Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D

Alex Nash[*] and **Sven Koenig**
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{anash,skoenig}@usc.edu

Craig Tovey
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0205, USA
craig.tovey@isye.gatech.edu

Abstract

Grids with blocked and unblocked cells are often used to represent continuous 2D and 3D environments in robotics and video games. The shortest paths formed by the edges of 8-neighbor 2D grids can be up to $\approx 8\%$ longer than the shortest paths in the continuous environment. Theta* typically finds much shorter paths than that by propagating information along graph edges (to achieve short runtimes) without constraining paths to be formed by graph edges (to find short "any-angle" paths). We show in this paper that the shortest paths formed by the edges of 26-neighbor 3D grids can be $\approx 13\%$ longer than the shortest paths in the continuous environment, which highlights the need for smart path planning algorithms in 3D. Theta* can be applied to 3D grids in a straight-forward manner, but it performs a line-of-sight check for each unexpanded visible neighbor of each expanded vertex and thus it performs many more line-of-sight checks per expanded vertex on a 26-neighbor 3D grid than on an 8-neighbor 2D grid. We therefore introduce Lazy Theta*, a variant of Theta* which uses lazy evaluation to perform only one line-of-sight check per expanded vertex (but with slightly more expanded vertices). We show experimentally that Lazy Theta* finds paths faster than Theta* on 26-neighbor 3D grids, with one order of magnitude fewer line-of-sight checks and without an increase in path length.

Introduction

We are interested in path planning for robotics and video games. Path planning consists of discretizing a continuous environment into a graph (*generate-graph* problem) and propagating information along the edges of this graph in search of a short path from a given start vertex to a given goal vertex (*find-path* problem) (Wooden 2006; Murphy 2000). Roboticists and video game developers solve the *generate-graph* problem by discretizing the continuous environment into regular 2D grids composed of squares

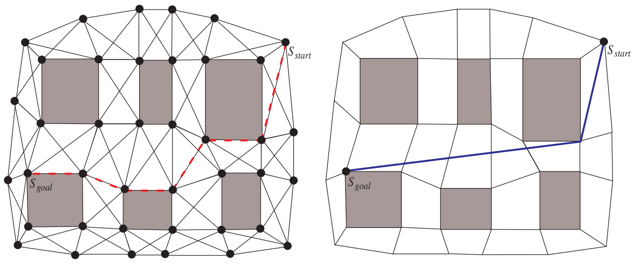

Figure 1: NavMesh: Shortest Path Formed by Graph Edges (left) vs. Truly Shortest Path (right), adapted from (Patel 2000)

(square grids), hexagons or triangles; regular 3D grids composed of cubes (cubic grids); visibility graphs; waypoint graphs; circle based waypoint graphs; space filling volumes; navigation meshes (NavMeshes, tessellations of the continuous environment into n-sided convex polygons); hierarchical data structures such as quad trees or framed quad trees; probabilistic road maps (PRMs) or rapidly exploring random trees (Björnsson et al. 2003; Choset et al. 2005; Tozour 2004). Roboticists and video game developers typically solve the *find-path* problem with A* because A* is simple, efficient and guaranteed to find shortest paths formed by graph edges when used with admissible heuristics. A* and other traditional *find-path* algorithms propagate information along graph edges *and* constrain paths to be formed by graph edges. However, shortest paths formed by graph edges are not necessarily equivalent to the truly shortest paths (in the continuous environment). This can be seen in Figure 1 (right), where a continuous environment has been discretized into a NavMesh. Each polygon is either blocked (grey) or unblocked (white). The path found by A* can be seen in Figure 1 (left) while the truly shortest path can be seen in Figure 1 (right). In this paper, we therefore develop sophisticated any-angle *find-path* algorithms that, like A*, propagate information along graph edges (to achieve short runtimes) but, unlike A*, do not constrain paths to be formed by graph edges (to find short "any-angle" paths). The shortest paths formed by the edges of square grids can be up to $\approx 8\%$ longer than the truly shortest paths. We show in this paper that the shortest paths formed by the edges of cubic grids can be $\approx 13\%$ longer than the truly shortest paths, which highlights the need for any-angle *find-path* algorithms in 3D. We therefore extend Theta* (Nash et al. 2007), an existing any-angle *find-path* algorithm, from an algorithm that only applies to square grids to an algorithm that applies to

[*]Alex Nash was supported by the Northrop Grumman Corporation. This material is based upon work supported by, or in part by, NSF under contract/grant number 0413196, ARL/ARO under contract/grant number W911NF-08-1-0468 and ONR in form of a MURI under contract/grant number N00014-09-1-1031. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

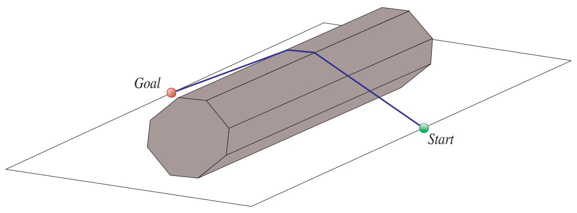

Figure 2: Truly Shortest Path in 3D

any Euclidean solution of the *generate-graph* problem. This is important given that a number of different solutions to the *generate-graph* problem are used in the robotics and video game communities. However, Theta* performs a line-of-sight check for each unexpanded visible neighbor of each expanded vertex and thus it performs many more line-of-sight checks per expanded vertex on 26-neighbor cubic grids than on 8-neighbor square grids. We therefore introduce Lazy Theta*, a variant of Theta* which uses lazy evaluation to perform only one line-of-sight check per expanded vertex (but with slightly more expanded vertices). We show experimentally that Lazy Theta* finds paths faster than Theta* on cubic grids, with one order of magnitude fewer line-of-sight checks and without an increase in path length.

Path Planning in 3D

Path planning is more difficult in continuous 3D environments than it is in continuous 2D environments. Truly shortest paths in continuous 2D environments with polygonal obstacles can be found by performing A* searches on visibility graphs (Lozano-Pérez and Wesley 1979). However, truly shortest paths in continuous 3D environments with polyhedral obstacles *cannot* be found by performing A* searches on visibility graphs because the truly shortest paths are not necessarily formed by the edges of visibility graphs (Choset et al. 2005). This can be seen in Figure 2, where the truly shortest path between the start and goal does not contain any vertex of the polyhedral obstacle. In fact, finding truly shortest paths in continuous 3D environments with polyhedral obstacles is NP-hard (Canny and Reif 1987). Roboticists and video game developers therefore often attempt to find reasonably short paths by performing A* searches on cubic grids. The shortest paths formed by the edges of 8-neighbor square grids can be up to $\approx 8\%$ longer than the truly shortest paths, however we show in this paper that the shortest paths formed by the edges of 26-neighbor cubic grids can be $\approx 13\%$ longer than the truly shortest paths. Thus, neither A* searches on visibility graphs nor A* searches on cubic grids work well in continuous 3D environments. We therefore develop more sophisticated *find-path* algorithms.

Notation and Definitions

Cubic grids are 3D grids composed of (blocked and unblocked) cubic grid cells, whose corners form the set of all vertices V. $s_{start} \in V$ is the start vertex of the search, and $s_{goal} \in V$ is the goal vertex of the search. $\mathcal{L}(s, s')$ is the line segment between vertices s and s', and $c(s, s')$ is the length of this line segment. *lineofsight*(s, s') is true iff vertices s and s' have line-of-sight, that is, the line segment $\mathcal{L}(s, s')$ neither traverses the interior of blocked cells nor passes between blocked cells that share a face. For simplicity, we allow $\mathcal{L}(s, s')$ to pass between blocked cells that share an edge or vertex, but this assumption is not required for the *find-path* algorithms in this paper to function correctly. $nghbr_{vis}(s) \subseteq V$ is the set of visible neighbors of vertex s, that is, neighbors of s with line-of-sight to s.

```
1  Main()
2      $open := closed := \emptyset$;
3      $g(s_{start}) := 0$;
4      $parent(s_{start}) := s_{start}$;
5      $open.Insert(s_{start}, g(s_{start}) + h(s_{start}))$;
6      while open $\neq \emptyset$ do
7          $s := open.Pop()$;
8          [SetVertex($s$)];
9          if $s = s_{goal}$ then
10             return "path found";
11         $closed := closed \cup \{s\}$;
12         foreach $s' \in nghbr_{vis}(s)$ do
13             if $s' \notin$ closed then
14                 if $s' \notin$ open then
15                     $g(s') := \infty$;
16                     $parent(s') := NULL$;
17                 UpdateVertex($s, s'$);
18     return "no path found";
19 end
20 UpdateVertex(s, s')
21     $g_{old} := g(s')$;
22     ComputeCost($s, s'$);
23     if $g(s') < g_{old}$ then
24         if $s' \in$ open then
25             $open.Remove(s')$;
26         $open.Insert(s', g(s') + h(s'))$;
27 end
28 ComputeCost(s, s')
29     /* Path 1 */
30     if $g(s) + c(s, s') < g(s')$ then
31         $parent(s') := s$;
32         $g(s') := g(s) + c(s, s')$;
33 end
```

Algorithm 1: A*

A*

All *find-path* algorithms discussed in this paper build on A* (Hart, Nilsson, and Raphael 1968), shown in Algorithm 1 [Line 8 is to be ignored].[1] To focus its search, A* uses an h-value $h(s)$ for every vertex s, that approximates the goal distance of the vertex. We use the 3D octile distances as h-values in our experiments because the 3D octile distances cannot overestimate the goal distances on 26-neighbor cubic grids if paths are formed by graph edges. The 3D octile dis-

[1] *open.Insert*(s, x) inserts vertex s with key x into the open list. *open.Remove*(s) removes vertex s from the open list. *open.Pop*$()$ removes a vertex with the smallest key from the open list and returns it.

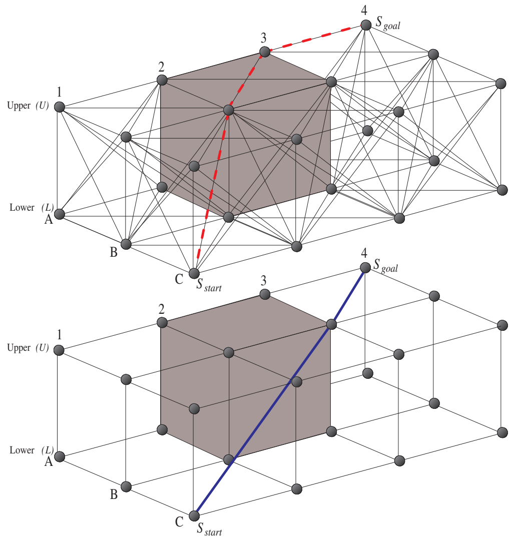

Figure 3: Cubic Grid: Shortest Path Formed by Graph Edges (top) vs. Shortest Vertex Path (bottom)

tances are the distances on 26-neighbor cubic grids without blocked cells. More information on computing the length of the shortest path on a 26-neighbor cubic grid can be found in the next section. A* maintains two values for every vertex s: **(1)** The g-value $g(s)$ is the length of the shortest path from s_{start} to s found so far. **(2)** The parent $parent(s)$ which is used to extract the path after the search halts by following the parent pointers from s_{goal} to s_{start}. A* also maintains two global data structures: **(1)** The open list is a priority queue that contains the vertices considered for expansion. **(2)** The closed list is a set that contains the vertices that have already been expanded. A* updates the g-value and parent of an unexpanded visible neighbor s' of a vertex s in procedure ComputeCost by considering the path from s_{start} to s and from s to s' in a straight line (Line 30). It updates the g-value and parent of s' if this new path is shorter than the shortest path from s_{start} to s' found so far.

Analytical Results

A* finds shortest paths formed by graph edges when used with admissible heuristics. We now prove that the shortest paths formed by the edges of 26-neighbor cubic grids and thus the paths found by A* can be $\approx 13\%$ longer than the truly shortest paths. To prove this result, we introduce the notion of **vertex paths**, which are sequences of (straight) line segments whose end points are vertices. An example of a shortest vertex path on a 26-neighbor cubic grid can be seen in Figure 3 (bottom). The relationship between truly shortest paths and shortest vertex paths is simple since truly shortest paths are at most as long as shortest vertex paths. We argued earlier that the shortest vertex path is equivalent to the truly shortest path for continuous 2D environments (Figure 1 (right)), but that they are not necessarily equivalent for continuous 3D environments. The relationship between shortest vertex paths and shortest paths formed by graph edges is simple as well since shortest vertex paths are at most as long as shortest paths formed by graph edges. For 26-neighbor cubic grids the shortest paths formed by graph edges can be longer than the shortest vertex path, which can be seen in Figure 3 (top and bottom). The question is how large can this difference be. We now prove that the shortest paths formed by the edges of 26-neighbor cubic grids can be up to $\approx 13\%$ longer than the shortest vertex paths and thus can be $\approx 13\%$ longer than the truly shortest paths.

For the proof, consider an arbitrary 26-neighbor cubic grid composed of (blocked and unblocked) cubic grid cells, whose corners form the set of all vertices V. Construct two graphs. Graph $G = (V, E)$ contains an edge (v, w) iff *lineofsight*(v, w). Graph $G' = (V, E')$ contains an edge (v, w) iff v and w are visible neighbors. Let P be a shortest path from s_{start} to s_{goal} in G (that is, a shortest vertex path). Let $d(u, v)$ be the length of this path. We construct a path P' from P which is a shortest path from s_{start} to s_{goal} in G' (that is, a shortest path formed by the edges of the 26-neighbor cubic grid). Let $d'(u, v)$ be the length of this path. Both P and P' are sequences of line segments.

Theorem 1. *If there exists a path P between s_{start} and s_{goal} in G, then $d'(s_{\text{start}}, s_{\text{goal}}) \leq \sqrt{9 - 2\sqrt{2} - 2\sqrt{2}\sqrt{3}} \cdot d(s_{\text{start}}, s_{\text{goal}}) \approx 1.1281 \cdot d(s_{\text{start}}, s_{\text{goal}})$. This bound is asymptotically tight.*

Proof. We apply Lemma 1 (see below) to the sequence of line segments that compose P to yield a sequence of paths in G', each of length at most ≈ 1.1281 times the length of the corresponding line segment. Hence, $d'(s_{start}, s_{goal}) \leq \sqrt{9 - 2\sqrt{2} - 2\sqrt{2}\sqrt{3}} \cdot d(s_{start}, s_{goal}) \approx 1.1281 \cdot d(s_{start}, s_{goal})$. This bound is asymptotically tight because Lemma 1 is a special case of Theorem 1. □

Lemma 1 is a special case of Theorem 1 where path P is a single edge and thus a straight line. Its proof contains two parts: **(1)** We show that the path P' can be chosen to be formed of only edges of cells the interior of which P traverses. This allows us to determine the length $d'(u, v)$ of P' for a given length $d(u, v)$ of P independent of which cells are blocked. **(2)** We maximize the ratio of the lengths of P' and P.

Lemma 1. *If $(u, v) \in E$, then $d'(u, v) \leq \sqrt{9 - 2\sqrt{2} - 2\sqrt{2}\sqrt{3}} \cdot d(u, v) \approx 1.1281 \cdot d(u, v)$. This bound is asymptotically tight.*

Proof. Assume without loss of generality that u is at the origin, that is, $u = (0, 0, 0)$. If the edge $(u, v) \in E$ can be formed by edges in E', then the theorem trivially holds. Otherwise, consider the prefix L of line segment $\mathcal{L}(u, v)$ that ends at the first reached vertex (that is, corner of a cell), which we call $t = (r', u', b')$. Assume without loss of generality that $r' > b' \geq u' \geq 1$ since the 2D case has been analyzed before and the $r' = b'$ case is trivial. We convert L into a step sequence of right (R), back (B) and up (U) steps. R, B and U steps increase the x, z and y coordinate (respectively) by one. We assign coordinates to cells, namely the coordinates of their corners that are farthest away from the origin. We start with the empty step sequence and traverse L from u to t. Whenever we leave the interior of one cell c and enter the interior of another cell c', we append one or two steps to the step sequence depending on the 6 possible differences between the coordinates of c' and c. We append $\underline{\text{R}}$ for $(1,0,0)$, $\underline{\text{B}}$ for $(0,0,1)$, $\underline{\text{U}}$ for $(0,1,0)$, B$\underline{\text{R}}$ for $(1,0,1)$, R$\underline{\text{U}}$ for $(1,1,0)$ and B$\underline{\text{U}}$ for $(0,1,1)$. The resulting step sequence moves from $(1,1,1)$ to $t=(r',u',b')$. The cell with coordinates $(1,1,1)$ and all cells with coordinates that are reached after each underlined step are unblocked because L traverses their interior. The step sequence always has at least one R between two Bs and two Us (or the start and a U), and at least one B between two Us (or the start and a U), and ends in an R (Step Sequence Property).

We convert the step sequence into a move sequence of right (r), right-back (rb) and right-back-up (rbu) moves. r moves increase the x coordinate by one and have length 1; rb moves increase the x and z coordinates by one each and have length $\sqrt{2}$; and rbu moves increase the x, y and z coordinates by one each and have length $\sqrt{3}$. We start with the move sequence that contains a single rbu and use the following algorithm to append moves to the move sequence (the underlined statements are redundant and could be removed):

```
1. Set $storeR := 0$ and set $storeB := 0$
2. IF the next step in the step sequence is R THEN (IF $storeR = 1$
   THEN append r and $\underline{\text{set } storeR := 1}$ ELSE set $storeR := 1$)
3. ELSE (IF the next step in the step sequence is B THEN (IF
   $storeR = 1$ and $storeB = 1$ THEN append rb and set
   $storeR := 0$ and $\underline{\text{set } storeB := 1}$ ELSE set $storeB := 1$))
4. ELSE (IF the next step in the step sequence is U THEN (IF
   $storeR = 1$ and $storeB = 1$ THEN append rbu and set
   $storeR := 0$ and set $storeB := 0$ ELSE append rbu and
   $\underline{\text{set } storeR := 0}$ and set $storeB := 0$ and delete the step after
   U (an R) from the step sequence))
5. Delete the step just processed from the step sequence
6. IF there are steps remaining in the step sequence THEN go to 2
   ELSE (IF $storeR = 1$ and $storeB = 1$ THEN append rb and
   set $storeR := 0$ and set $storeB := 0$ ELSE (IF $storeR = 1$
   THEN append r and set $storeR := 0$))
```

The algorithm does not cover some cases because they are impossible. First, if the next step in the step sequence is B, then it cannot be that $storeR = 0$ and $storeB = 1$. Only a B without any Rs or Us afterwards can result in $storeR = 0$ and $storeB = 1$ (because Rs result in $storeR = 1$, Us result in $storeB = 0$ and only Bs result in $storeB = 1$). Thus, the next step in the step sequence cannot be B due to the Step Sequence Property. Second, if the next step in the step sequence is U, then it cannot be that $storeB = 0$. Only a U (or the start of the step sequence) followed by zero or more Rs can result in $storeB = 0$ (because Bs result in $storeB = 1$). Thus, the next step in the step sequence cannot be U due to the Step Sequence Property. Third, if there are no steps remaining in the step sequence, then it cannot be that $storeR = 0$ and $storeB = 1$, because the step sequence ends in R and thus either $storeR = 1$ (if the algorithm processes R on Line 1) or $storeR = 0$ and $storeB = 0$ (if the algorithm processes R on Line 4).

The algorithm has one requirement (as stated in the algorithm). If the next step in the step sequence is U and it is not the case that $storeR = 1$ and $storeB = 1$, then the step after U in the step sequence must be R. If the next step in the step sequence is U, then we have already shown that $storeB = 1$, which implies $storeR = 0$. Thus, the step before U in the step sequence must have been B (because Rs result in $storeR = 1$, Us result in $storeB = 0$ and only Bs result in $storeB = 1$). Thus, the step after U in the step sequence must be R since it cannot be B or U due to the Step Sequence Property.

By design, the algorithm obeys the following invariant at the end of each iteration. Consider the cell with the coordinates reached from $(1,1,1)$ after the execution of all steps deleted so far. Decrease its x coordinate by one iff $storeR = 1$. Decrease its z coordinate by one iff $storeB = 1$. The move sequence created so far moves from $(0,0,0)$ to exactly these coordinates. In particular, the move sequence at the end of the last iteration moves from $u = (0,0,0)$ to $t = (r',u',b')$. It is a shortest path from u to t in G' due to the following two properties. First, every U step in the step sequence corresponds to an rbu move in the move sequence. (When the algorithm processes the U step, it creates the rbu move.) Every B step in the step sequence that does not correspond to an rbu move corresponds to an rb move. (When the algorithm processes the B step, it either creates the rb move or sets $storeB := 1$ and later creates the rb move.) Thus, the length of the move sequence is $\sqrt{3}u' + \sqrt{2}(b'-u') + (r'-b')$, and no path from u to t in G' can be shorter. Second, it is formed of edges in G' since all moves either traverse the edges, the faces or the interior of the cell with coordinates $(1,1,1)$ or cells with coordinates that are reached after each underlined step, which are known to be unblocked.

Now we begin part 2 of the proof. Let x_3 be the number of moves of length $\sqrt{3}$, x_2 be the number of moves of length $\sqrt{2}$ and x_1 be the number of moves of length 1. We now maximize the ratio of the lengths of L' and L by using Lagrange Multipliers. We want to minimize $f(x_1,x_2,x_3)$ subject to $g(x_1,x_2,x_3) = c$ for some constant c, where $f(x_1,x_2,x_3) = \sqrt{(x_1+x_2+x_3)^2 + (x_2+x_3)^2 + x_3^2} = d(u,t)$ and $g(x_1,x_2,x_3) = x_1 + \sqrt{2}x_2 + \sqrt{3}x_3 = d'(u,t)$, resulting in:

$$L(x_1,x_2,x_3,\lambda) = f(x_1,x_2,x_3) + \lambda(g(x_1,x_2,x_3) - c).$$

We remove the square root from the minimization because it is a monotonic function. $\nabla L = 0$ then implies the system of equations:

$$(1) \qquad \frac{\partial L}{\partial x_1} = 0: \quad 2x_1 + 2x_2 + 2x_3 = -\lambda$$

$$(2) \qquad \frac{\partial L}{\partial x_2} = 0: \quad 2x_1 + 4x_2 + 4x_3 = -\lambda\sqrt{2}$$

$$(3) \qquad \frac{\partial L}{\partial x_3} = 0: \quad 2x_1 + 4x_2 + 6x_3 = -\lambda\sqrt{3}.$$

From Equations 1, 2 and 3 we get $x_3 = \frac{1}{2}(\sqrt{3} - \sqrt{2})(-\lambda)$, $x_2 = \frac{1}{2}(2\sqrt{2} - \sqrt{3} - 1)(-\lambda)$ and $x_1 = \frac{1}{2}(2 - \sqrt{2})(-\lambda)$. The worst case ratio of the lengths of L' and L is:

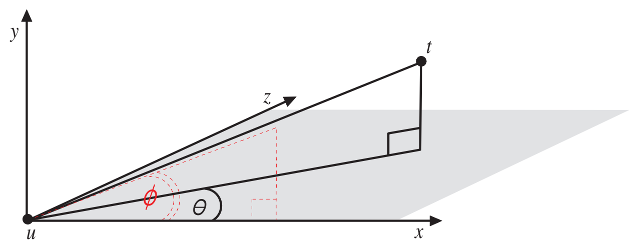

Figure 4: Projection of L onto *x*-*z* and *x*-*y* Planes

$$(4)\quad \frac{d'(u,t)}{d(u,t)} = \frac{x_1+\sqrt{2}x_2+\sqrt{3}x_3}{\sqrt{(x_1+x_2+x_3)^2+(x_2+x_3)^2+x_3^2}}$$
$$= \frac{2-\sqrt{2}+\sqrt{2}(2\sqrt{2}-\sqrt{3}-1)+\sqrt{3}(\sqrt{3}-\sqrt{2})}{\sqrt{1+(\sqrt{2}-1)^2+(\sqrt{3}-\sqrt{2})^2}}$$
$$= \sqrt{9-2\sqrt{2}-2\sqrt{2}\sqrt{3}} \approx 1.1281.$$

This bound is asymptotically tight. Consider a 26-neighbor cubic grid without blocked cells where $s_{start} = (0,0,0)$ and $s_{goal} = (\lfloor x_1\rfloor+\lfloor x_2\rfloor+\lfloor x_3\rfloor, \lfloor x_3\rfloor, \lfloor x_2\rfloor+\lfloor x_3\rfloor)$. As $\lambda \rightarrow -\infty$, $\frac{d'(u,t)}{d(u,t)}$ approaches, but never reaches $\approx$ 1.1281. □

There is an interesting geometric relationship between L and L'. Figure 4 shows the spherical coordinates θ and ϕ, which are the angles of the projection of L onto the *x*-*z* and *x*-*y* planes, respectively. It holds that $\tan(\theta) = (x_2+x_3)/(x_1+x_2+x_3) = \sqrt{2}-1 = \tan(\pi/8)$ and $\tan(\phi) = x_3/(x_1+x_2+x_3) = \sqrt{3}-\sqrt{2}$ for a suitable coordinate system, which implies that $\theta = \pi/8$ and that $\phi = \frac{1}{2} \times \arctan(1/\sqrt{2})$ since $\tan(2\phi) = 2\tan(\phi)/(1-\tan^2(\phi)) = 1/\sqrt{2}$. Thus, θ is exactly half of the smaller angle between $\vec{v_{xz}} = (1,0,1)$ and $\vec{v_x} = (1,0,0)$ (that is, $\pi/4$) and ϕ is exactly half of the smaller angle between $\vec{v_{xz}} = (1,0,1)$ and $\vec{v_{xyz}} = (1,1,1)$ (that is, $\arctan(1/\sqrt{2})$). Thus, L diverges from L' as much as possible.

Equation 4 applies to grids with different numbers of neighbors. For example, we obtain the following known result by setting $x_3 = 0$ and thus eliminating moves of length $\sqrt{3}$: The shortest paths formed by the edges of 8-neighbor square grids can be up to $\sqrt{4-2\sqrt{2}} \approx 8\%$ longer than the shortest vertex paths and thus also the truly shortest paths. The geometric relationship between L and L' continues to hold since $\tan(\theta) = x_2/(x_1+x_2) = \sqrt{2}-1 = \tan(\pi/8)$, which implies $\theta = \pi/8$. This angle is exactly half of the smaller angle between $\vec{v_{xy}} = (1,1)$ and $\vec{v_x} = (1,0)$ (that is, $\pi/4$). Thus, L diverges from L' as much as possible.

A* with Post Smoothing

We proved that the paths found by A* on 26-neighbor cubic grids can be $\approx$ 13% longer than the truly shortest paths. However, simple post processing steps can be used to shorten these paths. For example, A* with Post-Smoothing (A* PS) first runs A* to find a shortest path formed by graph edges and then smoothes this path by repeatedly removing a vertex from the path that is between two vertices on the path with line-of-sight. A* PS typically finds shorter paths than A*, but is not guaranteed to find truly shortest paths because it only considers paths that are formed by graph edges during the search, which often makes post smoothing ineffective (Nash et al. 2007; Ferguson and Stentz 2006). This can be seen in Figure 3 (top), where post smoothing deletes only vertex $A3_U$, resulting in a path that is still longer than the shortest vertex path in Figure 3 (bottom). This insight led to the development of smarter any-angle *find-path* algorithms, such as Theta*.

```
34 ComputeCost(s, s')
35     if lineofsight($parent(s), s'$) then
36         /* Path 2 */
37         if $g(parent(s)) + c(parent(s), s') < g(s')$ then
38             $parent(s') := parent(s)$;
39             $g(s') := g(parent(s)) + c(parent(s), s')$;
40     else
41         /* Path 1 */
42         if $g(s) + c(s, s') < g(s')$ then
43             $parent(s') := s$;
44             $g(s') := g(s) + c(s, s')$;
45 end
```

Algorithm 2: Theta*

Existing Work: Theta*

Theta* is an any-angle *find-path* algorithm that empirically finds shorter paths on square grids than both A* and A* PS with a similar runtime (Nash et al. 2007).[2] Theta* is shown in Algorithm 2. All procedures other than ComputeCost are identical to those of Algorithm 1 and thus are not shown [Line 8 is still to be ignored]. We use the straight line distances $c(s, s_{goal})$ as h-values in our experiments because the 3D octile distances can overestimate the goal distances on 26-neighbor cubic grids if paths do not have to be formed by graph edges. The key difference between Theta* and A* is that Theta* allows the parent of a vertex to be any vertex, while A* restricts the parent of a vertex to be a visible neighbor of that vertex. Theta* is identical to A* except that Theta* updates the g-value and parent of an unexpanded visible neighbor s' of a vertex s in procedure ComputeCost by considering two paths: **Path 1:** Like A*, Theta* considers the path from s_{start} to s and from s to s' in a straight line (Line 42). **Path 2:** To allow for any-angle paths, Theta* also considers the path from s_{start} to $parent(s)$ and from $parent(s)$ to s' in a straight line (Line 37). It considers Path 2 if s' and $parent(s)$ have line-of-sight since Path 2 is guaranteed to be no longer than Path 1 due to the triangle inequality. Otherwise, it considers Path 1. Theta* updates the g-value and parent of s' if the considered path is shorter than the shortest path from s_{start} to s' found so far.

An example trace of Theta* can be seen in Figure 5. Ver-

[2]There is no known analytical bound on the ratio of the lengths of the paths found by Theta* and the truly shortest paths.

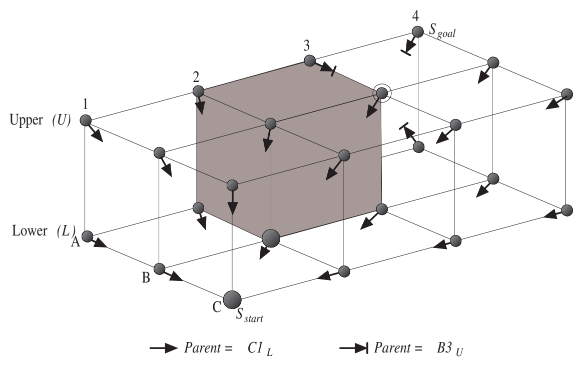

Figure 5: Example Trace of Theta*

```
46 SetVertex(s)
47     if $NOT$ lineofsight($parent(s), s$) then
48         /* Path 1*/
49         $parent(s) := argmin_{s' \in nghbr_{vis}(s) \cap closed}(g(s') + c(s', s))$;
50         $g(s) := min_{s' \in nghbr_{vis}(s) \cap closed}(g(s') + c(s', s))$;
51 end
52 ComputeCost(s, s')
53     /* Path 2 */
54     if $g(parent(s)) + c(parent(s), s') < g(s')$ then
55         $parent(s') := parent(s)$;
56         $g(s') := g(parent(s)) + c(parent(s), s')$;
57 end
```

Algorithm 3: Lazy Theta*

tices are labeled with arrows pointing to their parents, larger spheres represent expanded vertices, and the hollow sphere represents the vertex currently being expanded. The start vertex $C1_L$ is expanded first, followed by $B2_L$ and $B3_U$ (Figure 5). When $B3_U$ with parent $C1_L$ is being expanded, $A3_U$ is an unexpanded visible neighbor of $B3_U$ which does not have line-of-sight to $C1_L$ and thus is updated according to Path 1. $B4_U$ is an unexpanded visible neighbor of $B3_U$ which does have line-of-sight to $C1_L$ and thus is updated according to Path 2.

Extending Theta*: Lazy Theta*

We can easily extend Theta* from an algorithm that only applies to square grids to an algorithm that applies to any Euclidean solution of the *generate-graph* problem (e.g. cubic grids) without any changes to the pseudo code.[3] This is a result of the fact that Theta* is based on the triangle inequality which is guaranteed to hold for any Euclidean solution to the *generate-graph* problem. We only need to adapt its line-of-sight checks to the solution of the *generate-graph* problem. However, Theta* is less efficient on cubic grids than square grids because it performs a line-of-sight check for each unexpanded visible neighbor of each expanded vertex. Line-of-sight checks on grids can be implemented efficiently with line drawing algorithms (Bresenham 1965), but the runtime per line-of-sight check can still be linear in the number of cells and there are many more line-of-sight checks per expanded vertex on 26-neighbor cubic grids than on 8-neighbor square grids. Line-of-sight checks for other solutions of the *generate-graph* problem, such as NavMeshes, typically cannot be implemented as efficiently. We therefore reduce the number of line-of-sight checks that Theta* performs. Our inspiration is provided by PRMs, where lazy evaluation has been used to reduce the number of line-of-sight checks (collision checks) by delaying them until they are absolutely necessary (Bohlin and Kavraki 2000). We therefore introduce Lazy Theta*, a variant of Theta* which uses lazy evaluation to perform only one line-of-sight check per expanded vertex (but with slightly more expanded vertices), while Theta* performs a line-of-sight check for each unexpanded visible neighbor of each expanded vertex. Lazy Theta* is shown in Algorithm 3. All procedures other than ComputeCost are identical to those of Algorithm 1 and thus are not shown [Line 8 is to be executed from now on].

[3]It is important that Theta* be easy to extend given that a number of different solutions to the *generate-graph* problem are used in the robotics and video game communities. While A* can be extended in a similar manner, interpolation based any-angle *find-path* algorithms cannot. For example, Field D* is based on a closed form linear interpolation equation which requires that the search be performed on square grids (Ferguson and Stentz 2006). The extension of Field D* from an algorithm that only applies to square grids to and algorithm that applies to cubic grids requires substantial modifications and additional approximations (Carsten, Ferguson, and Stentz 2006).

Theta* updates the g-value and parent of an unexpanded visible neighbor s' of a vertex s in procedure ComputeCost by considering Path 1 and Path 2. It considers Path 2 if s' and $parent(s)$ have line-of-sight. Otherwise, it considers Path 1. Lazy Theta* optimistically assumes that s' and $parent(s)$ have line-of-sight without performing a line-of-sight check. Thus, it delays the line-of-sight check and considers only Path 2. This assumption may of course be incorrect. Therefore, Lazy Theta* performs the line-of-sight check in procedure SetVertex immediately before expanding vertex s'. If s' and $parent(s')$ indeed have line-of-sight (Line 47), then the assumption was correct and Lazy Theta* does not change the g-value and parent of s'. If s' and $parent(s')$ do not have line-of-sight, then Lazy Theta* updates the g-value and parent of s' according to Path 1 by considering the path from s_{start} to each expanded visible neighbor s'' of s' and from s'' to s' in a straight line and choosing the shortest such path. We know that s' has at least one expanded visible neighbor because s' was added to the open list when Lazy Theta* expanded such a neighbor.

An example trace of Lazy Theta* can be seen in Figure 6, which is annotated in the same manner as Figure 5. When $B3_U$ with parent $C1_L$ is being expanded, $A4_U$ is an unexpanded visible neighbor of $B3_U$. Lazy Theta* optimistically assumes that $A4_U$ has line-of-sight to $C1_L$. $A4_U$ is expanded next. Since $A4_U$ and $C1_L$ do not have line-of-sight, Lazy Theta* updates the g-value and parent of $A4_U$ according to Path 1 by considering the paths from the start vertex $C1_L$ to each expanded visible neighbor s'' of $A4_U$ (namely, $B3_U$) and from s'' to $A4_U$ in a straight line. Lazy

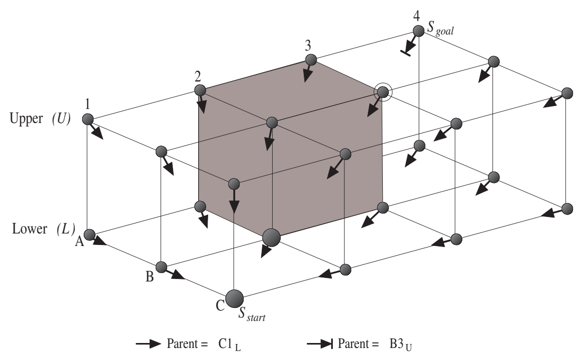

Figure 6: Example Trace of Lazy Theta*

Theta* sets the parent of $A4_U$ to $B3_U$ since the path from $C1_L$ to $B3_U$ and from $B3_U$ to $A4_U$ in a straight line is the shortest such path. In this example, Lazy Theta* and Theta* find the same path from the start vertex $C1_L$ to the goal vertex $A4_U$, but Lazy Theta* performs 4 line-of-sight checks, while Theta* performs 36 line-of-sight checks.

Variants of Lazy Theta*

We now introduce two variants of Lazy Theta* that will help us better understand the performance of Theta* with different lazy evaluation techniques.

- **Lazy Theta*-R:** Lazy Theta*-R is identical to Lazy Theta* except for procedure SetVertex. If Lazy Theta*-R considers a vertex s for expansion and updates it according to Path 1 in procedure SetVertex, it re-inserts s into the open list with an updated key (which we call a **key update**) and continues on Line 7 rather than expanding s. The idea behind deferring the expansion of s is that it gives Lazy Theta*-R the opportunity to discover shorter paths from s_{start} to s since the shortest path from s_{start} to s cannot change once s is expanded.
- **Lazy Theta*-P:** Lazy Theta*-P is identical to A* except for procedure SetVertex. Like A*, Lazy Theta*-P updates the g-value and parent of an unexpanded visible neighbor s' of a vertex s in procedure ComputeCost by considering only Path 1. Unlike A*, Lazy Theta*-P updates the g-value and parent of s' in procedure SetVertex by considering Path 2 immediately before expanding s'. During procedure SetVertex Lazy Theta*-P checks whether or not s' has line-of-sight to $parent(parent(s'))$. If they have line-of-sight, Lazy Theta*-P updates the g-value and parent of s' according to Path 2, namely the path from s_{start} to $parent(parent(s'))$ and from $parent(parent(s'))$ to s' in a straight line. Thus, in procedure ComputeCost, Lazy Theta*-P pessimistically assumes that an unexpanded visible neighbor s' of a vertex s does not have line-of-sight to $parent(s)$, while Lazy Theta* optimistically assumes that it does. Both Lazy Theta*-P and Lazy Theta* then check their assumption in procedure SetVertex and, if necessary, correct it immediately before expanding s'. The idea behind the pessimistic assumption is that it allows Lazy Theta*-P to maintain a desirable property of Theta*, namely that every vertex in the open list has line-of-sight to its parent. This property is required by some variants of Theta*, such as Incremental Phi* (Nash, Koenig, and Likhachev 2009).

Experimental Results

We now compare the average path lengths (Path Length), runtimes (Time), number of vertex expansions (Exp) and number of line-of-sight checks (LOS) of Theta*, Lazy Theta* (Lazy), Lazy Theta*-R (Lazy-R) and Lazy Theta*-P (Lazy-P) when finding paths on $100 \times 100 \times 100$ 26-neighbor cubic grids. We average over 100 path planning problems with different percentages of randomly blocked cells (0%, 5%, 10%, 20% and 30%). The start vertex is always the origin $(0, 0, 0)$, and the goal vertex is $(99, y, z)$ for randomly chosen values of y and z. Table 1 compares, from left to right, the ratio of the lengths of the paths found by A^* and the different any-angle *find-path* algorithms and the ratios of the number of expanded vertices, the number of line-of-sight checks and the runtimes of *Theta** and the other any-angle *find-path* algorithms. Larger values imply better results. We make the following observations. Experiments on smaller grids yield similar results.

- **Path Length:** The lengths of the paths found by all any-angle *find-path* algorithms are similar to one another and shorter than the lengths of the paths found by A*. The shortest paths on 8-neighbor square grids and thus the paths found by A* can be at most $\approx 8\%$ longer than the truly shortest paths. Thus, the paths found by A* can be at most $\approx 8\%$ longer than the paths found by Theta*. Table 1 shows that the shortest paths found by A* on 26-neighbor cubic grids without blocked cells (0%) are on average more than $\approx 8\%$ longer than the shortest paths found by Theta* and thus also the truly shortest paths.
- **Expansions:** Lazy Theta* (which makes optimistic assumptions) and Lazy Theta*-P (which makes pessimistic assumptions) both expand more vertices than Theta* (which makes no assumptions). The assumptions made by both Lazy Theta* and Lazy Theta*-P make their g-values (and thus the keys of the vertices in the open list) less informed than those of Theta*. Lazy Theta* expands more vertices than Lazy Theta*-R, but Lazy Theta*-R performs about two key updates for every vertex that it expands, which increases both its runtime and the number of line-of-sight checks that it performs.
- **Line-of-Sight:** Theta* performs more line-of-sight checks than any of the other any-angle *find-path* algorithms because it does not use lazy evaluation to reduce the number of line-of-sight checks that it performs. Lazy Theta*-R performs more line-of-sight checks than Lazy Theta* and Lazy Theta*-P since it can repeatedly consider a vertex for expansion, each of which requires a line-of-sight check. Lazy Theta*-P performs more line-of-sight checks than Lazy Theta* because it expands more vertices than Lazy Theta*.
- **Runtime:** Theta* has a longer runtime than any of the other any-angle *find-path* algorithms. Lazy Theta* has a shorter runtime than any of the other any-angle *find-path* algorithms, which is not surprising since the runtime is

%	Theta*	Lazy	Lazy-R	Lazy-P	Lazy	Lazy-R	Lazy-P	Lazy	Lazy-R	Lazy-P	Lazy	Lazy-R	Lazy-P
0	1.0836	1.0836	1.0836	1.0836	1.00	1.00	0.45	18.12	18.12	8.11	5.72	5.51	2.33
5	1.0807	1.0803	1.0806	1.0788	0.66	0.95	0.37	11.55	4.87	6.54	1.54	1.04	0.95
10	1.0816	1.0811	1.0816	1.0795	0.81	1.02	0.53	13.04	5.31	8.58	1.66	1.03	1.07
20	1.0761	1.0749	1.0762	1.0719	0.79	1.00	0.54	11.53	5.08	7.88	1.33	0.96	0.98
30	1.0761	1.0748	1.0762	1.0712	0.85	1.04	0.62	11.43	5.09	8.36	1.31	0.92	1.08
	A* Path Length / Path Length				Theta* Exp / Exp			Theta* LOS / LOS			Theta* Time / Time		

Table 1: Experimental Results

heavily influenced by the number of line-of-sight checks and Lazy Theta* performs the fewest line-of-sight checks.

To summarize, the experimental results demonstrate that all variants of Lazy Theta* are superior to Theta* and that Lazy Theta* has the best tradeoff between the number of line-of-sight checks and runtime on one hand and path length on the other hand. Our experimental results may represent a conservative estimate of the runtime advantage of Lazy Theta* over Theta* because our line-of-sight check implementation was optimized by taking advantage of both the techniques described in (Vykruta 2002) and the simplicity of cubic grids. Each Theta* search performed on the order of 50,000 line-of-sight checks. The runtime advantage of Lazy Theta* over Theta* is thus likely larger on solutions of the *generate-graph* problem, such as NavMeshes, where line-of-sight checks may take longer.

Conclusions

We showed in this paper that the shortest paths formed by the edges of 26-neighbor cubic grids and thus the paths found by A* can be $\approx 13\%$ longer than the truly shortest paths. We therefore extended Theta*, an existing any-angle *find-path* algorithm, from square grids to cubic grids. We introduced Lazy Theta*, a variant of Theta*, that propagates information along graph edges (to achieve short runtimes), like A*, but unlike A*, does not constrain paths to be formed by graph edges (to find short "any-angle" paths). Lazy Theta*, unlike Theta*, uses lazy evaluation to perform only one line-of-sight check per expanded vertex (but with slightly more expanded vertices). We showed experimentally that Lazy Theta* finds paths faster than Theta* on 26-neighbor cubic grids, with one order of magnitude fewer line-of-sight checks and without an increase in path length. Theta* and Lazy Theta* apply to any Euclidean solution of the *generate-graph* problem without any changes to the pseudo code, which is important given that a number of different solutions to the *generate-graph* problem are used in the robotics and video game communities.

References

Björnsson, Y.; Enzenberger, M.; Holte, R.; Schaeffer, J.; and Yap, P. 2003. Comparison of different grid abstractions for pathfinding on maps. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Bohlin, R., and Kavraki, L. 2000. Path planning using lazy PRM. In *Proceedings of the IEEE Transactions on Robotics and Automation*.

Bresenham, J. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal* 4:25–30.

Canny, J., and Reif, J. 1987. New lower bound techniques for robot motion planning problems. In *Proceedings of the Symposium on the Foundations of Computer Science*.

Carsten, J.; Ferguson, D.; and Stentz, A. 2006. 3D Field D*: Improved path planning and replanning in three dimensions. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*.

Choset, H.; Lynch, K.; Hutchinson, S.; Kantor, G.; Burgard, W.; Kavraki, L.; and Thrun, S. 2005. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press.

Ferguson, D., and Stentz, A. 2006. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics* 23:79–101.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4:100–107.

Lozano-Pérez, T., and Wesley, M. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communication of the ACM* 22:560–570.

Murphy, R. 2000. *Introduction to AI Robotics*. MIT Press.

Nash, A.; Daniel, K.; Koenig, S.; and Felner, A. 2007. Theta*: Any-angle path planning on grids. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Nash, A.; Koenig, S.; and Likhachev, M. 2009. Incremental Phi*: Incremental Any-Angle Path Planning on Grids. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Patel, A. 2000. *Amit's Game Programming Information*. available online at http://theory.stanford.edu/∼amitp/GameProgramming/MapRepresentations.html.

Tozour, P. 2004. Search space representations. In Rabin, S., ed., *AI Game Programming Wisdom 2*. Charles River Media. 85–102.

Vykruta, T. 2002. Simple and efficient line-of-sight for 3D landscapes. In Rabin, S., ed., *AI Game Programming Wisdom*. Charles River Media. 83–89.

Wooden, D. 2006. *Graph-based Path Planning for Mobile Robots*. Ph.D. Dissertation, Georgia Institute of Technology.