an animated character.

Finally, we note that the embedded graph in the largest environment evaluated (100m by 50m) contained 122,368,278 nodes and 2,798,837,985 edges. Explicitly storing this embedded graph would have required approximately 11GB of memory; computing it via the method of Reitsma and Pollard [2004] would have required approximately 50GB, as compared to the 0.5GB used by our method.

6.4.2 *Scaling with Increasing Motion Graph Size.* Table VII shows the time and memory requirements for evaluating the Baseline Environment with motion graphs of various sizes. Larger motion graphs were formed from the same database of motions by progressively lowering the threshold for acceptable transitions. Total running times ranged from under half an hour to over six hours, with metric evaluation taking approximately 75% of the overall time. Much of the heightened requirements were due to the path searches through the

Motion Graph	Time(s)			Memory Used	
	Total	Embedding	Metrics	Base	Peak
87s	1,613	399	1,201	182MB	200MB
163s	10,616	1,602	9,001	352MB	594MB
248s	19,214	4,014	15,185	523MB	995MB
350s	29,519	8,204	21,303	741MB	1,246MB

Table VII. Time and memory requirements for embedding different sized motion graphs in the Baseline Environment. Increasing the size of the motion graph was one of the most effective ways to improve its capability, but also one of the most costly in terms of resources required for tasks such as path search.

large motion graphs required for the Efficiency metrics, as A* search is exponential in the branching factor, which tends to be higher in larger motion graphs.

6.4.3 *On-Demand Edge Computation Overhead.* Table VIII shows the time and memory requirements for evaluating environments using different levels of edge caching. Default caching places edge-lists into a very simple hash table of fixed size, using usage-frequency information to determine which list to keep in the event of a collision. Full caching stores all edges prior to metric computation, so stored edges are included in base memory. Explicit caching computes all candidate edges and uses Depth-First Search to find the embedded graph as in [Reitsma and Pollard 2004].

Explicit computation of the embedded graph rapidly becomes intractable due to the memory requirements. Moreover, the minimum possible time such an algorithm could take – conservatively estimated by adding together the time to find the edges, find the largest SCC (without augmentations), and compute the metrics with fully-cached edges, but ignoring any other necessary operations – is at best lower than the flood-based algorithm used in this paper by a small constant factor, due to the smaller number of nodes expanded by the flood-based algorithm (see Section 4.2.2).

For the flood-based algorithm, the three caching regimes tested offer a tradeoff between storage space and computation time. Caching all edges in the embedded graph before computing the metrics is only a small amount of memory overhead for small scenarios and results in almost a 50% reduction in computation time, but memory requirements become increasingly costly as environments or motion graphs grow larger, for increasingly smaller gains in computation time (only 30% for the larger environment). Caching frequently used edges (the default) provides a tradeoff that maintains most of the speed benefits for only a fraction of the memory.

6.5 Validity of the Evaluations

While the results of these metrics seem intuitively reasonable, one of the key goals of this work was to reduce the reliance on animator intuition by providing objective and verifiable evaluations.

Past a minimum threshold resolution, the metrics are relatively stable, with evaluation results steadily but slowly improving with increasingly fine resolution; however, the problems identified in the default resolution remain in the finer resolution analyses. This trend suggests that the baseline evaluation at which our initial tests were performed should be a reasonable (if pessimistic) estimate of the metric values that would be obtained in the limit as grid sizes went to zero. While drastic improvements in the metric values were not observed in our experiments, such improvements are possible, making this approach a conservative estimate.

In addition, while Local Maneuverability results are only given for the running-to-pick-up action transition and Action Efficiency results are only given for paths ending in a picking-up motion, several other actions were tested (run-to-duck, sneak-to-duck, paths ending in a duck, etc.), with substantially similar results to those reported for picks. Similarly, the Path Efficiency and Action Efficiency metrics gave essentially the same results regardless of whether path-planning was unbounded or limited to a generous horizon. For

Cache	Area(m^2)	Time(s)		Memory Used	
		Total	Metrics	Base	Peak
None	56	992	822	182MB	200MB
Default	56	612	437	182MB	204MB
Full	56	504	324	191MB	209MB
Explicit	56	499+	324	572MB	590MB
None	120	2,791	2,118	182MB	258MB
Default	120	1,736	1,068	182MB	289MB
Full	120	1,412	690	235MB	306MB
Explicit	120	1,181+	690	1,064MB	1,135MB
None	240	6,934	4,076	183MB	325MB
Default	240	5,496	2,639	183MB	382MB
Full	240	4,307	1,302	357MB	498MB
Explicit	240	3,192+	1,302	2,070MB	2,211MB
None	1,000	41,780	15,117	189MB	274MB
Default	1,000	30,027	9,870	189MB	361MB
Full	1,000	27,859	5,954	1,305MB	1,390MB
Explicit	1,000	13,747+	5,954	9,832MB	9,917MB

Table VIII. Time and memory requirements for different edge-caching schemes. All times are in seconds. Explicit caching refers to the algorithm of Reitsma and Pollard 2004. The "+" notation is used as we replicated only the first parts of their algorithm for comparative purposes; some additional computation beyond the amount timed is required by their algorithm. The caching scheme used allows a tradeoff between space and time requirements for the evaluation, although a simple caching scheme achieved most of the benefits of full caching while keeping the memory footprint low: the running time of our on-demand algorithm is within a small constant factor of what could be achieved if the embedded graph were built explicitly, while the memory required grows at an asymptotically lower rate. (The Metrics column corresponds to the relative performance one would expect when using the embedded graph (e.g., searching for paths).)

efficiency, both metrics were run with a horizon of six times the length of the reference path; failure to find a path within that distance resulted in an effective path length equal to the horizon distance for metric purposes.

Practical Local Maneuverability was tested at sampling densities ranging from 100% (i.e., the true value) down to 0.03%. The computed values were highly stable past approximately 1% sampling density, validating our probabilistic sampling approach. The baseline at which our evaluations were performed was 10 samples per square meter of the environment, or about 2.3% sampling density for the Baseline Environment.

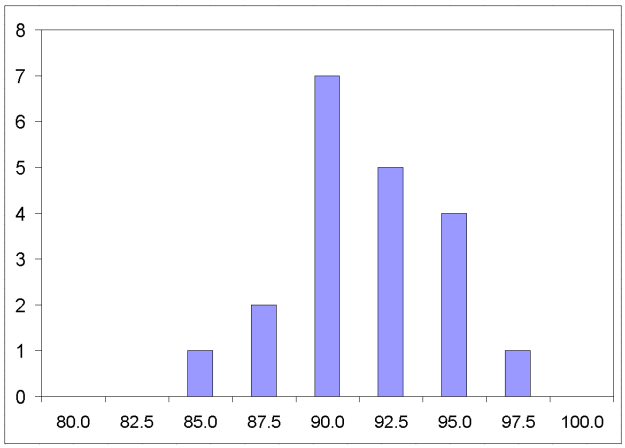
Path Efficiency and Action Efficiency metrics were run with between 50 and 500 samples, as compared to the baseline of 150 sample paths. Metric results were stable across different numbers of samples, suggesting that the default number of samples evaluated should provide a reasonable estimate of the true metric values.

6.5.1 *Measurement Stability Across Different Environments.* Table IX gives the distributions of evaluation results for the baseline motion graph run in 20 randomly-generated environments. The area ($120m^2$) of the environments and types of obstacles placed were the same as those in the Random Environment used for initial testing (see Figure 11(b)), although configuration details (height/width of environment as well as number/size/shape/placement of obstacles) were determined randomly. Figure 18 shows the XZ Coverage overlaid on three representative environments.

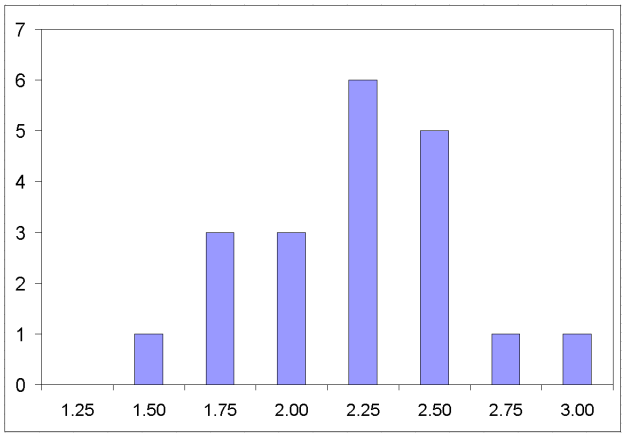
In contrast to the drastic changes seen with the shortened Baseline Environment (Figure 15), results were relatively stable across different environment configurations, with no drastic changes in the capability of the motion graph. As Figure 17 shows, the distribution of values for metrics evaluated on the randomly-generated environments was approximately normally distributed. This result suggests motion graph capability will tend to be strongly correlated across many classes of similar environments.

Total Times(s)		XZ Cvgs(%)		Prac LM		Path Efficiencies		Action Efficiencies	
Mean	StdDev	Mean	StdDev	Mean	StdDev	Mean	StdDev	Mean	StdDev
1,535	260	90.3	3.1	5.8s	0.55s	1.25	0.11	2.11	0.36

Table IX. Mean values and standard deviations for evaluation results of randomly-generated environments. Environments with similar types and densities of obstacles tend to induce similar levels of capability on motion graphs.



(a) XZ Coverage



(b) Action Efficiency

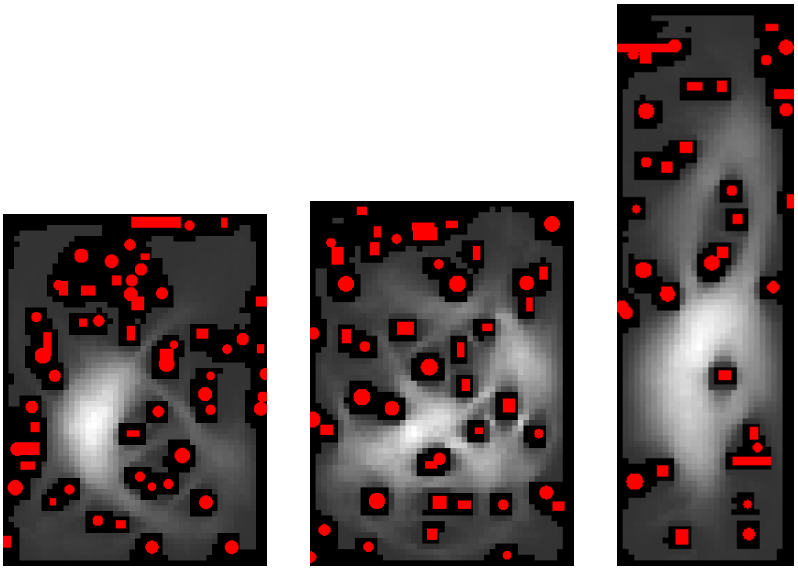
Fig. 17. Distributions of a representative pair of metric values for the random environments evaluated. Y axis is count of number of environments in each bin. Metric values were distributed approximately normally.

7. DISCUSSION

7.1 Findings

When using regular motion graphs on general environments, our experiments identified several problems.

7.1.1 *Effects of Complexity.* One fundamental result of our experiments is the observation that when working with a motion graph data structure, *easy tasks are easy; more complex tasks are hard.* For a simple task, such as basic locomotion in an obstacle-free environment, all motion graphs we tested performed well,



(a) Difficult environment (b) Typical environment (c) Easy environment

Fig. 18. Representative examples of the random environments created for testing, with XZ Coverage displayed in grey.

with extremely good coverage and average paths under 1% longer than ideal reference paths.

Making either the task or the environment more complex, however, had a substantial effect on the capabilities of the motion graph. More complex tasks, such as transitioning to a particular action type or performing a particular action type at a designated location in the environment, were more difficult to perform in even the obstacle-free environment, with Local Maneuverability and Action Efficiency values both higher by about 20% (Table I). Moreover, both measures degraded rapidly even in the relatively-open random environments, and especially in the highly-congested Baseline Environment. Similarly, even basic locomotion suffered in the relatively-open random environments, going from near-perfect efficiency to an average path length 25% longer than the ideal reference, and locomotion in the Baseline Environment was extremely inefficient, with average paths generated by the baseline motion graph being over 85% longer than the reference paths.

We note that one approach to handling this complexity has been pioneered by Lee and his colleagues ([Lee et al. 2002][Lee et al. 2006]). Briefly, their approach is to collect substantial quantities of motion data interacting with each obstacle or cluster of obstacles in close proximity in the environment, in a sense forming a specialized motion graph for each unique region in the environment. As this technique relies on re-using motions captured for the specific parameters of each object cluster, however, it is unclear how the technique might apply to more general environments.

While motion graphs are effective for simple animations in simple environments, we note that the ability to deal with increased complexity of both task and environment is necessary to make motion graphs a more general tool. In particular, simple tasks in environments with few or no obstacles may not provide an adequate test of a motion generation algorithm's abilities.

7.1.2 *Reactivity.* Even the Theoretical Local Maneuverability of a typical motion graph is quite poor, and embedding the motion graph in even a relatively open environment degrades this measure of reactivity substantially (Table I). For example, changing from a running motion to an evasive action (ducking) in the baseline motion graph took an average of 3.6s even in theory, increasing to an average of 5.8s in the random

environments (Table IX) and 6.6s in the Baseline Environment. Considering that interactive environments such as games and training scenarios will tend to operate at timescales driven by users' recognition and choice reaction times (approximately 0.5s, rising to 1s for unexpected stimuli; see [Green 2000a] [Green 2000b] [Kosinski 2005]), substantial improvements in the ability of motion graphs to react to interactive control are necessary. Ikemoto et al. [2006] examine one possible approach to this problem.

7.1.3 *Embedding.* We note the importance of evaluating the capability of a motion graph within the context of its target environment. Our experiments with variants of the Baseline Environment (Figure 15) show that minor changes to an environment can potentially cause drastic changes in the effective capability of one motion graph – in this case making the upper portion of the environment largely inaccessible – while causing virtually no changes in the effective capability of another motion graph. However, our experiments with randomly-generated environments (Table IX) demonstrate that there is a strong correlation between the capability of a motion graph in environments with similar types and densities of small obstacles. The capabilities of motion graphs in our randomly-generated environments were distributed approximately normally (Figure 17). Based on the results with the Shortened Baseline Environment (Figure 15), however, it appears that smaller numbers of longer or wider obstacles will lead to less predictable capability than the larger numbers of smaller obstacles used in the randomly-generated environments.

7.2 Identified Causes

Our experiments suggested three main problems with the motion graphs tested.

First, many clips, especially those of actions such as picking-up, ducking, or jumping, were effectively "set pieces" – i.e., they were linked into the motion graph in such a way that substantial amounts of specific motion was unavoidable both before and after the action itself. Both instances of punching actions, for example, required about *eight meters* of mostly straight-line movement to perform, resulting in the almost complete inability to perform punches in the Baseline Environment (Figure 14(c)). By contrast, other actions were embedded in linear paths through the motion graph consisting of about 2-4m of movement, which was much easier to place within the environment (Figure 14). Lengthy "set pieces" of motion such as these are not only easily disrupted by obstacles, reducing the coverage of the action in the environment, but also drastically worsen Local Maneuverability.

Similarly, actions of these sorts tend to have only a few instances in the motion graph in order to keep the overall size of the graph down as the variety of actions it contains increases. Unfortunately, the paths corresponding to the instances of these actions are often poorly linked into the motion graph, with the start of the branchless portion of the path typically being accessible from very few (often only two) other clips in the motion graph. In a congested environment, however, the ability to take any particular transition from the character's current location can easily be blocked by obstacles, meaning that the small number of direct paths through the motion graph to the desired action can easily be rendered unavailable. This makes the character's Local Maneuverability unacceptably fragile and variable, as well as contributing to the inefficiency of paths.

Finally, many obstacles in close proximity, such as the lower-left region of the Baseline Environment (Figure 11(c)), create a need for paths with specific patterns of left-and-right turning in order to wend between the obstacles. Motion graphs, due to their heavy reliance on the source data, have trouble substantially changing the curvature patterns of their constituent paths, although some of this is possible during clip transitions.

We examined three common approaches to resolving these problems: adding more data to the basic motion graph; allowing more extensive motion editing at clip transitions; and using a small but highly-connected, hub-based motion graph.

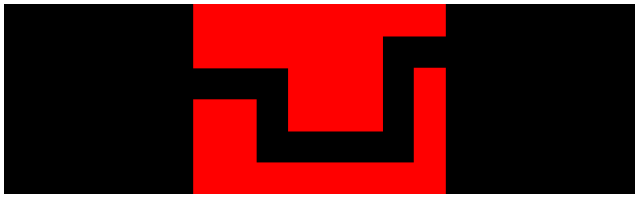


Fig. 19. This environment requires a particular pattern of turns and movement lengths be available in the database, and hence could be hard to navigate with semantically-invariant editing methods.

7.2.1 *Generality from Motion Graph Size.* We note that in our experiments the most effective method for improving the capability of a motion graph was simply to add more data, increasing the number of nodes while keeping the density of the graph roughly constant. The apparent reason for this improvement was the way the increased variety of actions allowed increased flexibility in navigating constrained areas (i.e., higher or lower curvature of turns allows threading between obstacles), as well as the larger number of target nodes in the motion graph for paths constrained to end in a specific actions. Both of these factors made it more likely that an appropriate clip of motion would be available for any given point in the environment.

This had an especially large effect on the ability of the motion graph to efficiently reach specific actions. For the largest motion graph, Practical Local Maneuverability improved sharply to only about 15% higher than Theoretical Local Maneuverability, and Action Efficiency for "pick" actions was only 20% above the ideal (Table II). Reducing the effect of "set piece" actions is only part of this improvement, however, as the shortest of the available "pick" action sets was still 3.0m, vs. 3.7m in the baseline motion graph.

This was also, however, one of the most computationally expensive methods, especially for tasks using the motion graph, such as path-planning. Accordingly, to scale to very large motion databases will require either separating out distinct behaviors as suggested by Kovar et al. [2002] or pursuing an approach that involves clustering and/or multiple levels of resolution, as in the work of Arikan et al. [2003] or Lau and Kuffner [2005].

7.2.2 *Generality from Motion Editing.* Increasing the amount of motion editing permitted had a significant effect on the capability of the motion graph, but did not resolve the problems identified, especially for tasks requiring specific actions: even when allowing 50% more motion editing – a level apt to produce poor animations – Practical Local Maneuverability for "pick" actions was still nearly five seconds, with average paths required to end in a "pick" action being over 60% longer than the ideal reference (Table III).

We note that our motion editing model assumes editing is largely *semantically-invariant*; i.e., edits are conducted purely at a local level, using incremental changes to produce acceptable variations of a source motion. Editing approaches of this type, while commonly used, have limited flexibility. Consider Figure 19; only source motions with the correct number of turns in nearly the correct places can easily navigate this environment when only local edits are permitted. By contrast, more sophisticated editing techniques which can curve paths to suit the demands of the environment could allow greater capability with the same source motions. Recent work by Shin and Oh [2006] and by Heck and Gleicher [2007] on increasing the flexibility of each edge (or node) of a motion graph may be particularly appropriate to threading a character between nearby obstacles.

7.2.3 *Generality from Motion Graph Connectivity.* We note that the densely-connected, hub-based motion graph we used has slightly better performance than the larger automatically-generated motion graph, but that the difference is surprisingly small; both produced poor paths in the baseline and random environments, with poor Local Maneuverability.

One reason for the smaller-than-expected improvement is that the lengths of the available pieces of motion

are not altered by this increased connectivity, and the character must often travel a significant distance before returning to a hub pose and becoming able to transition to a new clip of motion. For example, performing a "pick" action required playing approximately 3.7m of motion in the baseline motion graph before a choice of which clip to transition to next was available, vs. 3.5m in the hub-based motion graph. Due to this, the immediate-horizon ability of the character to navigate through a congested environment is not greatly improved.

Accordingly, a highly-connected, hub-based motion graph which does not address the problem of long "set piece" chunks of motion does not appear to adequately address the deficiencies observed.

One potential solution would be to increase the degree of connectivity not only at the ends of clips, but *inside* clips as well. This approach would allow the character to move between actions such as locomotion with different radius of curvature more freely, permitting a greater ability to navigate in highly-congested environments.

Unfortunately, this approach would substantially increase the size and density of the motion graph. The resulting graph would be significantly harder to reason about, either intuitively or analytically, and would be much more expensive to use for operations such as path planning. Both of these drawbacks undermine some of the important benefits of motion graphs, suggesting the importance of a careful and well-informed tradeoff.

7.3 Scaling Behavior

7.3.1 *Memory.* As noted in the theoretical analysis of memory requirements (Section 4.2.1), base memory required increases linearly with the size of the target environment, but the constant is so small that the amount of memory required is not a bottleneck for our algorithm. The main component of base memory is caching prerotated clips for faster edge computation; however, our results suggest that the overall base memory requirements are reasonable.

Peak memory requirements are driven by the A* search technique used to evaluate the ability of the motion graph to produce efficient paths in the target environment. Due to the exponential nature of A*, this value is highly sensitive to parameters such as the planning horizon, and in practice memory limitations can be significantly lessened by appropriate parameter settings and heuristics. Moreover, applications using motion graphs will by necessity already have a path-planning method in place, and that method will be the most appropriate one to use when evaluating scenarios for that application. As such, peak memory requirements and their solutions are amply discussed in the path-planning and search literature.

7.3.2 *Computation Time.* For larger and more complex scenarios, computation time is typically a stronger concern than memory requirements. For larger environments, computation time is increasingly dominated by the time required to embed the graph in the environment (increasing from under 30% for the smallest environment to over 60% for the largest). In practice, larger environments are often qualitatively different from smaller environments, and their greater openness leads to denser embedded graphs.

By contrast, larger motion graphs continue to have the large part of their runtime (around 70%) come from path search. As with memory requirements, computational requirements for path search and path planning have been a subject of much research, and whatever method is intended for use in the application employing the motion graph can be used efficiently in the embedded graph.

7.4 Alternative Editing Models

The analysis in this paper examined the capability of motion graphs constructed via a concatenation-based approach to motion editing (i.e., clips of source data are edited to fit together seamlessly, and are played one after another to animate the character). This approach typically uses displacement splines or blending (e.g., [Kovar et al. 2002], [Stone et al. 2004], [Kwon and Shin 2005], [Lee et al. 2006], etc.), and can be augmented

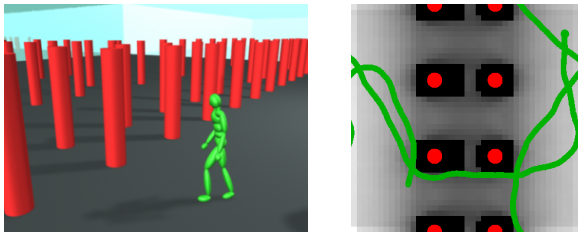


Fig. 20. (Left) The character is following a random path through a motion graph that has been embedded into an environment tiled with a repeating pattern. (Right) A random path wraps around from left to right, bottom to top, then right to left.

with techniques such as splicing (e.g., [Heck et al. 2006]) and timewarping to increase the richness of the database. Minor modifications, however, would allow the approach to be used with alternative types of motion editing, such as controllers (e.g., [Zordan et al. 2005]), interpolation (e.g., [Shin and Oh 2006]), or certain types of spacetime optimization (e.g., [Abe et al. 2004]).

Shin and Oh [2006] and Heck and Gleicher [2007] introduced methods to allow a single motion graph edge (or node) to represent a parametric family of related motions created by blending between discrete source motions. For example, a family of walks with variable turn rates could be created by blending together a set of source walks with different turn rates. Rather than corresponding to a discrete path through the environment, such a family of motions would effectively trace a continuous *path volume* between the start point and the ending footprint (which is itself simply the set of all endpoints of the paths making up the path volume). Points in the path volume would correspond to one instant of a motion which could be generated by blending together source motions; accordingly, a grid point is reachable from the start point by this edge if (a) the grid point is in the footprint of the path volume (i.e., at the endpoint of one of the paths in the family represented by the edge), and (b) the editing method can create some path between the start point and the end point which is contained entirely within the path volume.

Computing the footprint and doing collision detection for this (potentially complex) (x, z, θ) path volume may require substantial computation; however, extending our discretization approach to the path volume should allow efficient computation. The path volume can be precomputed and a discretized version of it stored with the motion graph edge. Collision detection could be performed efficiently by aligning this occupancy grid to the start point and testing each of its cells for validity in the environment. In addition, sampling the path volume with a representative array of precomputed paths, each with a small but finite footprint, would allow an efficient test for path-existence. Each such path could precompute a bit vector representing which cells of the occupancy grid it traverses, and a simple logical AND with a bit vector representing occluded cells of the occupancy grid would determine whether the path was available.

This approach would require additional storage, but only a relatively small amount even for finely-sampled path volumes, as the storage required would depend only on the number of motion families (super-edges or super-nodes) in the original motion graph, rather than on the environment or its discretization.

Controller-based editing typically produces motions that do not deviate too far from the source motion, and could potentially be thought of as creating similar families of motion, as could spacetime optimization-based editing techniques such as [Abe et al. 2004]; accordingly, we expect that a similar treatment would allow this evaluation approach to be applied to either of these motion editing methods.

8. CONCLUSIONS AND FUTURE WORK

The results presented in this paper, along with the earlier version in [Reitsma and Pollard 2004], are to our knowledge the first attempts to evaluate global properties of a motion graph data structure such as ability to efficiently reach all points in an environment and ability to quickly respond to user control in a complex

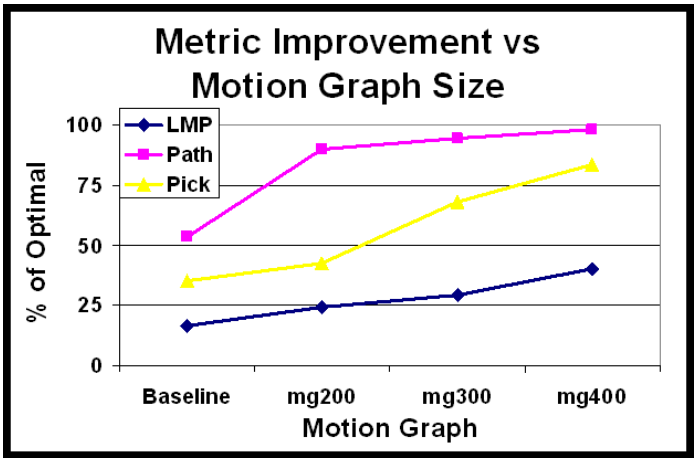


Fig. 21. Increasing the size of a motion graph improves capability for all tasks, but different metrics approach their optimal value at different rates. The Y axis shows how close a particular metric is to its ideal value, while the X axis shows the size of the motion graph (approximately 100, 200, 300, or 400 clips). Simpler tasks, such as efficient navigation ("Path", pink line) approach optimal more rapidly than more complex tasks, such as efficiently picking up an object at a specified location ("Pick", yellow line), suggesting that past a certain size it may be most useful to ignore the unconstrained navigation task and instead target added clips towards improving the picking task. By contrast, tasks such as rapid response to user input (Practical Local Maneuverability or "LMP", blue line) are not improved to near their optimal level by to this improvement method, suggesting that a radically different method, or even an algorithmic change, may be necessary.

environment.

Embedding a motion graph into the environment and assessing global properties of this data structure allows us to compare motion datasets and identify weak points in these motion datasets and problematic areas in the environments. In addition, analysis of a range of motion graphs across a range of environments can shed light on the strengths and weaknesses of the motion graph algorithm in general.

We also provide a method for determining a reasonable answer to the standard question of "how much motion is enough?" Our analysis techniques, coupled with task-based metrics representing the application at hand, can be used to evaluate the capabilities of a motion graph, allowing one to be selected that is neither too big for efficient searching and planning nor too small for effective task performance.

Our experiments highlighted several important considerations regarding generating animations with motion graphs.

First, capability degrades surprisingly rapidly with increasing complexity. Tasks more complex than basic navigation and environments with obstacles in close proximity sharply narrow the options available when path-planning with a motion graph. Accordingly, testing an animation system with only simple tasks or relatively unconstrained environments may not provide the full picture of the system's abilities.

Reactivity for characters animated by motion graphs is often poor even in theory, and is easily degraded further by complex tasks or environments. The ability to react rapidly to user control, however, is crucial to interactive applications, and an inability to move efficiently between motion types lessens the capability of motion graphs with regards to more complex tasks.

Several approaches can help alleviate these problems, including denser motion graphs (perhaps with shorter clips or edges transitioning from the inside of clips), more sophisticated editing techniques, and adding more data to motion graphs. These approaches tend to offer performance tradeoffs, however, increasing capability while increasing the computational cost of using a motion graph.

Our hope is that analysis methods such as the one in this paper will be useful in managing tradeoffs such as these. For example, our analysis revealed that the motion graph improvement methods tended to improve the different metrics at different rates (Figure 21). One potential application of these methods is to direct available memory or computation resources towards those tasks which will generate the greatest capability bang for the resource buck.

In addition, such methods potentially allow a motion graph to be certified as sufficient for a particular environment and set of tasks, or even certified as suitable for a class of environments (e.g., environments whose obstacles do not cluster more tightly than a certain threshold). Additionally, a conservative estimation of capability may allow deemed-acceptable motion graphs to be used with confidence in their target environments.

We see several directions for extensions to this research. A more general scenario could include dynamic environments, multiple autonomous characters, deformable objects, and complex interactions, and we hope to relax some of the simplifying assumptions we have made about the nature of the scenario to be evaluated. In addition, this evaluation approach could be placed in a feedback loop with a motion graph generation system, potentially allowing automatic optimization of motion graphs. As well, we are interested in generalizing the evaluation approach, both to other task domains and to other motion generation algorithms.

Finally, we note that the main bottleneck in the current approach is computation time. One option for reducing this time is to divide the environment into segments, either examining only those segments required (such as in the case of localized changes to obstacles) or forming the environment out of a number of repeatable tiles (see Figure 20; this approach is examined in more detail in [Reitsma and Pollard 2004] and [Lee et al. 2006]). Alternatively, it should be noted that all major steps in the evaluation process are highly parallelizeable, allowing a tradeoff between computation time and required hardware.

In summary, the techniques shown here provide a way to evaluate a character's capabilities that is more sound than the trial and error approach commonly used. Evaluation techniques such as these can help a user to compare alternative data structures and provide detailed information on performance to aid the user in refining their motion graph or motion generation algorithm.

ACKNOWLEDGMENTS

Supported in part by the NSF under grants IIS-0326322, CCF-0343161, and ECS-0325383. Alias/Wavefront donated their Maya software for use in this research.

A. EDITING FOOTPRINTS

The editing model is based on the intuition that the amount a motion can be changed will tend to grow with the distance it covers, as well as with the time spanned by the clip—in essence, we assume an error will be acceptably small if it is small relative to the rate at which other actions are taking place.

We specify the amount that V, the root position and orientation, can be adjusted as a linear function of both distance traveled and number of frames in the clip:

$$abs\,(V_i - V\prime) < (s + \alpha d) \begin{bmatrix} r_x \\ r_z \\ r_\theta \end{bmatrix} \tag{15}$$

where $(V_i - V\prime)$ is the vector difference between the new and original root configurations, s is the arclength of the original motion clip, d is the duration of the motion clip, α is a scaling factor used to weight arc length vs. duration in terms of editing importance, and $(s + \alpha d)[r_x \;\; r_z \;\; r_\theta]^T$ is the size of the ellipsoidal footprint representing allowable edits to root configuration.

The new path taken by the clip is computed by adding the currently-accumulated portion of $(V_i - V\prime)$ to

each frame:

$$V_i(k) = V\prime(k) + \frac{(s(k) + \alpha d(k))}{s + \alpha d}(V_i - V\prime) \tag{16}$$

where $V_i(k)$ is the edited (position,orientation) of the character's root at frame k, $V\prime(k)$ is the unedited (position,orientation) of the character's root at frame k (i.e., corresponding to original endpoint $(p\prime, \theta\prime)$), $s(k)$ is the arclength of the path through frame k, and $d(k)$ is the duration of the path through frame k.

B. TRANSITION SELECTION

To create the motion graphs used in our experiments, we used a globally-greedy algorithm that attempts to coalesce nearby edges into hubs of high-quality transitions.

First, we find the frame-to-frame similarity matrix in a manner similar to Lee et al. [2002]. From this data, we consider only the local maxima (each maximum dominates a 5-frame radius in the matrix), and consider only maxima above a pre-set threshold. This defines a set of *candidate transitions*, and could be used directly to form a motion graph. The resulting motion graph would have no guarantees on minimum or maximum clip length, however, whereas we wished to enforce a minimum length of 0.5s and a maximum of 1.5s.

Enforcing the minimum length is accomplished by *coalescing* nearby candidate transitions; i.e., deleting all candidate transitions within a window of frames and replacing them with transitions to a single frame within that window. The algorithm for this is given in detail below; in brief, the approach is to examine each window of frames (i.e., every set of sequential frames of length 0.5s) and select the optimal frame within that set to reroute all of the transitions in that set through; this frame is given a score equal to the sum of the quality of all acceptable (i.e., above threshold) transitions which will be re-routed through it. Once this value is calculated for all windows of frames, the globally-highest value is selected, the transitions in the corresponding window are coalesced, and the process iterates until no window contains more than one transition (i.e., all clips are at least 0.5s long). A maximum clip length of 1.5s is enforced as a post-processing step by splitting long clips. Although $O(n^4)$ in a naive implementation, the coalescing step can be implemented to incrementally update the window values based only on changes since the last step, making the process run in $O(n^2)$ and, in practice, take only a few hours even for our largest motion graphs.

B.1 Coalescing Candidate Transitions

Repeat
—Find window w with $count(w) > 1$, frame $c \in w$ that maximizes $value(w, c)$
—$coalesce(w, c)$
Until $count(w) \leq 1$ $\forall$ windows w

window(s, f, m): the set of m frames in source file s ranging from f to $f + m - 1$. Here, m is always the minimum clip size, and w will always refer to a window of this sort.

count(w): the number of frames $f \in w$ s.t. $\exists$ frame $g \in DB$ s.t. $equivalency(f, g) > 0$.

DB: the set of frames of the input source motion files

value(w,c): $\sum_g equivalency(c, g)$ $\forall g$ s.t. $\exists f \in w$ s.t. $equivalency(f, g) > 0$. i.e., the value of window w if all transitions must go through frame c.

equivalency(f,g): the similarity of frames f and g, if that value is greater than the acceptance threshold, else 0.

coalesce(w,c): $\forall g \in DB$ s.t. $\exists f \in w$ s.t. $equivalency(f, g) > 0$ recompute $equivalency(c, g)$ and set $equivalency(f, g)$ to 0. i.e., force all transitions inside window w to use frame c or no frame at all.

Motion Graph		
Clips	**Transitions**	**Motion**
98	288	87s
190	904	163s
282	1,712	248s
389	3,198	350s

Table X. Clips, transitions, and total amount of motion contained by the different motion graphs created for our main evaluations. The full motion database contained 439 seconds of motion.

B.2 Crafting Representative Motion Graphs

The frame-to-frame similarity matrix computed above will depend strongly on the precise details of the input motion capture clips. In practice, even a relatively small change in the transition qualities between an input file and other files in the database can result in a substantially different motion graph. If one running motion is replaced by another, for example, or even if the same running motion is used but with a different transition-selection threshold, a very different motion graph can result due to the discrete nature of taking local maxima in the similarity matrix, coalescing them to enforce minimum clip lengths, and taking the largest strongly connected component of the result.

In order to ensure that high-quality and representative motion graphs were used for our tests, each of the 50 input files was assigned a "desirability" weight which directly modified the relative quality of any transitions to or from that motion. These weights were manually adjusted until an acceptable motion graph was obtained. Our baseline motion graph was crafted via these weights to have a representative mix of all available motion types, with an emphasis on the two main locomotions (running and sneaking), a smaller amount of idling motion, and one to four examples of each of the actions (jumping, picking-up, ducking, punching, kicking) in the database. In addition, we ensured that each instance of an action was directly accessible from locomotion (i.e., it was not necessary to pass through a kicking motion to reach a ducking motion, for example), and that all three types of locomotion (including idling) could be reached from each other without passing through an action clip (i.e., the motion graph consisting of just the three locomotions was also a strongly connected component).

Three larger motion graphs were created from the primary motion database in the manner detailed above (see Table X). The sneaking and running motions contained in the baseline motion graph created by this process is visualized in Figure 22.

REFERENCES

Abe, Y., Liu, C. K., and Popović, Z. 2004. Momentum-based parameterization of dynamic character motion. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 173–182.

Arikan, O. and Forsyth, D. A. 2002. Interactive motion generation from examples. *ACM Transactions on Graphics 21,* 3 (July), 483–490.

Arikan, O., Forsyth, D. A., and O'Brien, J. F. 2003. Motion synthesis from annotations. *ACM Transactions on Graphics 22,* 3 (July), 402–408.

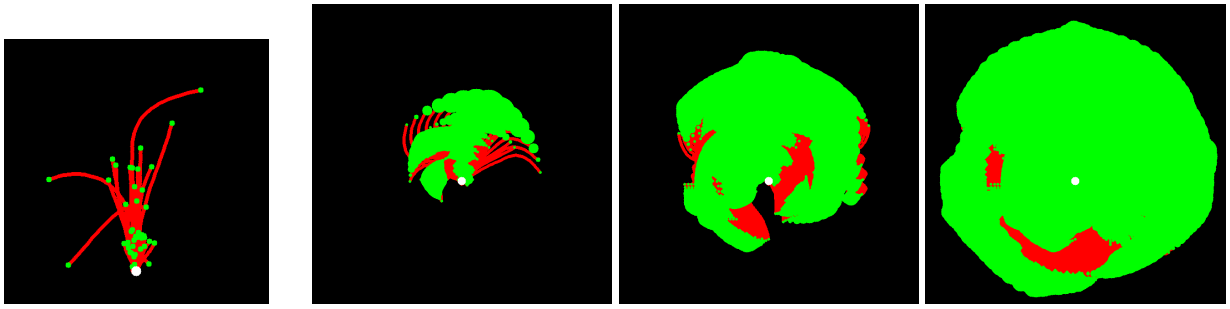
Aubin, J.-P. 1990. A survey of viability theory. *Society for Industrial and Applied Mathematics Journal of Control and Optimization, 28(4), p.749-788*.

Choi, M. G., Lee, J., and Shin, S. Y. 2003. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Transactions on Graphics 22,* 2 (Apr.), 182–203.

Donald, B., Xavier, P., Canny, J., and Reif, J. 1993. Kinodynamic motion planning. *Journal of the ACM 40,* 5, 1048–1066.

Gleicher, M., Shin, H. J., Kovar, L., and Jepsen, A. 2003. Snap-together motion: Assembling run-time animation. *ACM Transactions on Graphics 22,* 3 (July), 702–702.

Gottschalk, S., Lin, M. C., and Manocha, D. 1996. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics 30,* Annual Conference Series, 171–180.



(a) Paths (4x4m) (b) Depth 1 (11x11m) (c) Depth 2 (11x11m) (d) Depth 3 (11x11m)

Fig. 22. Figure 22(a) shows the sneaking and running motions available in the Baseline motion graph, displayed in a 4m by 4m area. The character starts at the white dot, facing upwards. The other figures show in an 11m by 11m area all available sneaking and running motions unrolled to different depths of graph expansion, along with additional clips representing discretized sample of the result of applying our editing model to those source motions. Endpoint markers (green) mark the extent of each clip's editing footprint.

Green, M. 2000a. Driver reaction time. *http://www.visualexpert.com/Resources/reactiontime.html*.

Green, M. 2000b. How long does it take to stop? methodological analysis of driver perception-brake times. *Transportation Human Factors, 2(3), 195-216*.

Harrison, J., Rensink, R. A., and van de Panne, M. 2004. Obscuring length changes during animated motion. *ACM Transactions on Graphics 23,* 3 (Aug.), 569–573.

Heck, R. and Gleicher, M. 2007. Parametric motion graphs. In *2007 ACM Symposium on Interactive 3D Graphics*.

Heck, R., Kovar, L., and Gleicher, M. 2006. Splicing upper-body actions with locomotion. *Computer Graphics Forum 25,* 3 (Sept.), 459–466.

Hodgins, J. K., O'Brien, J. F., and Tumblin, J. 1998. Perception of human motion with different geometric models. *IEEE Transactions on Visualization and Computer Graphics 4,* 4 (October), 307–316.

hoon Kim, T., Park, S. I., and Shin, S. Y. 2003. Rhythmic-motion synthesis based on motion-beat analysis. *ACM Transactions on Graphics 22,* 3 (July), 392–401.

Ikemoto, L. K. M., Arikan, O., and Forsyth, D. 2006. Quick motion transitions with cached multi-way blends. Tech. Rep. UCB/EECS-2006-14, EECS Department, University of California, Berkeley. February 13.

Kavraki, L. E. and Latombe, J.-C. 1998. Probabilistic roadmaps for robot path planning. In *Practical Motion Planning In Robotics: Current Approaches and Future Directions*, K. Gupta and A. del Pobil, Eds. John Wiley, 33–53.

Kosinski, R. J. 2005. A literature review on reaction time. *http://biae.clemson.edu/bpc/bp/Lab/110/reaction.htm*.

Kovar, L., Gleicher, M., and Pighin, F. 2002. Motion graphs. *ACM Transactions on Graphics 21,* 3 (July), 473–482.

Kwon, T. and Shin, S. Y. 2005. Motion modeling for on-line locomotion synthesis. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 29–38.

Latombe, J. C. 1991. *Robot Motion Planning*. Kluwer Academic Publishers, Boston.

Lau, M. and Kuffner, J. J. 2005. Behavior planning for character animation. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 271–280.

Lee, J., Chai, J., Reitsma, P. S. A., Hodgins, J. K., and Pollard, N. S. 2002. Interactive control of avatars animated with human motion data. *ACM Transactions on Graphics 21,* 3 (July), 491–500.

Lee, K. H., Choi, M. G., and Lee, J. 2006. Motion patches: building blocks for virtual environments annotated with motion data. *ACM Transactions on Graphics 25,* 3 (July), 898–906.

Li, Y., Wang, T., and Shum, H.-Y. 2002. Motion texture: A two-level statistical model for character motion synthesis. *ACM Transactions on Graphics 21,* 3 (July), 465–472.

Lozano-Pérez, T. and O'Donnell, P. A. 1991. Parallel robot motion planning. In *Proc. IEEE Intl. Conference on Robotics and Automation*.

Luenberger, D. G. 1979. *Introduction to Dynamic Systems: Theory, Models, and Applications*. John Wiley & Sons.

Mizuguchi, M., Buchanan, J., and Calvert, T. 2001. Data driven motion transitions for interactive games. In *Short Presentation, Eurographics 2001*.

Molina-Tanco, L. and Hilton, A. 2000. Realistic synthesis of novel human movements from a database of motion capture examples. In *In proceedings of IEEE Workshop on Human Motion 2000*.

Muthukrishnan, S. and Pandurangan, G. 2005. The bin-covering technique for thresholding random geometric graph properties. In *Proceedings of SODA 2005*.

Oesker, M., Hecht, H., and Jung, B. 2000. Psychological evidence for unconscious processing of detail in real-time animation of multiple characters. *Journal of Visualization and Computer Animation 11*, 105–112.

Reitsma, P. S. A. and Pollard, N. S. 2003. Perceptual metrics for character animation: Sensitivity to errors in ballistic motion. *ACM Transactions on Graphics 22,* 3 (July), 537–542.

Reitsma, P. S. A. and Pollard, N. S. 2004. Evaluating motion graphs for character navigation. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 89–98.

Ren, L., Patrick, A., Efros, A. A., Hodgins, J. K., and Rehg, J. M. 2005. A data-driven approach to quantifying natural human motion. *ACM Transactions on Graphics 24,* 3 (Aug.), 1090–1097.

Safonova, A. and Hodgins, J. K. 2005. Analyzing the physical correctness of interpolated human motion. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 171–180.

Shin, H. J. and Oh, H. S. 2006. Fat graphs: Constructing an interactive character with continuous controls. In *2006 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 291–298.

Stone, M., DeCarlo, D., Oh, I., Rodriguez, C., Stere, A., Lees, A., and Bregler, C. 2004. Speaking with hands: creating animated conversational characters from recordings of human performance. *ACM Transactions on Graphics 23,* 3 (Aug.), 506–513.

Sung, M., Kovar, L., and Gleicher, M. 2005. Fast and accurate goal-directed motion synthesis for crowds. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 291–300.

Suthankar, G., Mandel, M., Sycara, K., and Hodgins, J. K. 2004. Modeling physical capabilities of humanoid agents using motion capture. In *AAMAS 2004 Proceedings*.

Wang, J. and Bodenheimer, B. 2003. An evaluation of a cost metric for selecting transitions between motion segments. In *2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 232–238.

Wang, J. and Bodenheimer, B. 2004. Computing the duration of motion transitions: an empirical approach. In *2004 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*. 335–344.

Zordan, V. B., Majkowska, A., Chiu, B., and Fast, M. 2005. Dynamic response for motion capture animation. *ACM Transactions on Graphics 24,* 3 (Aug.), 697–701.

Received Month Year Revised Month Year Accepted Month Year