Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2006)
M.-P. Cani, J. O'Brien (Editors)

Precomputed Search Trees: Planning for Interactive Goal-Driven Animation

Manfred Lau and James J. Kuffner

{mlau,kuffner}@cs.cmu.edu
Carnegie Mellon University

Abstract

We present a novel approach for interactively synthesizing motions for characters navigating in complex environments. We focus on the runtime efficiency for motion generation, thereby enabling the interactive animation of a large number of characters simultaneously. The key idea is to precompute search trees of motion clips that can be applied to arbitrary environments. Given a navigation goal relative to a current body position, the best available solution paths and motion sequences can be efficiently extracted during runtime through a series of table lookups. For distant start and goal positions, we first use a fast coarse-level planner to generate a rough path of intermediate sub-goals to guide each iteration of the runtime lookup phase.

We demonstrate the efficiency of our technique across a range of examples in an interactive application with multiple autonomous characters navigating in dynamic environments. Each character responds in real-time to arbitrary user changes to the environment obstacles or navigation goals. The runtime phase is more than two orders of magnitude faster than existing planning methods or traditional motion synthesis techniques. Our technique is not only useful for autonomous motion generation in games, virtual reality, and interactive simulations, but also for animating massive crowds of characters offline for special effects in movies.

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

1. Introduction

In recent years, a number of techniques for organizing and playing back clips of motion data have been developed to generate animations that are natural and lifelike. In this paper, we explore the use of precomputation to increase the runtime speed for synthesizing animation. Runtime efficiency is particularly important for games, virtual reality applications, and interactive simulations. The AI that controls the autonomous characters in these applications is often limited by the computation time that is available. This leads to very simple scripted behaviors; more complex behaviors are not possible due to the time constraints. Moreover, the time constraints are worse if there are a large number of characters. This motivates our careful investigation of how much of the computational cost of planning and complex motion synthesis can be effectively precomputed. The techniques presented in this paper can be used to interactively plan long sequences of motion for multiple characters navigating in dynamic environments with local minima.

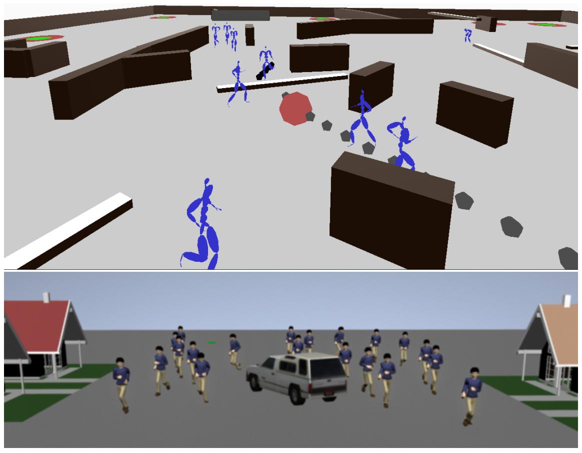


Figure 1: *Screenshot of the interactive viewer (top). The characters respond to user changes in real-time while navigating in large and dynamic environments.*

The main contributions of our method include: 1) The concept of a **Precomputed Search Tree** to increase the speed of motion generation during runtime. We build only one tree, and each character can re-use the same tree. Because we map environment obstacles to the region covered by the tree, we can also reuse the same tree for arbitrary environments; 2) We can **globally plan** for long sequences of motion. This is in contrast to local policy methods, typically used in games, that can fail in environments with local minima; 3) The precomputed search tree can incorporate **complex behaviors** such as jumping and ducking; hence the characters are not limited to navigating on a flat terrain; 4) We can plan motions for a **large number of characters interactively**. We present real-time demos for up to 150 characters moving simultaneosly in dynamic environments. Aside from real-time applications, our method can also be used for synthesizing offline animations of massive crowds of characters for cinematic special effects.

2. Related Work

There has been much work on synthesizing realistic human motions for animated characters. Motion editing techniques [WP95, Gle97] allow clips of motions to be modified for use in different environments or situations. Motion interpolation methods [WH97, RCB98] can generate new motions from a set of examples. Motion graphs [AF02, KGP02, LCR*02, PB02] allow better re-use of motion capture data by re-playing through a connected graph of clips. Physics-based approaches allow us to generate a variety of natural human movements [FvdPT01], and learn motion styles from examples [LHP05].

Interactive applications such as games often use *local policy* methods to generate motions for its characters. These methods use a set of simple rules [Rey87]. Local policies are typically very fast to compute, but often fail in complicated maze-like environments with local minima. It is also difficult to adapt them appropriately for more complex character skeletons such as human figures. For example, the policy may cause a character to turn suddenly to avoid something, resulting in footskate or other undesirable artifacts. In addition, local policies often cannot guarantee non-collision or deadlock between multiple characters. Because our method is built on top of a *global planning* technique, it outputs motions that avoid local minima. Our representation of motion data as high-level behaviors is reminiscent of *move trees* [Men99, MBC01], which are often used in games to represent the motions available for a character.

Many planning approaches have been used for creating animations. Planners based on grasp primitives [KvdP01], probabilistic roadmaps [CLS03], and a two-stage probabilistic planning and motion blending approach [PLS03] have been used. Sung et al [SKG05] also use a two-level planning technique to animate their characters: a PRM approach first generates approximate motions; these motions are then refined to precisely satisfy certain constraints. Planning approaches have also been used to generate human arm motions [KKKL94] and characters performing manipulation tasks [YKH04]. These previous approaches have not shown motions for a large number of characters synthesized in real-time.

Reitsma and Pollard [RP04] introduced the idea of embedding a motion graph into a 4-D grid. The 4-D grid then

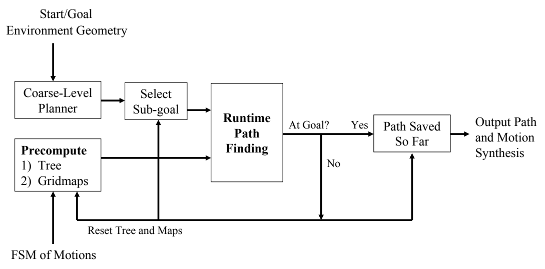


Figure 2: *Overview of the motion generation system.*

represents the possible ways the character can move in the environment. The embedding only works for a specific static environment, and takes on the order of minutes to compute. This is not a problem for their work because their focus was on the evaluation of motion graphs. In contrast, we are concerned with improving the runtime speed of the search. Lee and Lee [LL04] have also explored the idea of precomputation. They preprocess sets of motion data to compute a *control policy* for a character's actions. This policy was then used at runtime to efficiently animate boxing characters. Our approach is different in that we precompute a search tree of possible future actions. We then build gridmaps over this tree so that we can efficiently index to the relevant portions of the tree during runtime.

3. Precomputed Search Trees

Figure 2 shows an overview of our system. The inputs to the system include: a starting position and orientation, a goal position, a description of the environment geometry, and a Finite-State Machine of motion states. In the precomputation phase, we build a search tree of the states in the FSM. We also compute gridmaps over this tree so that we can access the nodes and paths in the tree more efficiently later. During runtime, we first use a bitmap planner to search for a coarse path. This path is repeatedly used to select intermediate sub-goals for use in the *Runtime Path Finding* module. This module returns a sub-path, part of which might be saved as the overall path. We continue this *Runtime Path Finding* process until we have reached the final goal. The resulting path corresponds to a sequence of states chosen from the FSM. This sequence is then converted into motion that allows the character to move from the start to the goal.

Environment Representation The inputs to the system include an annotated geometric description of the obstacles in the environment, similar to the one in [LK05]. In particular, there can be special obstacles such as archways or overhangs where the character can duck underneath, or low obstacles on the floor which the character can jump over. We must have a corresponding *valid region* for each of the special obstacles when we perform the *Runtime Path Finding*.

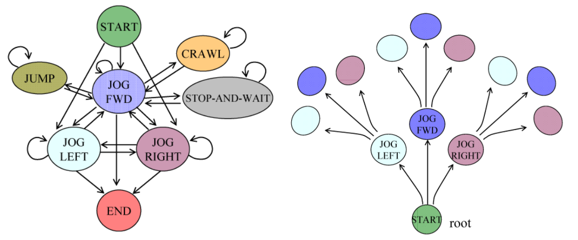


Figure 3: ***Left:*** *A small FSM of behaviors.* ***Right:*** *The first two levels of a small search tree example.*

For the purpose of collision detection, we bound the character by a cylinder of radius r. We enlarge the size of the obstacles by r, and shrink the character to a point. The problem then becomes planning a path for this point in the enlarged obstacle space [LaV06].

Finite-State Machine We have an FSM of behavior states which is used to precompute the tree. Figure 3(left) shows a small FSM. Each state or node contains a set of motion clips representing a high-level behavior, and each edge represents transitions between the behaviors. In order for solution paths to be found for a variety of environments and goal positions, it is important that we design this FSM carefully. There is one cost for each behavior state, equal to the approximate distance the character travels multiplied by a user weight.

Precomputing the Search Tree We build a tree of the states in the FSM up to a certain depth level. Figure 3(right) shows the beginning of such a tree for a small set of behaviors. This search tree represents all the possible paths that the character can take from an initial start position and orientation. If we build the exhaustive tree (Figure 4 left), its size grows exponentially with respect to depth level. Equation 1 shows the total number of tree nodes for an exhaustive tree with branching factor b and depth level d. The branching factor also depends on the number of behavior states and how they transition to each other. In general, the tree size is $O((average\ b)^d)$. Our precomputed tree is a pruned tree that limits the number of paths that can reach each point (see Figure 4 right). The intuition is that there are parts of the exhaustive tree that have more than 1,000 paths reaching it. A large number of these paths are similar, and hence it is reasonable to prune them.

$$total\ number\ of\ tree\ nodes = \sum_{k=0}^{d} b^k \tag{1}$$

Each node of the tree has the position, orientation, cost, and time of the path up to that node starting from the root node. We initialize the tree with an empty root node. We also initialize an empty grid similar to the one in Figure 5(left). To build the nodes in the $d+1$ level, we consider all the child nodes of the nodes in the d level. We must consider the transitions (as specified in the FSM) when we build this set of child nodes. We randomly go through this set of child nodes, and decide if each one should be added. If the node's corresponding cell (x_i, y_i) (see Figure 5 left) has less than a prespecified number of k nodes already in it, we will add the node and increment the number of nodes in that cell. We thereby limit the number of nodes in each cell to k. In Figure 4(right), k is set to 100. We also limit the total number of nodes we built.

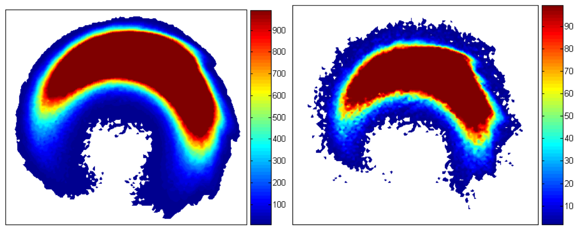


Figure 4: *Frequency plot of the precomputed search tree. Each point represents the number of paths that can reach that point from the root of the tree. The root is near the middle of each figure and the tree progresses in a forward direction (or up in the figure). The tree covers an area that is approximately a half circle of radius 16 meters, with the character starting at the center of the half circle. The majority of paths end up in an area between 8 and 14 meters away from the start. We used about 1,500 frames of motion at 30 Hz.* ***Left:*** *Exhaustive tree of 6 depth levels built from FSM with 21 behavior states. This tree has over 6 million nodes (over 300 MB).* ***Right:*** *The pruned tree has 220,000 nodes (about 10 MB).*

Since each node has only one parent, we can trace back the path to the root to find the sequence of behavior states that can reach that point. Hence each node also represents the path up to and including that point. Each node also has a *blocked* variable, initialized here with *UNBLOCKED*. If this variable is set to *BLOCKED*, this means we know for sure that we can neither reach that point nor any of the corresponding descendant nodes. However, the path from the root to the parent node of that point may still be reachable.

Environment Gridmap We build an *environment gridmap* over the tree as shown in Figure 5(left). The gridcells are all initially marked as *UNOCCUPIED*. Each node of the tree can then be associated with a gridcell. For example, node i corresponds to cell (x_i, y_i) in Figure 5(right). We precompute and explicitly store the corresponding (x_i, y_i) value in each node so that we can quickly access the cell that a node is in during runtime.

The size of the gridcells is a parameter of the system, and we used a range of sizes from 14 cm to 28 cm. This parameter, however, affects the runtime phase significantly. It may increase the time for mapping the environment to the tree (Section 4.2) by approximately nine if the size of each cell is cut to a third of the original. We want to balance between a large cell size which decreases the runtime, and a small cell size which represents the environment more accurately.

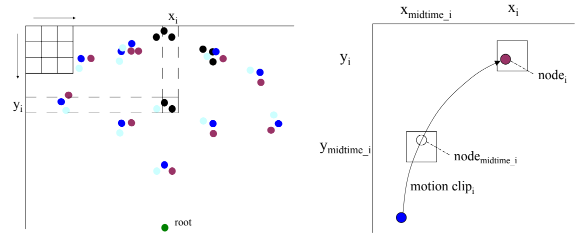


Figure 5: *Left: An **environment gridmap** initialized with UNOCCUPIED cells. The intuition for this gridmap is that if cell (x_i, y_i) is occupied by an obstacle, the tree nodes corresponding to this cell and their descendant nodes (the black ones) are BLOCKED. **Right:** For each node i, we precompute and store the corresponding values x_i, y_i, $x_{midtime\_i}$, and $y_{midtime\_i}$.*

For each tree node *i*, we also precompute and store the values $x_{midtime\_i}$ and $y_{midtime\_i}$ (Figure 5 right). We first take half (with respect to time duration) of motion clip *i* to reach node *midtime_i*. The values $x_{midtime\_i}$ and $y_{midtime\_i}$ are then stored in node *i*. Node *midtime_i* is used temporarily in this calculation and does not exist in the tree. If the motion clip is one of the special motions such as jumping, we do not take half of the clip to get node *midtime_i*. Instead we use the point where the special motion is actually executed. For the example of jumping, this is where the character is at or near the highest point of its jump. This information should already be pre-annotated in the motion data. These "midtime" positions are used for collision checking in the runtime phase. For the special motions, they are used to see if the character successfully passes through the corresponding special obstacle. The choice of taking half of the clip is only a discretization parameter. For more accurate collision checking, we can continue to split the clip and compute similar information. We choose only the "midtime" discretization because our motion clips are short in length (hence midtime in enough), and a smaller discretization gives a faster runtime.

Goal Gridmap We precompute a *goal gridmap* (Figure 6) used in the runtime path finding phase (Section 4.3). For every node in the tree, we place it in the corresponding cell in the gridmap. Each cell then contains a sorted list of nodes. We used a range of cell sizes from 45 to 90 cm. The *environment gridmap* does not explicitly store this list of nodes, but the *goal gridmap* does.

4. Runtime Path and Motion Generation

4.1. Coarse-Level Planner

Figure 2 illustrates how each module works with the rest of the system. We first use a fast bitmap planner to generate a coarse-level path from the start to the goal. This path is then used as a guideline for picking sub-goals to run each subcase of the runtime path finding phase. In our implementation, we used a bitmap planning algorithm [Kuf04] optimized for 2D grids. Note that the coarse-level planner is not necessary if the goal position lies within the space covered by the precomputed tree. In this case, we can apply the runtime path finding module once and immediately find a solution path.

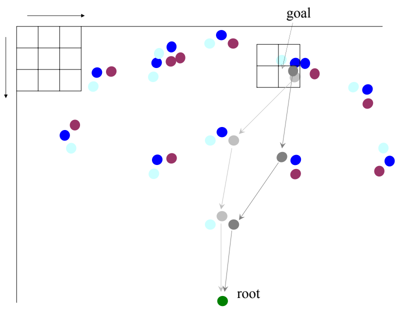


Figure 6: *In the **goal gridmap**, each cell contains a sorted list of nodes. The list of nodes in each cell can also be thought of as a list of paths derived by tracing the node back towards the root of the tree. They are sorted by the sum of the cost of the motion states in the path.*

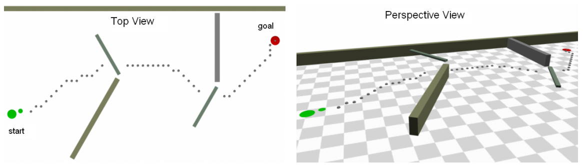


Figure 7: *The points of the path that are eventually chosen by the coarse-level planner. The smaller green circle represents the starting orientation.*

A coarse-level map of the environment is used as input to the bitmap planner. The size of the gridcells was about 70 cm. We map the obstacles to this coarse gridmap using the technique discussed in Section 4.2. In particular, we only use the regular obstacles in this step, and not the special ones (ie. one that the character has to jump over). A path is returned that may go through these special obstacles. This is fine since the character has the motion capabilities (ie. jump) to go through these obstacles.

The special obstacles are then added to the coarse-level map. We can eliminate parts of the returned path that collide with these obstacles. In addition, we annotate the parts of the path that appear just before the special obstacles, and the parts that appear just before the final goal position (but not the final goal position). Figure 7 shows an example of the points in the path that are eventually chosen. Note that in this example, there is an obstacle that the character must duck under, and one that it must jump over.

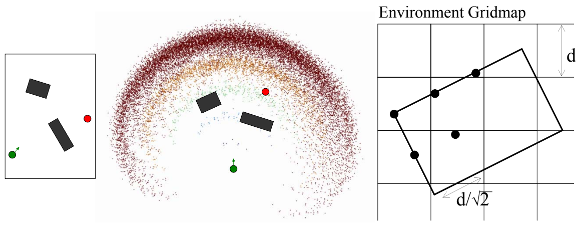


Figure 8: ***Left:*** *Translating and rotating the start configuration and the obstacles to fit the coordinate system of the precomputed tree.* ***Right:*** *If the size of the gridcell is d, we can guarantee that the mapping is correct if the sampling of points for the obstacle is at most $d/\sqrt{2}$ apart.*

4.2. Mapping Obstacles to Environment Gridmap

In this module, we want to mark each cell of the environment gridmap as either *OCCUPIED* or as a *valid region* of a special obstacle. We first translate the starting position to match the root node of the precomputed tree, and rotate the starting orientation to face the forward direction (Figure 8 left). All the obstacles in the environment are translated and rotated similarly. Each obstacle is then mapped to the environment gridmap.

If an obstacle is outside the region covered by the tree, we can safely ignore it. Otherwise, we map it to the environment gridmap by iterating through a discretized set of points inside the obstacle (Figure 8 right). If a gridcell is *OCCUPIED*, we know that the tree nodes in that cell are *BLOCKED*. But in order to save time, we will not mark them as such until it is necessary to do so in the runtime path finding step. The indices of each gridcell that gets marked as being occupied are recorded as the mapping proceeds. We use this information to quickly reset the environment gridmap every time we re-execute this mapping module.

4.3. Runtime Path Finding

We repeatedly use the path from the coarse bitmap planner to select sub-goals, which are then used in repeated iterations of the path finding technique in Algorithm 1.

Sub-goal Selection The coarse level path has many points that we can use as sub-goals. Intuitively, we would like to find ones that will be within the dark red regions (Figure 4) of the precomputed tree. We choose the sub-goal to be the point in the coarse path that is closest to a fixed distance away from the start (Figure 9). A distance between 10 and 12 meters worked well in our examples. Note that the start is different for each iteration of the path finding phase.

Runtime Path Finding Algorithm Algorithm 1 takes a goal position as input, and returns the tree node that represents the solution path. If there is no possible path in the precomputed tree that can reach the goal, it will recognize that there is no solution. The inputs also include the precomputed tree, the environment gridmap, and the goal gridmap. The

Algorithm 1: RUNTIME PATH FINDING

```
  $Goal_{GoalGrid} \leftarrow T(Goal_{Global})$
  $P \leftarrow$ GoalGrid[$Goal_{GoalGrid}.x$][$Goal_{GoalGrid}.y$].Nodes()
  foreach $p \in P$ do
      while ( ($p.BLOCKED == false$) and
              ($EnvGrid[p.x_i][p.y_i] == UNOCCUPIED$) and
              ($p$ != $rootNode$) ) do
1         $p.BLOCKED \leftarrow$ true
          // midtime collision check
          if isSpecialMotion(p.motionState) then
2             if $EnvGridmap[p.x_{midtime\_i}][p.y_{midtime\_i}]$ !=
                specialObstacle(p.motionState) then
                  continue to next $p$
              end
          else
3             if $EnvGridmap[p.x_{midtime\_i}][p.y_{midtime\_i}]$ ==
                OCCUPIED then
                  continue to next $p$
              end
          end
          $p \leftarrow p.parent$
      end
      // this path traced through before
4     if $p.BLOCKED == true$ then continue to next $p$
      // this path is blocked by an obstacle
5     if $EnvGridmap[p.x_i][p.y_i] == OCCUPIED$ then
6         $p.BLOCKED \leftarrow$ true
          continue to next $p$
      end
      // reached rootNode and path found
7     return node representing current path
  end
  return no path found
```

goal position is first translated and rotated from its global coordinates to the coordinate system of the goal gridmap (function T). The transformed indices are used to find P, the list of nodes sorted in increasing cost. We then go through each node p in P, and try to trace it back towards the root node (which is where the start is in the current iteration). As we trace back towards the root, we mark each node as BLOCKED, if it is not already BLOCKED or not obstructed by an obstacle. The intuition behind this is that we want to find the shortest path that is not obstructed in any way. We also check to see that the "midtime" point of the motion clip reaching that node is not obstructed (line 3) before tracing back to its parent node. Furthermore, if we have arrived at that node through a special motion (line 2), we check to see if the motion successfully goes through the corresponding special obstacle by checking to see if the "midtime" point is a *valid region* of that type of special obstacle. The $specialObstacle()$ function returns the type of this corresponding obstacle. If the "midtime" point is obstructed in any way, the algorithm will continue to the next possible node in P.

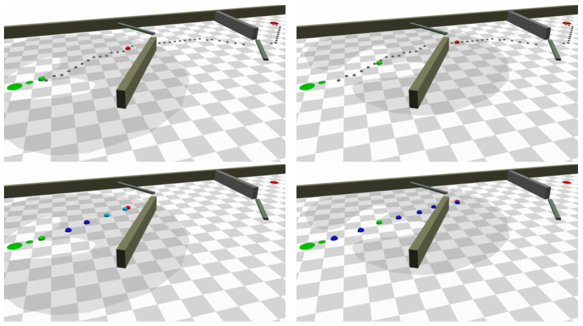
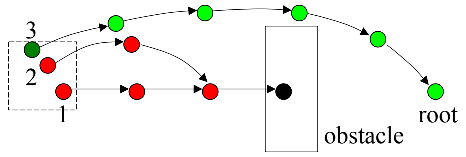


Figure 9: *The 2 columns correspond to the first 2 iterations of the runtime path finding phase for this example. The top row shows the start (green sphere) in each iteration, and the sub-goal (red sphere) selected from the coarse-level path. The bottom row shows the path returned by the path finding algorithm (light and dark blue) and the partial path chosen (dark blue only). An estimate of the outline of the precomputed tree is shown. The tree is transformed to the global space only in the figure to show how it relates to the other parts of the environment. There is only one precomputed tree, and it is never transformed to the global space in the algorithm.*

There are three conditions under which each trace of node p towards the root node stops (Figure 10):

1. Pointer p arrives at a node that is obstructed by an obstacle (case 1 in Figure 10(top) and line 5 in Algorithm 1). When this happens, the path from the root node to p cannot be a solution. We mark that node as BLOCKED (these are the black nodes in Figure 10 and also the ones marked BLOCKED in line 6) and proceed to test the next node in P.
2. Pointer p arrives at a BLOCKED node (case 2 and line 4). When this happens, the path from the root node to p also cannot be a solution. The algorithm then continues to test the next node in P. The red nodes in Figure 10 are the nodes that were traced back. These are the ones that are marked as BLOCKED in line 1 of Algorithm 1.
3. Pointer p arrives at the root node (case 3 and line 7). The green nodes correspond to the nodes traced back for this p. Note that these are also marked as BLOCKED, but it does not matter since we already have the solution. We stop testing the list of nodes P, and return the original node that p points to in this case (the darker green node in Figure 10 top) to represent the solution path. If we have gone through the whole list P without having reached the root node, there is no possible solution.

The solution path from the algorithm is in the coordinate system of the precomputed tree. We must therefore transform (T^{-1}) each node in the path back to the global coordinate system. The bottom row of Figure 9 shows examples of the algorithm's output. Before running a new iteration of the path finding phase, we reset the tree by UNBLOCK-ing the nodes that were previously BLOCKED.



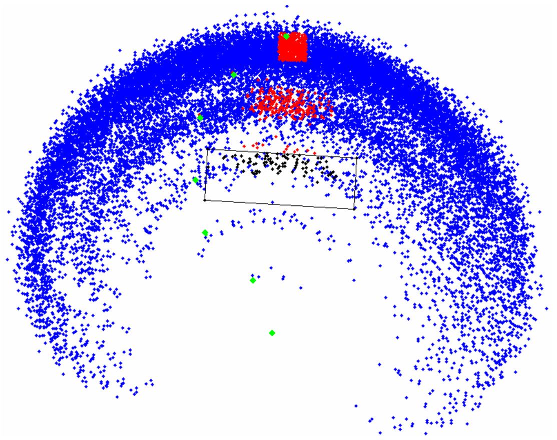


Figure 10: *The process of tracing back the list of sorted nodes P towards the root node in Algorithm 1.* ***Top:*** *The 3 cases under which each trace of node p stops. The sub-goal is inside the dashed square (a cell of the goal gridmap).* ***Bottom:*** *Simple example. The blue nodes are the nodes of the precomputed tree. The sub-goal is somewhere in the square-shaped box of red nodes. The other colored nodes correspond to the 3 cases.*

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

More specifically, if the current sub-goal is not annotated as being near a special obstacle or the final goal, we keep the whole solution path. Otherwise, we only keep the first 2 motion states of the solution path (Figure 9 bottom). In addition, if the solution path includes a special motion state, we take all the states up to and including it. This is an optional adjustment that again helps the character in adjusting itself before executing a special motion. Our experiments show that without this adjustment, the algorithm may sometimes inaccurately report that there is no solution.

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

4.4. Motion Synthesis

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

5. Experimental Setup

To demonstrate the efficiency of our technique, we have built interactive applications in which the user can modify the environment, and the characters will react to the changes autonomously. The "runtime path finding" phase is executed repeatedly to adapt to changes in the environment.

Single Character Mode The complete path for one character is continuously re-generated as the user changes the environment. We draw some of the character's poses to represent the animation. The precomputation phase (Section 3) is first executed once before the draw loop begins. If the user changes the position and orientation of an obstacle or the goal position, we update this information and apply the runtime path finding phase as described in Section 4. If no changes are detected, we can just draw the previously found solution.

Multiple Character Mode The motions for multiple characters are generated in real-time. We do not generate the full path at the beginning. We execute a "runtime path finding" phase to synthesize the next partial path only after we start rendering the first frame from the previous partial path. Hence the characters can respond to environment changes continuously.

To handle multiple characters (Algorithm 2), we specify a maximum planning time ($T_{planning}$), and plan as many characters as possible within each iteration of the draw loop. The exception is that we plan for every character once when we execute the planning loop for the first time. We are *ready to plan the current character* (line 1 of Algorithm 2) after we start rendering the first frame (of the current character) from the previously planned partial path.

To *generate the next partial path/motion* (line 2), we execute the runtime path finding phase (Section 4). We run the coarse-level planner with the updated starting location as the

Algorithm 2: MULTIPLE CHARACTER PLANNER

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

end of the last partial path. The sub-goal selection and runtime iteration is done just once, since we only need the first partial path here. We use the same precomputed tree for all the characters, so we reset the gridmaps and precomputed tree after each character's runtime iteration.

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

In addition, we need to deal with collision avoidance between characters. We apply these 3 steps. First, we use a *global characters gridmap* to store the time-indexed global positions of the characters after their poses are stored to the data buffer. These positions are placed into the correct bucket in the gridmap for efficient access later. Second, in addition to mapping the obstacles to the environment gridmap (Section 4.2), we map the characters' positions to a *local characters gridmap* using a similar procedure. During the runtime iteration, we select the cells in the global characters gridmap that are relevant in each local sub-case. This assures that the collision check between characters is linear in the number of characters, instead of being quadratic in the naive case. The positions are also placed into the appropriate bucket in the local gridmap. Third, as we trace through the tree nodes in Algorithm 1, we check to see if each node can collide with the locally relevant characters. An additional test is needed in the *while* loop that performs a Euclidean distance check between the node's position and each of the relevant charac-

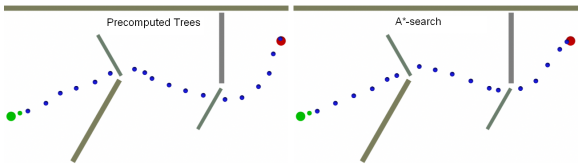


Figure 11: *The solution paths of motion states for the environment in Figure 7.*

ters' position. With the use of the local characters gridmap, this step is fast because we rarely have to perform a distance check.

6. Results

We measured the execution time of the path finding phase across a variety of example scenarios. In tables 1 and 2, the "average runtime" for Precomputed Trees gives the execution time of the techniques in Section 4. It does not include the time for motion synthesis (Section 4.4), and for rendering the character's motion. The "average output time" is the length of the resulting animation (at 30 Hz) returned by the system. The A*-Search columns are the results for the same input environments and using a *Behavior Planning* technique [LK05] that builds the search tree during runtime.

	Precomputed Tree	A*-Search
avg runtime (μs)	3,131	550,591
	176 times faster	
avg output time (μs)	13,123,333	12,700,000
real-time speedup	4,191	23
average μs/frame	7.95	1,445
average pathcost	361	357

Table 1: *Runtime for example with special obstacles (Figure 7). The FSM used to build the tree has 21 motion states. The average precomputation time was 49.3 seconds. The results are averaged over 10 executions of randomly selected start and goal locations.*

Table 1 shows the average runtime for the environment in Figure 7. For a *trivial case* where there is a direct path between the start and goal, the Precomputed Trees method was 143 times faster than A*-Search (571 vs. 81,535 μs). And it took 2.62 μs for Precomputed Trees and 372 μs for A*-search to compute each frame. Figure 11 shows the solution paths for one test case. The A*-Search solution is globally optimal with respect to the cost of the motion states. Our experiments have shown that the Precomputed Trees technique produces solution paths that are near optimal up to the resolution of the grid maps.

Table 2 is for the maze example in Figure 12. In the *most time-consuming case* for both methods, Precomputed Trees was 4,163 times faster than A*-Search (6.23 vs. 25,940 ms). And it took 5.89 μs for Precomputed Trees and 26,687 μs for A*-Search to compute each frame. These results demonstrate that our method scales well to both trivial and complex cases.

	Precomputed Tree	A*-Search
avg runtime (μs)	4,865	17,598,088
	3,617 times faster	
avg output time (μs)	29,072,222	27,600,000
real-time speedup	5,976	1.57
average μs/frame	5.58	21,253
average pathcost	1,113	1,064

Table 2: *Runtime for maze example (Figure 12). The main differences from Table 1 are: the FSM used to build this tree has 13 motion states (no special motions), and the environment here is larger, more complex and has local minima. The average precomputation time was 3.1 seconds. The results are averaged over 6 executions.*

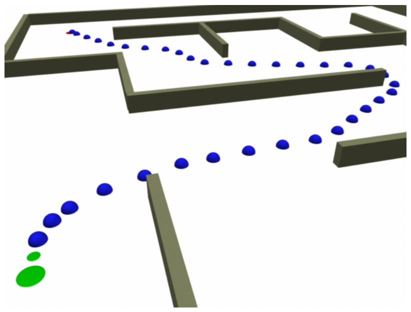


Figure 12: *A sample solution path (using Precomputed Trees) for the maze environment.*

The runtime in Tables 1 and 2 is the average *search time* for each frame in the solution. The output of the search is a sequence of states. If we convert the states into motion, we need to translate and rotate each pose, which takes an average of 10 μs. So the total time to generate each pose is about 18 μs (adding the 7.95 from Table 1).

Figure 13 shows the effect of pruning the precomputed tree (Section 3). We can see that there is only a small difference in path cost even if the size of the tree decreases significantly. If we use a tree of less than 10 MB, the tree is so small that the algorithm may sometimes report incorrectly that there is no solution. Furthermore, the graph shows that as the memory decreases, the path tends to get worse (higher cost). Thus, there is a clear tradeoff between the memory used and the quality and variety of motions that can be generated.

Single Character Mode The full solution path for each character in Figure 14 (both cases) can be re-generated at approximately 25 times per second. This means that about 1,200 frames of motion (for the larger case) can be continuously re-synthesized with respect to user changes 25 times each second.

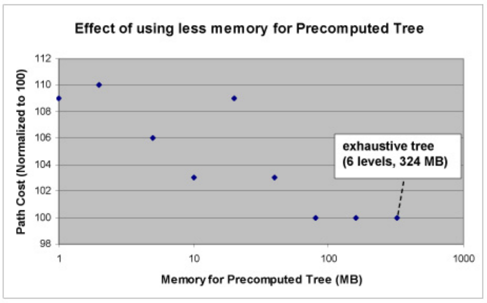


Figure 13: *Using a significantly smaller precomputed tree has only a minimal effect on the overall quality of the paths that can be synthesized. We start from an exhaustive tree with 6 levels, and keep building smaller trees with half the memory each time. We used these trees to synthesize motion and measured the average path cost. The 324 MB tree is shown in Figure 4(left). The 10 MB tree is in Figure 4(right). The larger trees do not necessarily include the smaller ones as subtrees because of the way the pruning is done.*

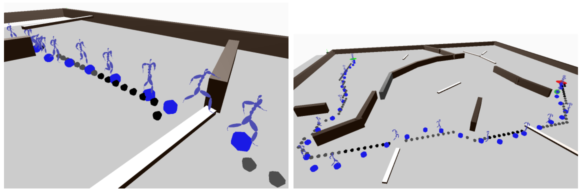


Figure 14: *Screenshots of the one character viewer. The gray spheres show the coarse-level path. The darker gray ones are the ones annotated as appearing just before a special obstacle or the goal. The blue spheres show the solution path of motion states.* ***Left:*** *Simple environment. Each complete solution path has about 20 seconds or 600 frames of motion. The poses are drawn 1 second apart.* ***Right:*** *Larger (70 by 70 meters) and more complex environment. Each solution has about 1,200 frames of motion. Poses are drawn 2 seconds apart.*

The screen's frame rate for the viewer is 28 Hz at a resolution of 1280 x 960. For the simple environment, the frame rate changes to 26 Hz if we continuously move an obstacle to allow the "runtime path finding" to execute as much as possible. It takes about 3 ms to generate the full solution path, which is approximately what we expect from the runtime tables above. For the larger environment, the frame rate changes to 24 Hz if we allow the runtime phase to execute as much as possible. In this case, it takes about 6 ms to generate the full path. The one character viewer is included mainly as a demonstration of speed, since other techniques can be used to generate the motion for one character in real-time.

Multiple Character Mode We demonstrate experimental results for (1) multiple characters following navigation goals that the user can interactively modify, (2) many characters in a large complex environment (Figure 1), (3) the user "shooting" an obstacle at many characters (Figure 15), and (4) an

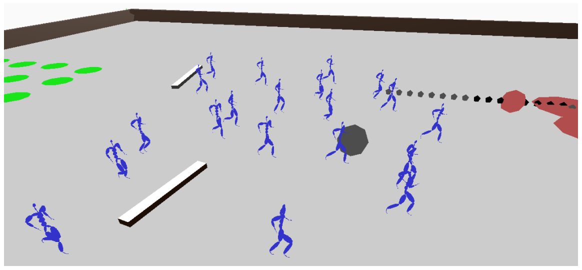


Figure 15: *Screenshot of the multiple character viewer. The small gray spheres show the most recently planned coarse path for the current character being planned for. The red sphere shows the sub-goal chosen in the corresponding subcase. The user can "shoot" an obstacle (the larger gray sphere) and the characters will update their motion plan to avoid it. The positions of the obstacle are computed and stored in the* global characters gridmap.

example containing 150 characters. Each iteration of the runtime phase takes about 8.5 *ms*. The screen rate is about 23 Hz at a resolution of 1280 by 960.

7. Discussion

We have presented a "Precomputed Search Tree" technique for interactively generating realistic animations for virtual characters navigating in a complex environment. The main contribution is the idea of computing the search tree and related gridmaps in advance so that we can efficiently synthesize solution paths during runtime. Our technique can *globally plan* in large environments with local minima, generate *complex behaviors* such as jumping and ducking, and synthesize motions for multiple characters in real-time.

The average search time per frame of 8 μs is two orders of magnitude faster than previous approaches, and 4,000 times faster than real-time. The speedup of our technique comes primarily from not having to build the search tree during runtime, and from the coarse-level planner generating subgoals to guide the runtime phase. We also avoid having to access large numbers of tree nodes during runtime by building gridmaps in advance. In addition, the environment is mapped to the tree during the path finding phase, rather than mapping the tree to the environment which would be much more expensive. Another reason for the speedup is that the search is performed on the motion states rather than individual poses. The search time is therefore amortized over the frames of motion in each state.

We have successfully used a precomputed tree of 10 MB to synthesize motions. We have shown that scalability is not an issue because we can significantly reduce the size of the tree without sacrificing the variety of paths and motions that we can generate. Our animations demonstrate that our dataset is complex enough that we can produce a large variety of motions.

One drawback of our system is that we sacrifice global optimality in order to achieve interactive runtime speed. However, our results have demonstrated that our solution paths are very close to the optimal paths given by a traditional A* planner. Another limitation of our method is the handling of goal positions that are beyond the search horizon of the precomputed tree. There may exist a round-about path to reach such a goal position, but our algorithm will not find it because it is not reachable at the given tree depth. Having more varied motions (ie. turning on-the-spot, stepping sideways) can help to expand the overall reachability of the space covered by the tree. Finally, our system has no notion or rules that specifically lead to crowd animation control, even though it can be successfully used to interactively animate a large number of characters simultaneously.

One area of future work is to explore the tradeoffs between memory use, motion generation power, and the speed of search. We have only shown that a decrease in memory use does not lead to a significant decrease in motion generation power nor motion quality. Moreover, it would be useful to explore how we can build an optimized tree. For example, one may want to produce a better tree node distribution by using the same amount of memory to cover a larger area while decreasing the average number of paths reaching each point.

Acknowledgements: We thank Moshe Mahler for helping with video production, and the CMU Graphics Lab for providing many resources used for this project. We are grateful to Alias/Wavefront for the donation of their Maya software. This research was partially supported by NSF grants ECS-0325383, ECS-0326095, and ANI-0224419.

References

[AF02] ARIKAN O., FORSYTH D. A.: Interactive motion generation from examples. *ACM Transactions on Graphics 21*, 3 (July 2002), 483–490.

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

[FvdPT01] FALOUTSOS P., VAN DE PANNE M., TERZOPOULOS D.: The virtual stuntman: dynamic characters with a repertoire of autonomous motor skills. *Computers and Graphics 25*, 6 (2001), 933–953.

[Gle97] GLEICHER M.: Motion editing with spacetime constraints. In *1997 Symposium on Interactive 3D Graphics* (Apr. 1997), pp. 139–148.

[KGP02] KOVAR L., GLEICHER M., PIGHIN F.: Motion graphs. *ACM Transactions on Graphics 21*, 3 (July 2002), 473–482.

[KKKL94] KOGA Y., KONDO K., KUFFNER J. J., LATOMBE J.-C.: Planning motions with intentions. In *Proceedings of SIGGRAPH 94* (July 1994), pp. 395–408.

[Kuf04] KUFFNER J. J.: Efficient optimal search of euclidean-cost grids and lattices. *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)* (July 2004).

[KvdP01] KALISIAK M., VAN DE PANNE M.: A grasp-based motion planning algorithm for character animation. *J. Visualization and Computer Animation 12*, 3 (2001), 117–129.

[LaV06] LAVALLE S. M.: *Planning Algorithms*. Cambridge University Press (also available at http://msl.cs.uiuc.edu/planning/), 2006. To be published in 2006.

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

[LHP05] LIU C. K., HERTZMANN A., POPOVIC Z.: Learning physics-based motion style with nonlinear inverse optimization. *ACM Trans. Graph 24*, 3 (2005).

[LK05] LAU M., KUFFNER J. J.: Behavior planning for character animation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aug. 2005), pp. 271–280.

[LL04] LEE J., LEE K. H.: Precomputing avatar behavior from human motion data. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2004), ACM Press, pp. 79–87.

[MBC01] MIZUGUCHI M., BUCHANAN J., CALVERT T.: Data driven motion transitions for interactive games. *Eurographics 2001 Short Presentations* (September 2001).

[Men99] MENACHE A.: *Understanding Motion Capture for Computer Animation and Video Games*. Morgan Kaufmann Publishers Inc., 1999.

[PB02] PULLEN K., BREGLER C.: Motion capture assisted animation: Texturing and synthesis. *ACM Transactions on Graphics 21*, 3 (July 2002), 501–508.

[PLS03] PETTRE J., LAUMOND J.-P., SIMEON T.: A 2-stages locomotion planner for digital actors. *Symposium on Computer Animation* (Aug. 2003), 258–264.

[RCB98] ROSE C., COHEN M., BODENHEIMER B.: Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Application 18*, 5 (1998), 32–40.

[Rey87] REYNOLDS C. W.: Flocks, herds, and schools: A distributed behavioral model. In *Computer Graphics (SIGGRAPH '87 Proceedings)* (July 1987), vol. 21, pp. 25–34.

[RP04] REITSMA P. S. A., POLLARD N. S.: Evaluating motion graphs for character navigation. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aug. 2004).

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

[WH97] WILEY D., HAHN J.: Interpolation synthesis of articulated figure motion. *IEEE Computer Graphics and Application 17*, 6 (1997), 39–45.

[WP95] WITKIN A. P., POPOVIĆ Z.: Motion warping. In *Proceedings of SIGGRAPH 95* (Aug. 1995), Computer Graphics Proceedings, Annual Conference Series, pp. 105–108.

[YKH04] YAMANE K., KUFFNER J. J., HODGINS J. K.: Synthesizing animations of human manipulation tasks. *ACM Transactions on Graphics (SIGGRAPH 2004) 23*, 3 (Aug. 2004).