Any-Angle Path Planning

Alex Nash
Northrop Grumman
Integrated Systems
Carson, California 90746, USA
alex.nash@ngc.com

Sven Koenig
Computer Science Department
University of Southern California
Los Angeles, California 90089-0781, USA
skoenig@usc.edu

Abstract

In robotics and video games, one often discretizes continuous terrain into a grid with blocked and unblocked grid cells and then uses path-planning algorithms to find a shortest path on the resulting grid graph. This path, however, is typically not a shortest path in the continuous terrain. In this overview article, we discuss a path-planning methodology for quickly finding paths in continuous terrain that are typically shorter than shortest grid paths. Any-angle path-planning algorithms are variants of the heuristic path-planning algorithm A* that find short paths by propagating information along grid edges (like A*, to be fast) without constraining the resulting paths to grid edges (unlike A*, to find short paths).

Introduction

Path planning is central to many real-world applications since many fundamental problems in computer science can be modeled as path-planning problems (LaValle 2006). In robotics and video games, (continuous) terrain is often discretized into grids with blocked and unblocked grid cells and from there into grid graphs (Tozour 2004; Rabin 2000; Chrpa & Komenda 2011; Björnsson *et al.* 2003; Nash 2012). Our objective is to find short unblocked paths from given start vertices to given goal vertices. All path-planning algorithms trade off differently with respect to their memory consumption, the runtimes of their searches and the lengths of the resulting paths. We are interested only in their runtimes and path lengths since grids typically fit into memory. We discuss only path-planning algorithms that are correct (that is, if they find a path from the start vertex to the goal vertex, it is unblocked) and complete (that is, if there exists an unblocked path from the start vertex to the goal vertex, they find one) but not guaranteed to be optimal (that is, not guaranteed to find a shortest unblocked path from the start vertex to the goal vertex), unless stated otherwise. For example, the heuristic path-planning algorithm A* (Hart, Nilsson, & Raphael 1968) finds shortest grid paths on grids (that is, shortest paths constrained to grid edges). However, shortest grid paths can be unnatural looking and longer than shortest paths because their heading changes are artificially constrained to specific angles, which can result in heading changes in freespace (that is, terrain away from blocked grid cells). Smoothing shortest grid paths (that is, removing unnecessary heading changes from them) after the search typically shortens the paths but does not change the path topologies (that is, the manner in which they circumnavigate blocked grid cells). In this overview article, we discuss a path-planning methodology for quickly finding paths that are typically shorter than shortest grid paths. Any-angle path-planning algorithms are variants of A* that interleave the A* search and the smoothing. They propagate information along grid edges (like A*, to be fast) without constraining the resulting paths to grid edges (unlike A*, to find short paths). The fact that the heading changes on their paths are not artificially constrained to specific angles explains their name, which was coined by Nash et al. (Nash *et al.* 2007). We first analyze how much longer shortest grid paths can be than shortest paths and then discuss any-angle path-planning algorithms in known 2D, known 3D and unknown 2D terrain.

Assumptions

We use video games as the primary motivating application although any-angle path planning (in the form of Field D*) has also been used on mobile robots, including the Mars rovers Spirit, Opportunity and Curiosity (Carsten *et al.* 2009;

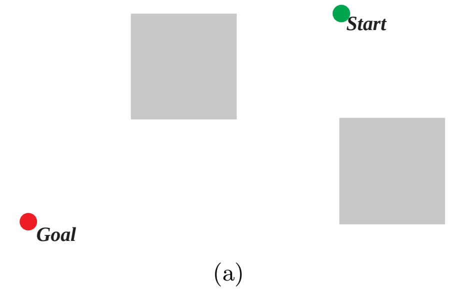


Figure 1: Path-Planning Example in 2D Terrain (a) and Corresponding 2D Grid (b) (Daniel *et al.* 2010)

Ferguson 2013). We assume that the terrain is a grid with grid cells that are either completely blocked (grey) or unblocked (white). Thus, there is no discretization bias (or, synonymously, digitization bias).[1] Vertices are placed at either the corners or centers of grid cells. Grid edges connect all pairs of visible neighbors with straight lines, where two vertices are visible from each other iff the straight line from one vertex to the other vertex does not traverse the interior of any blocked grid cell and does not pass between blocked grid cells that share a side.[2]

Figure 1 shows a path-planning example that we use throughout this article. Its terrain is discretized into a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is A4, and the goal vertex is C1. Figure 2(a) shows a shortest grid path, and Figure 2(b) shows a shortest path. The unnecessary heading change in freespace on the shortest grid path results in an unnatural-looking trajectory for agents such as robots and game characters and makes the shortest grid path longer than the shortest path.

[1]Figure 9(a) depicts an example in which there is discretization bias.

[2]We allow the straight line to pass through diagonally-touching blocked grid cells but can easily relax this restriction.

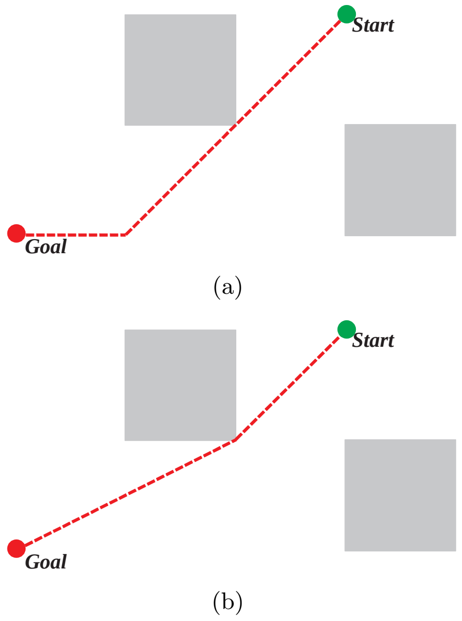


Figure 2: Shortest Grid Path (a) and Shortest Path (b) (Daniel *et al.* 2010)

Path-Length Analysis

We now sketch an analysis which determines how much longer shortest grid paths can be than shortest paths (with the same endpoints) (Nash 2012). It is more general than previous analyses (Nagy 2003; Ferguson & Stentz 2006) because it allows grid cells to be blocked and applies to different types of grids. We differentiate among several types of (unblocked) paths, namely grid paths (that is, paths formed by line segments whose endpoints are visible neighbors), vertex paths (that is, paths formed by line segments whose endpoints are visible vertices) and paths (that is, paths formed by line segments whose endpoints are either visible vertices or non-vertex locations). Shortest paths are no longer than shortest vertex paths, per definition of vertex paths (since they are paths). Shortest vertex paths are no longer than shortest grid paths, per definition of grid paths (since they are vertex paths). The analysis proceeds in two steps:

- First, for every line segment of a shortest vertex path, one shows that the ratio of the lengths of any shortest grid path with the same endpoints as the line segment and the line segment itself is not affected by which grid cells are blocked. This is done by showing that a shortest grid path exists that traverses only the interior of those grid cells that the line segment traverses as well. Since these grid cells cannot be blocked, the analysis

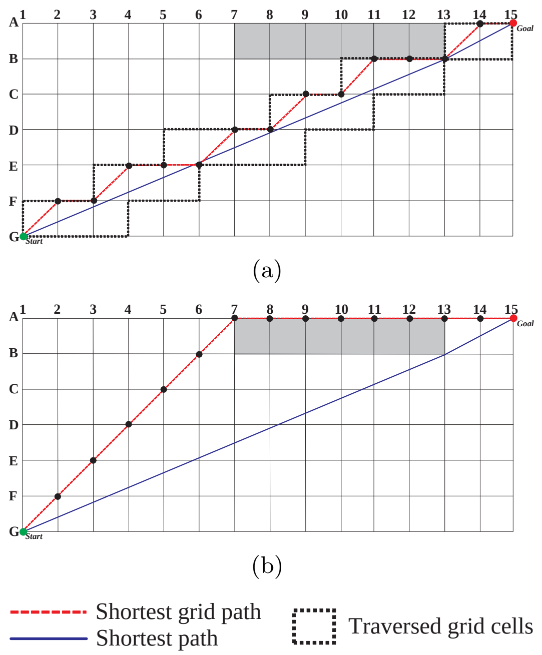


Figure 3: Shortest Grid Paths with Different Path Topologies (a and b) (Nash 2012)

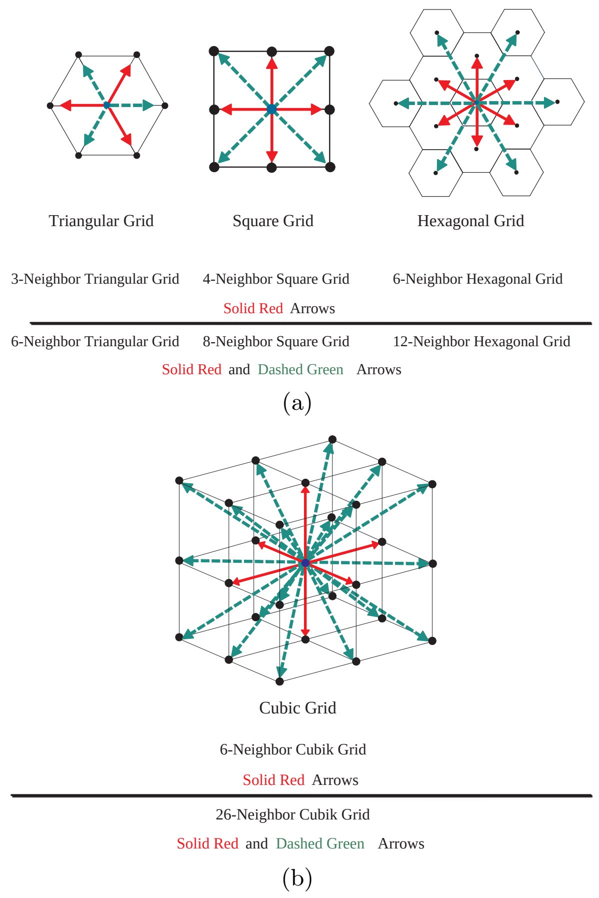


Figure 4: Regular Polygons (a) and Regular Polyhedron (b) (Nash 2012)

does not depend on which grid cells are blocked. Figure 3(a) illustrates this property with a path-planning example where the terrain is discretized into a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is G1, and the goal vertex is A15.

- Second, for all possible endpoints of a line segment, one maximizes the worst-case ratio of the lengths of any shortest grid path with the same endpoints as the line segment and the line segment itself. This can be done by solving an optimization problem with Lagrange multipliers.

This analysis provides upper bounds on the worst-case ratios of the lengths of shortest grid paths and shortest vertex paths (that is, the ratio has at most this value for every path-planning problem). These bounds are either tight (that is, attainable in the sense that there exists a path-planning problem for which the ratio has this value) or asymptotically tight (that is, attainable in the limit as the lengths of the shortest grid paths increase). Shortest vertex paths are of the same lengths as shortest paths on 2D grids with vertices placed at the corners of grid cells (Figure 2(b)), due to our simplifying assumption that grid cells are either completely blocked or unblocked. In this case, the analysis applies unchanged to the worst-case ratios of the lengths of shortest grid paths and shortest paths. Otherwise, the analysis provides (approximate) lower bounds on these worst-case ratios (that is, there exists a path-planning problem for which the ratio has (approximately) at least this value) because shortest paths can then be shorter than shortest vertex paths. In 2D or 3D, shortest paths can be shorter than shortest vertex paths if vertices are placed at the centers of grid cells (the shortest vertex paths then have heading changes in freespace rather than grid cell corners). In 3D, shortest paths can be shorter than shortest vertex paths because the shortest paths can contain heading changes at either the corners or sides of blocked grid cells (we explain this more clearly under "Known 3D Terrain"). Finally, in both 2D and 3D, shortest paths can be shorter than shortest vertex paths if grid cells are not guaranteed to be completely blocked or unblocked.

Only three types of regular (equilateral and equiangular) polygons tessellate 2D terrain, namely tri-

Dimension	Regular Grid	Neighbors	Ratio	In Relation to Shortest Vertex Paths	In Relation to Shortest Paths
2D	triangular grid with vertices at corners	3-neighbor	$= 100\%$	tight	tight
		6-neighbor	$\approx 15\%$	tight	tight
	square grid with vertices at corners	4-neighbor	$\approx 41\%$	tight	tight
		8-neighbor	$\approx 8\%$	asymptotically tight	asymptotically tight
	hexagonal grid with vertices at centers	6-neighbor	$\approx 15\%$	tight	lower bound
		12-neighbor	$\approx 4\%$	asymptotically tight	approximate lower bound
3D	cubic grid with vertices at corners	6-neighbor	$\approx 73\%$	tight	lower bound
		26-neighbor	$\approx 13\%$	asymptotically tight	approximate lower bound

Table 1: Path-Length Analysis of Shortest Grid Paths (Nash 2012)

angles (resulting in triangular grids), squares (resulting in square grids) and hexagons (resulting in hexagonal grids) (Figure 4(a)). Table 1 shows results for 2D 3-neighbor (solid red arrows in Figure 4(a)) and 2D 6-neighbor (solid red and dashed green arrows in Figure 4(a)) triangular grids with vertices placed at the corners of grid cells, 2D 4-neighbor (solid red arrows) and 2D 8-neighbor (solid red and dashed green arrows) square grids with vertices placed at the corners of grid cells (the 8-neighbor variant of which is, for example, used by robots (Carsten *et al.* 2009) and the video game Company of Heroes by Relic Entertainment) and 2D 6-neighbor (solid red arrows) and 2D 12-neighbor (solid red and dashed green arrows) hexagonal grids with vertices placed at the centers of grid cells (the 6-neighbor variant of which is, for example, used by robots (Chrpa & Komenda 2011) and the video game Sid Meier's Civilization V by Firaxis Games). Only one type of regular polyhedron tessellates 3D terrain, namely cubes (resulting in cubic grids) (Figure 4(b)). Table 1 shows results for 3D 6-neighbor (solid red arrows in Figure 4(b)) and 3D 26-neighbor (solid red and dashed green arrows in Figure 4(b)) cubic grids with vertices placed at the corners of grid cells.

Most percentages listed in the table are approximate because the actual percentages are irrational. For example, shortest grid paths on 2D 8-neighbor square grids with vertices placed at the corners of grid cells can be at least a factor of $2/\sqrt{2+\sqrt{2}} \approx 1.08$ (that is, approximately eight percent) longer than shortest paths (but not more), while shortest grid paths on 3D 26-neighbor cubic grids with vertices placed at the corners of grid cells can be at least a factor of $\sqrt{9-2\sqrt{2}-2\sqrt{2}\sqrt{3}} \approx 1.13$ (that is, approximately 13 percent) longer than shortest paths. These results suggest that it might be necessary to find shorter paths than shortest grid paths. In case the reader feels as though these percentages are insignificant, it is important to understand that on non-grid terrain discretizations (Figure 9) the worst-case ratios of the lengths of shortest "grid" paths and shortest paths can be larger.

We use 2D 8-neighbor square grids and 3D 26-neighbor cubic grids throughout the remainder of this article, both with vertices placed at the corners of grid cells. These cases allow us to generalize from 2D to 3D terrain, and their bounds on the worst-case ratios of the lengths of shortest grid paths and shortest paths are sufficiently small to make path planning on grids a strong competitor of any-angle path planning.

A*

All path-planning algorithms that we discuss are based on the heuristic path-planning algorithm A* (Hart, Nilsson, & Raphael 1968), which is probably the most popular path-planning algorithm in artificial intelligence and widely used in robotics and video games. Figure 5(a) shows the pseudo code of A*.[3] For the description of A*, we assume that all paths are constrained to the edges of the graph given by the neighbor relationship of vertices. To focus its search, A* requires a user-provided h-value (or, synonymously, heuristic value) $h(s)$ for every vertex s, that is an estimate of the goal distance of s (that is, the length of a shortest path from s to the goal vertex). The h-values are required to be consistent (that is, satisfy the triangle inequality) for our version of the pseudo code and, as a consequence, are admissible (that is, do not overestimate the goal distances of the vertices). A* maintains two values for every vertex s: **(1)** Its g-value $g(s)$ is an estimate of the start distance of s (that is, the

[3] In the pseudo code, s_{start} is the start vertex, and s_{goal} is the goal vertex. $\mathit{lineofsight}(s, s')$ is true iff vertices s and s' are visible from each other. $\mathit{nghr}_{vis}(s)$ is the finite set of visible neighbors of vertex s. $\mathit{open.Insert}(s, x)$ inserts vertex s with key x into priority queue open, $\mathit{open.Remove}(s)$ removes s from priority queue open, and $\mathit{open.Pop}()$ removes a vertex with the smallest key from priority queue open and returns it. Finally, $\arg\min_{x\in X} f(x)$ returns a value y such that $\min_{x\in X} f(x) = y$.

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

(a) A*

```
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          if $s = s_{goal}$ then
9              return "path found";
10         $closed := closed \cup \{s\}$;
11         foreach $s' \in nghbr_{vis}(s)$ do
12             if $s' \notin$ closed then
13                 if $s' \notin$ open then
14                     $g(s') := \infty$;
15                     $parent(s') := NULL$;
16                 UpdateVertex$(s, s')$;
17     return "no path found";
18 end
19 UpdateVertex(s, s')
20     $g_{old} := g(s')$;
21     ComputeCost$(s, s')$;
22     if $g(s') < g_{old}$ then
23         if $s' \in$ open then
24             $open.Remove(s')$;
25         $open.Insert(s', g(s') + h(s'))$;
26 end
27 ComputeCost(s, s')
28     if lineofsight$(parent(s), s')$ then
29         /* Path 2 */
30         if $g(parent(s)) + c(parent(s), s') < g(s')$
           then
31             $parent(s') := parent(s)$;
32             $g(s') := g(parent(s)) + c(parent(s), s')$;
33     else
34         /* Path 1 */
35         if $g(s) + c(s, s') < g(s')$ then
36             $parent(s') := s$;
37             $g(s') := g(s) + c(s, s')$;
38 end
```

(b) Theta*

```
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 2 */
30     if $g(parent(s)) + c(parent(s), s') < g(s')$ then
31         $parent(s') := parent(s)$;
32         $g(s') := g(parent(s)) + c(parent(s), s')$;
33 end
34 SetVertex(s)
35     if $NOT$ lineofsight$(parent(s), s)$ then
36         /* Path 1*/
37         $parent(s) :=$
           $argmin_{s' \in nghbr_{vis}(s) \cap closed}(g(s') + c(s', s))$;
38         $g(s) := min_{s' \in nghbr_{vis}(s) \cap closed}(g(s') + c(s', s))$;
39 end
```

(c) Lazy Theta*

Figure 5: Pseudo Code (Nash 2012)

length of a shortest path from the start vertex to s), namely the length of the shortest path from the start vertex to s that it has found so far. A* uses its g-value to calculate its f-value $f(s) = g(s) + h(s)$, which is an estimate of the length of a shortest path from the start vertex via s to the goal vertex. **(2)** Its parent $parent(s)$ is used to extract the resulting path after the A* search terminates. A* also maintains two global data structures: **(1)** The open list *open* is a priority queue that contains the vertices that A* considers to expand with their f-values as their keys. **(2)** The closed list *closed* is a set that contains the vertices that A* has already expanded and thus can be used to ensure that all vertices are expanded at most once. A* expands all vertices at most once and thus does not depend on the closed list if the h-values are consistent (Pearl 1985) since the f-values of all vertices along all branches of its search trees are then non-decreasing. However, any-angle path-planning algorithms typically do not have this property and thus rely on the closed list to prevent them from expanding vertices multiple times.

A* sets the g-value of every vertex to infinity and the parent of every vertex to NULL when it encounters the vertex for the first time [lines 14-15]. It sets the g-value of the start vertex to zero and the parent of the start vertex to the start vertex [lines 3-4]. It sets the open and closed lists to the empty list and then inserts the start vertex into the open list with its f-value as its key [lines 2 and 5]. A* then repeatedly executes the following procedure: If the open list is empty, then A* reports that there exists no path [line 17]. Otherwise, it removes a vertex s with the smallest f-value from the open list [line 7]. (It typically breaks ties among vertices with the same f-value in the open list in favor of vertices with larger g-values since this often reduces the number of vertex expansions and thus also the run time.) If this vertex is the goal vertex, then A* reports that it has found a path [line 9]. Path extraction (not shown in the pseudo code) follows the parents from the goal vertex to the start vertex to retrieve a path from the start vertex to the goal vertex in reverse, the length of which is equal to the g-value of the goal vertex. Otherwise, A* expands the vertex by inserting it into the closed list [line 10] and generating each of its unexpanded visible neighbors s', as follows: A* checks whether $g(s) + c(s, s')$ (where $c(s, s') > 0$ is the distance from s to s') is smaller than $g(s')$. If so, then it sets the parent of s' to s [line 30], sets $g(s')$ to $g(s) + c(s, s')$ [line 31] and finally inserts s' into the open list with its f-value as its key [line 25] or, if it was already in the open list, sets its key to its f-value [lines 23-25]. A* then repeats this procedure.

To summarize, A* updates the g-value and parent of each unexpanded visible neighbor s' of the vertex s that is currently being expanding as follows (in procedure ComputeCost): A* considers setting the parent of s' to s, resulting in a path of length $g(s)+c(s, s')$ from the start vertex to s and from there to s' in a straight line. A* updates the g-value and parent of s' if the length of this path is shorter than the length $g(s')$ of the shortest path from the start vertex to s' that it has found so far, namely the path that results (in reverse) from following the parents from s' to the start vertex.

Known 2D Terrain

Many agents operate in known 2D terrain.

Conventional Path-Planning Algorithms

We first discuss how A* operates on grid graphs and visibility graphs. The resulting trade-offs between the runtimes of its searches and the lengths of the resulting paths are at opposite ends of the spectrum. We then briefly discuss other conventional path-planing algorithms.

A* on Grid Graphs: A* on Grid Graphs is fast since it propagates information along grid edges, the number of which grows at most linearly in the number of grid cells (or vertices). It also finds shortest grid paths if the h-values are consistent, is simple and applies to every graph embedded in 2D or 3D terrain. Therefore, it is not surprising that A* on Grid Graphs is popular (Björnsson *et al.* 2003; Yap 2002). For example, the game characters in the video games Warcraft II: Tides of Darkness by Blizzard Entertainment and Climax Studios and Starcraft by Blizzard Entertainment and Mass Media seem to move on the grid edges of 2D 8-neighbor square grids.

Figure 6 shows A* on Grid Graphs in operation on a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is A4, and the goal vertex is C1. We use the straight-line distances as h-values. Arrows point to the parents of vertices. A red circle indicates the vertex that is currently being expanded, and a blue arrow indicates a vertex that is being generated during the current vertex expansion. A* expands start vertex A4, followed by B3 and C2. It terminates when it is about to expand goal vertex C1. Path extraction then retrieves the shortest grid path [A4, B3, C2, C1] from start vertex A4 to goal vertex C1.

A* on Grid Graphs can find shortest paths. However, this is not guaranteed, as shown in Figure 6 where the resulting path has an unnecessary heading change in freespace at C2 and is longer than the shortest path [A4, B3, C1] from start vertex A4 to goal vertex C1. We have explained how much longer shortest grid paths can be than shortest paths under "Path-Length Analysis."

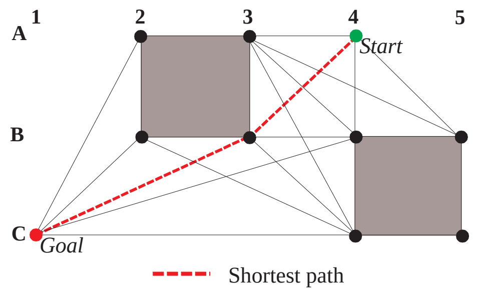


Figure 7: Visibility Graph (Daniel *et al.* 2010)

A* on Visibility Graphs: One constructs visibility graphs (Lee 1978; Lozano-Pérez & Wesley 1979) as follows: The vertices are placed at the convex corners of all obstacles and at the locations of the start and goal vertices. Visibility graph edges connect all pairs of visible vertices with straight lines. A* on Visibility Graphs finds shortest paths in 2D terrain with polygonal obstacles (Figure 7) (Lozano-Pérez & Wesley 1979). A shortest path from the start vertex to the goal vertex is part of the visibility graph and, since visibility graphs are a subset of the set of all vertex paths in 2D terrain with vertices placed at the corners of grid cells, also a vertex path on 2D grids with vertices placed at the corners of grid cells, which is the reason why shortest vertex paths are of the same lengths as shortest paths on 2D grids with vertices placed at the corners of grid cells.

However, A* on Visibility Graphs also has disadvantages. It can be slow since it propagates information along visibility graph edges, the number of which can grow quadratically in the number of grid cells, resulting in A* searches with large branching factors and many visibility checks. Sophisticated variants of A* on Visibility Graphs (Liu & Arimoto

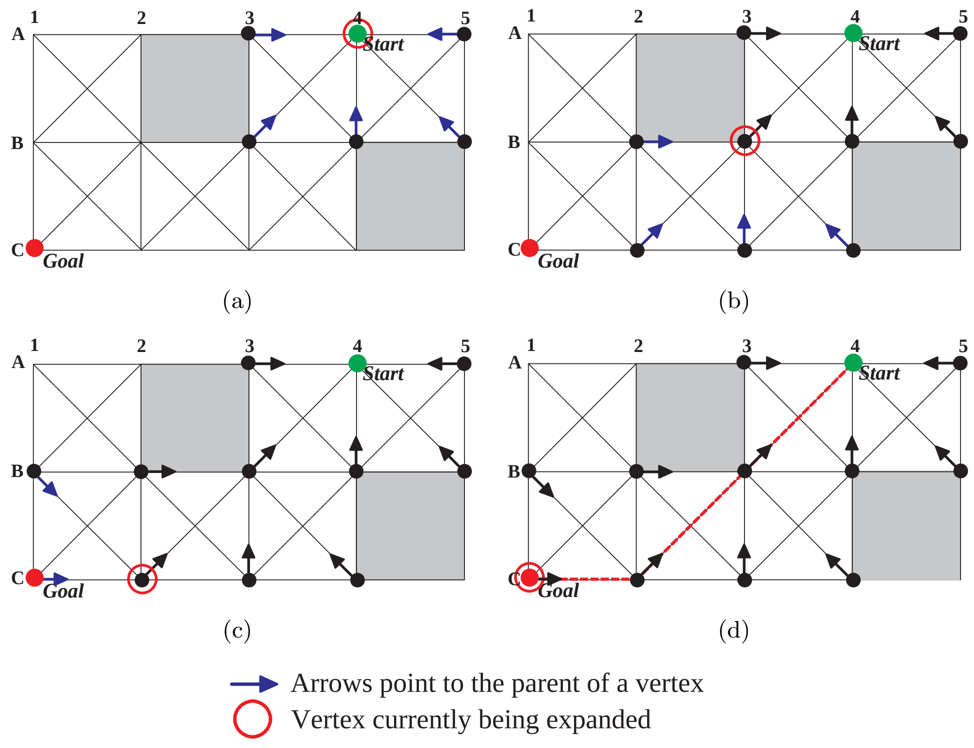

1
2
3
4
5
A
B
C
Start
Goal
(a)
(b)
(c)
(d)
Arrows point to the parent of a vertex
Vertex currently being expanded

Figure 6: Execution Trace of A* on Grid Graphs (Nash 2012)

1992; Mitchell & Papadimitriou 1991) can decrease the number of visibility checks. Path-planning algorithms such as Continuous Dijkstra and its variants (Mitchell, Mount, & Papadimitriou 1987; Hershberger & Suri 1999) as well as the recent Anya (Harabor & Grastien 2013) (which requires neither preprocessing nor large amounts of memory) also find shortest paths but have not yet been thoroughly evaluated experimentally.

Probabilistic Path-Planning Algorithms: Probabilistic path-planning algorithms, such as probabilistic roadmaps (Kavraki *et al.* 1996), or their special case, rapidly exploring random trees (LaValle & Kuffner 2001), discretize terrain by placing vertices randomly in the terrain. Roadmap edges connect some or all pairs of visible vertices with straight lines. Probabilistic path-planning algorithms then find paths on the resulting graphs with A* (or some other conventional path-planning algorithm) or, in case of trees, by reading them off directly. They are only probabilistically complete, can find paths that have heading changes in freespace and can be slow in the presence of narrow passages. Some researchers now advocate a systematic (rather than random) sampling of terrain to determine the locations of the vertices to mitigate these shortcomings (Lindemann & LaValle 2004).

Any-Angle Path-Planning Algorithms

We now discuss any-angle path-planning algorithms. A* on Grid Graphs finds long paths but is fast, while A* on Visibility Graphs finds short paths but is slow. Any-angle path-planning algorithms try to combine the best of both worlds. They are variants of A* that find paths by propagating information along grid edges (like A* on Grid Graphs, to be fast) without constraining the resulting paths to grid edges (like A* on Visibility Graphs, to find short paths). They are not typically guaranteed to find shortest paths. The asterisk in their names thus does not denote their optimality but rather their similarity to A*. Any-angle path-planning algorithms should aim for the following three properties:

- **Efficiency**: Any-angle path-planning algorithms should be faster than A* on Visibility Graphs and find shorter paths than A* on Grid Graphs (Figure 8). Different any-angle path-planning algorithms trade off differently between the runtimes of their searches and the lengths of the

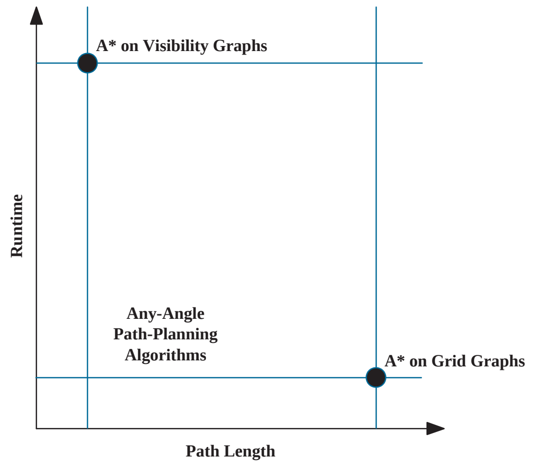


Figure 8: Runtimes versus Path Lengths

resulting paths. We do not provide a comprehensive quantitative analysis of this trade-off since comprehensive experimental comparisons are currently missing from the literature although we broadly average over all reported results to give the reader an approximate idea of the efficiency of the different any-angle path-planning algorithms. However, we encourage the reader to examine the literature themselves before drawing any conclusions, due to the following issues: First, the experimental setups (such as the type of grid, grid size, placement of blocked grid cells, locations of start and goal vertices, h-values and tie-breaking rule for selecting a vertex from those with the smallest f-value in the open list) can have large effects on the runtimes of the searches and the lengths of the resulting paths. Currently, there is no agreement on standard experimental setups in the literature. Second, measuring runtimes is especially difficult. Runtime proxies, such as the number of vertex expansions, cannot be used since different any-angle path-planning algorithms perform different operations when expanding a vertex and thus have different runtimes per vertex expansion. Furthermore, they typically operate on path-planning problems that fit into memory and are thus small. Therefore, big-O analyses are not meaningful, and implementation choices (such as data structures and coding details) can have large effects on the runtimes. It is currently unclear how to address these issues best.

- **Simplicity**: Any-angle path-planning algorithms should be simple to understand, implement, debug and extend.

- **Generality**: Any-angle path-planning algorithms should apply to every graph embedded in 2D or 3D terrain, independent of the terrain-discretization technique used. Generality is important because different video games use different terrain-discretization techniques (Figure 9) (Tozour 2008; Champandard 2010). For example, the video games Company of Heroes by Relic Entertainment and Sid Meier's Civilization V by Firaxis Games use regular grids. The video games Halo 2 by Bungie Studios, Counter-Strike: Source by Valve Corporation and Metroid Prime by Retro Studios and Nintendo use navigation meshes (that is, tessellations of terrain into n-sided convex polygons). Finally, the video game MechWarrior 4: Vengeance by FASA Interactive uses circle-based waypoint graphs (that is, graphs with circles around vertices that indicate freespace).

A* with Post Smoothing: A simple any-angle path-planning algorithm can be obtained as follows: One first executes A* on Grid Graphs and then uses simple post-processing techniques to smooth (that is, remove unnecessary heading changes) and thus shorten the path (at the expense of being slower). Smoothing has to be fast. There exist many ways to do that (Thorpe 1984; Botea, Müller, & Schaeffer 2004; Millington & Funge 2009). For example, A* with Post Smoothing first runs A* on Grid Graphs to find a shortest grid path and then smoothes this grid path in a post-processing step by repeatedly removing a vertex from the path that lies between two visible vertices on the path. This cannot make the path longer due to the triangle inequality.

Figure 10 shows A* with Post Smoothing in operation on a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is A4, and the goal vertex is C1. It runs A* on Grid Graphs to find the shortest grid path [A4, B3, C2, C1]. It removes B2 in the post-processing step, then unsuccessfully tries to remove B3 and then terminates. Path extraction then retrieves the shortest path [A4, B3, C1] from start vertex A4 to goal vertex C1.

A* with Post Smoothing typically finds shorter paths than A* on Grid Graphs and can find shortest paths (Figure 10). However, this is not guaranteed. Since its A* search considers only grid paths, it cannot make informed decisions regarding other paths (Daniel *et al.* 2010; Ferguson & Stentz 2006). Smoothing typically leaves the topologies of the paths unchanged and is thus not guaranteed to find shortest paths. For example, the post-processing step of A* with Post Smoothing does

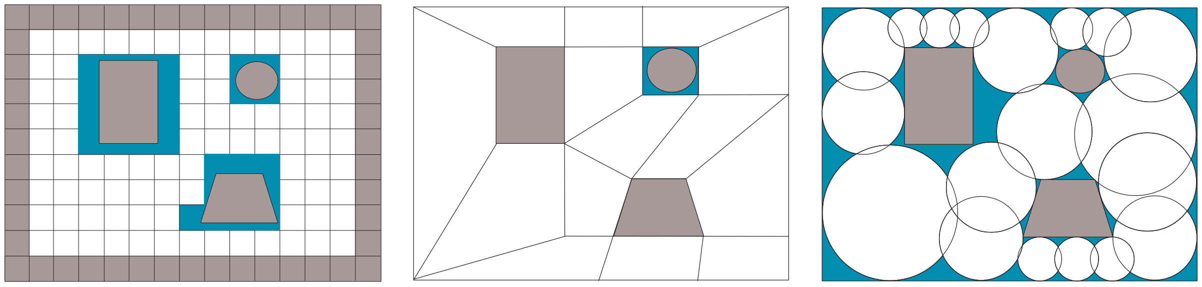
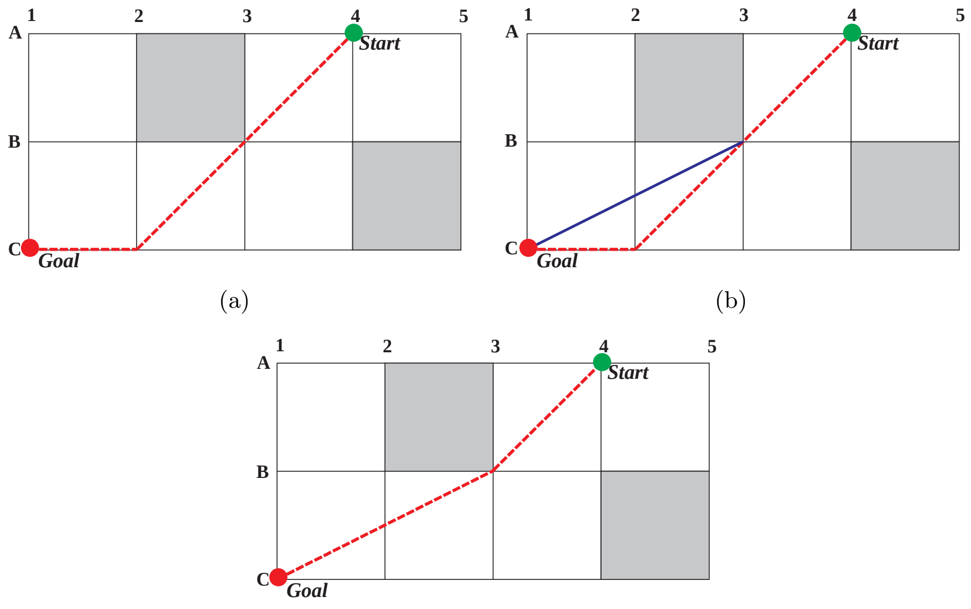

(a) Regular Grids (b) Navigation Meshes (c) Circle-Based Waypoint Graphs

Figure 9: Terrain Discretizations (Nash 2012)



(a) (b) (c)

Figure 10: Execution Trace of A* with Post Smoothing

not smooth the shortest grid path in Figure 3(b) at all but smoothes the shortest grid path in Figure 3(a) to the shortest path. However, the A* search of A* with Post Smoothing has no bias for one or the other and could thus find either shortest grid path. This suggests that one might want to (either perform the smoothing before the A* search or) interleave the smoothing with the A* search because the A* search then considers more than just grid paths during the search.

There exist many ways of interleaving the A* search with the smoothing. We discuss three of them in this article, resulting in different any-angle path-planning algorithms, namely Block A*, Field D* and Theta*. They trade off differently between the runtimes of their searches and the lengths of the resulting paths. Block A* uses a lookup table with precomputed short paths within given sets of grid cells. Field D* uses interpolation between the g-values of vertices to calculate the g-values of non-vertex locations, which allows it to set the parent of a vertex to any vertex or non-vertex location on the straight line between the neighbors of the vertex. Finally, Theta* checks for shortcuts during the expansion of a vertex by checking whether it can set the parent of each unexpanded visible neighbor of the vertex that is currently being expanded to the parent of the expanded vertex rather than the expanded vertex itself. We describe Theta* in more detail than the other any-angle path-planning algorithms simply because we, as the developers,

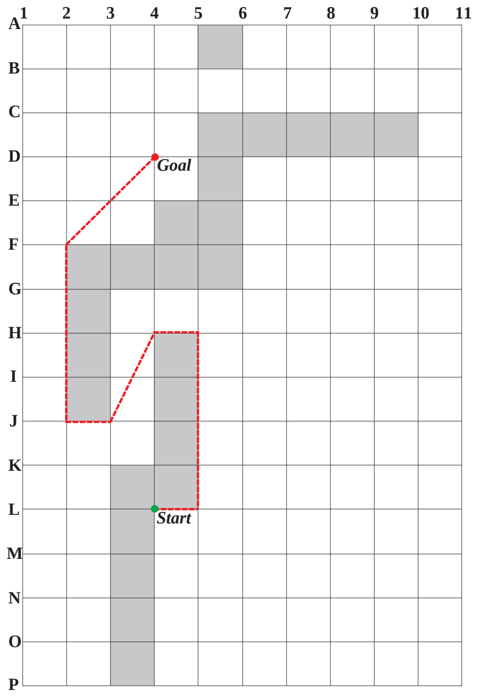


Figure 11: Execution Trace of Block A* (Yap *et al.* 2011b)

are very familiar with it. There likely exist additional any-angle path-planning algorithms, both new ones that still need to be discovered and existing ones that still need to be characterized as any-angle path-planning algorithms.

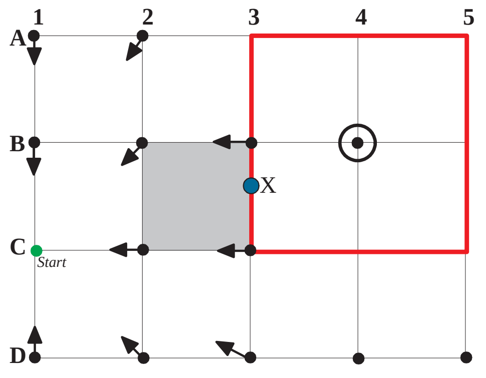
Block A*: Block A* (Yap *et al.* 2011b; 2011a) performs the smoothing before the A* search by using a lookup table with (the lengths of) precomputed short paths within given sets of grid cells. It partitions a 2D square grid into blocks of equal blocksize, uses an A* search that expands blocks rather than vertices (by putting blocks onto the open list) and, for every block, precomputes paths from every fringe vertex of the block (that is, every vertex along the border of the block) to every other fringe vertex of the block and stores (them and) their lengths in a lookup table to speed up the A* search. These paths can be shortest grid paths, shortest paths or any other paths. Block A* becomes an any-angle path-planning algorithm if the paths are precomputed with an any-angle path-planning algorithm.

Figure 11 shows Block A* in operation on a 2D square grid with vertices placed at the corners of grid cells, which is partitioned into six 5×5 blocks. The lookup table contains (the lengths of) shortest paths. The start vertex is L4, and the goal vertex is D4. The h-values are straight-line distances. When Block A* expands the start block K1-K6-P6-P1 (that is, the block that contains start vertex L4), it basically sets the g-value of each fringe vertex s of the start block to the length of a shortest path from start vertex L4 to s within the start block. It then generates each of the neighboring blocks of the start block (that is, the blocks that the start block shares at least one fringe vertex with), namely blocks F1-F6-K6-K1, K6-K11-P11-P6 and F6-F11-K11-K6, as follows: Block A* calculates the smallest f-value of all those fringe vertices of the neighboring block whose g-values decreased or were calculated for the first time. It inserts the neighboring block into the open list with this value as its key or, if it was already in the open list, sets its key to this value provided this decreases the key. (Thus, Block A* can re-expand blocks.) Block A* expands block F1-F6-K6-K1 next since it is the block in the open list with the smallest key. Block A* basically sets the g-value of each fringe vertex s of block F1-F6-K6-K1 to the minimum of the g-value of its fringe vertex s' plus the length of a shortest path from s' to s within block F1-F6-K6-K1 (which it retrieves from the lookup table), minimized over all of its fringe vertices s'. It then generates each of the neighboring blocks of block F1-F6-K6-K1 as before, and so on. Finally, path extraction retrieves the shortest path [L4, L5, H5, H4, J3, J2, F2, D4] from start vertex L4 to goal vertex D4 (Figure 11).

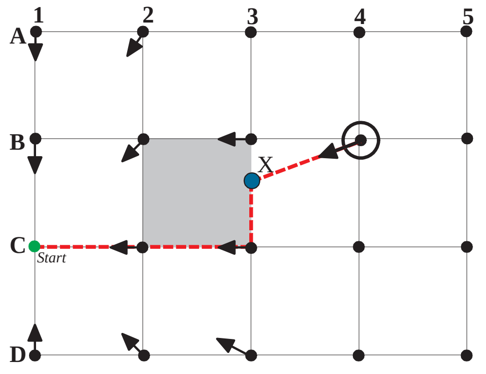
Block A* is fast and could be extended to all types of 2D grids but also has disadvantages: Block A* can find shortest paths (Figure 11). However, this is not guaranteed. For example, its paths can have heading changes in freespace (namely, at fringe vertices) and could thus be smoothed in a post-processing step. Block A* must be implemented with care because its lookup table can consume a lot of memory if it is not compressed. Finally, it can be difficult to determine the blocksize that trades off best between the runtimes of its searches and the lengths of the resulting paths.

Field D*: Field D* (Ferguson & Stentz 2006) is a variant of D* Lite (Koenig & Likhachev 2005) or D* (Stentz 1995) that interleaves the smoothing with the A* search by using interpolation between the g-values of vertices to calculate the g-values of non-vertex locations, which allows it to set the parent of a vertex to any vertex or non-vertex location on the straight line between the neighbors of the vertex.

The difference between Field D* and A* when they update the g-value and parent of an (for Field D* not necessarily) unexpanded visible neighbor s' of



(a)



(b)

Figure 12: Execution Trace of Field D* (Daniel *et al.* 2010)

the vertex s that is currently being expanded, is the following: Field D* considers setting the parent of s' to any vertex or non-vertex location X' on the perimeter of s' that is visible from s' (where the perimeter is the square formed by connecting the neighbors of s'), resulting in a path of length $g(X') + c(X', s')$ from the start vertex to X' and from there to s' in a straight line. It updates the g-value and parent of s' if the length of the shortest such path is smaller than the length $g(s')$ of the shortest path from the start vertex to s' that it has found so far.

Figure 12 shows Field D* in operation on a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is C1. The perimeter of $s' = B4$ is the red square with the thick border. Consider non-vertex location X on the perimeter. Field D* does not know the g-value of X since it stores g-values only for vertices. It calculates the g-value of X using linear interpolation between the g-values of the two vertices on the perimeter that are closest to X. Therefore, it linearly interpolates between $g(B3) = 2.41$ and $g(C3) = 2.00$, resulting in $g(X) = 0.55 \times 2.41 + 0.45 \times 2.00 = 2.23$ since 0.55 and 0.45 are the distances from X to B3 and C3, respectively. The calculated g-value of X is different from the length of a shortest path from the start vertex to X, namely 2.55, even though the g-values of B3 and C3 are both equal to the lengths of shortest paths from the start vertex to them, respectively. The reason for this mistake is that there exist shortest paths from start vertex C1 via either C3 or B3 to B4. Therefore, linear interpolation predicts that there must also exist a short path from start vertex C1 via every non-vertex location along the grid edge that connects B3 and C3 to B4. However, this is not the case since these paths have to circumnavigate blocked grid cell B2-B3-C3-C2, which makes them longer than expected. Field D* then finds the vertex or non-vertex location X' on the perimeter of B4 that is visible from B4 and minimizes the length $g(X') + c(X', B4)$ of the path from start vertex C1 to X' and from there to B4 in a straight line. There exist infinitely many vertex or non-vertex locations X' on the perimeter. However, the optimization problem can be solved quickly since the vertex or non-vertex locations that minimize $g(X') + c(X', B4)$ on each of the eight grid edges that comprise the perimeter of B4 can be found using closed form optimization equations (although the calculations require floating point operations that make Field D* slow). One can modify Field D* so that it applies to types of graphs other than 2D square grids, such as 2D triangular meshes, by changing the optimization equations (Sapronov & Lacaze 2008; Perkins *et al.* 2012).

Field D* has a disadvantage (Figure 12): As a result of miscalculating the g-value of X, Field D* sets the parent of B4 to X, resulting in a path that has an unnecessary heading change at X and is longer than even a shortest grid path. Field D* uses a 1-step lookahead post-processing technique during path extraction after the search to avoid some of these heading changes (Ferguson & Stentz 2006), such as the one depicted in Figure 12, but does not eliminate all of them. The resulting paths typically have lots of small heading changes in freespace and could thus be smoothed further in an additional post-processing step.[4]

[4]Figure 12 highlights both the operation of Field D* and its disadvantages to save space. The work of Ferguson and Stentz (Ferguson & Stentz 2006) contains additional examples of the operation of Field D*.

Theta*: Theta* (Nash *et al.* 2007) interleaves the smoothing with the A* search by checking for shortcuts during the expansion of a vertex, namely whether it can set the parent of each unexpanded visible neighbor of the vertex that is currently being expanded to the parent of the expanded vertex rather than the expanded vertex itself. Figure 5(b) shows the pseudo code of Theta* (the differences between the pseudo code of A* and the pseudo code of Theta* are highlighted in red, namely lines 28-32). The difference between Theta* and A* when they update the g-value and parent of an unexpanded visible neighbor s' of the vertex s that is currently being expanded, is the following (in procedure ComputeCost): If the parent of s is visible from s', then Theta* considers setting the parent of s' to the parent of s [lines 31-32], resulting in a path of length $g(parent(s)) + c(parent(s), s')$ from the start vertex to the parent of s and from there to s' in a straight line (Path 2), which does not constrain the path to grid edges since the parent of a vertex no longer has to be a neighbor of the vertex. Otherwise, it considers setting the parent of s' to s [lines 36-37] (like A*), resulting in a path of length $g(s) + c(s, s')$ from the start vertex to s and from there to s' in a straight line (Path 1). It updates the g-value and parent of s' if the length of the considered path is smaller than the length $g(s')$ of the shortest path from the start vertex to s' that it has found so far. Overall, Theta* considers updating the g-value and parent of s' according to Path 2 if the parent of s is visible from s' (that is, Path 2 is unblocked) since Path 2 is no longer than Path 1 due to the triangle inequality.

Figure 13 shows Theta* in operation on a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is A4, and the goal vertex is C1. The h-values are straight-line distances. Theta* expands start vertex A4, followed by B3 and B2. When Theta* expands B3 with parent A4, B2 is an example of an unexpanded visible neighbor of B3 from which start vertex A4 is not visible. Theta* thus updates B2 according to Path 1 and sets its parent to B3. On the other hand, C3 is an example of an unexpanded visible neighbor of B3 from which start vertex A4 is visible. Theta* thus updates it according to Path 2 and sets its parent to start vertex A4. Theta* terminates when it is about to expand goal vertex C1. Path extraction then retrieves the shortest path [A4, B3, C1] from start vertex A4 to goal vertex C1.

Theta* can find shortest paths (Figure 13). However, this is not guaranteed since the parent of a vertex can only be a visible neighbor of the vertex or the parent of a visible neighbor, which is

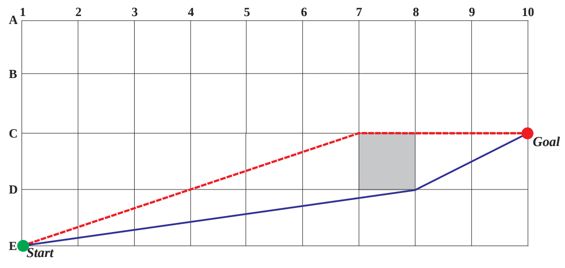


Figure 14: Nonoptimality of Theta* (Daniel *et al.* 2010)

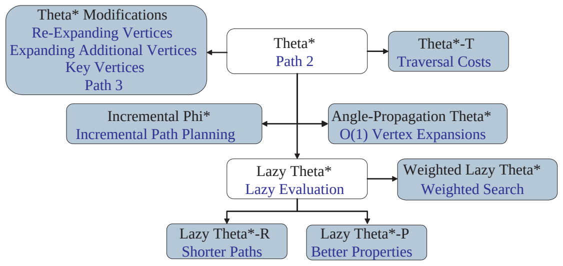
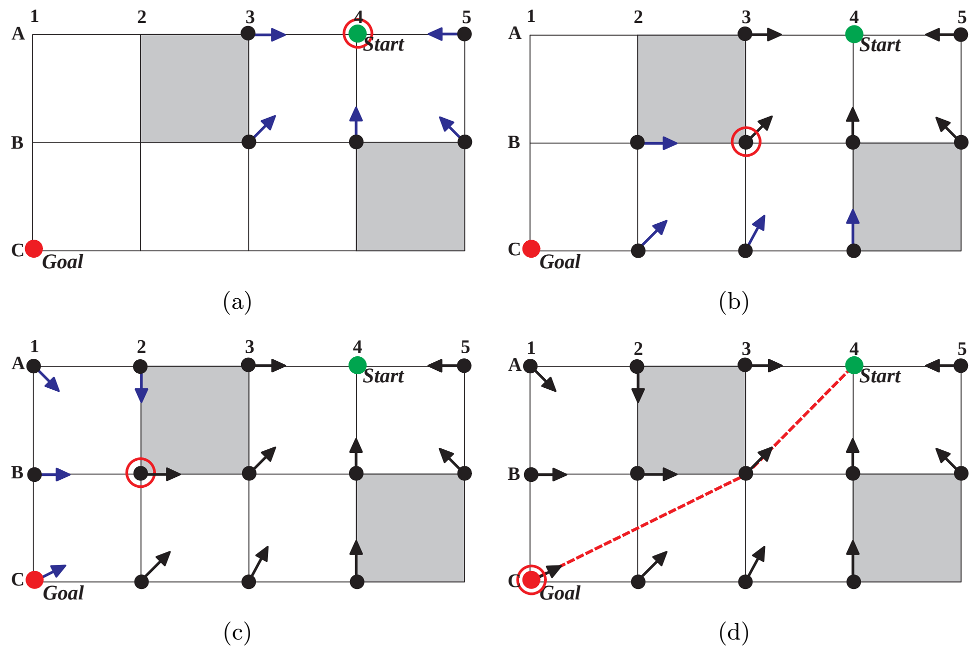


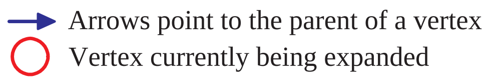
Figure 15: Variants of Theta*

not always the case for shortest paths. Figure 14 shows a path-planning example where the terrain is discretized into a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. Theta* finds shortest paths from start vertex E1 to all possible goal vertices other than C10. The visible neighbors of C10 are B10, B9, C9, D9 and D10. Start vertex E1 is the parent of B10, B9, D9 and D10 since it is visible from these vertices. Either C7 or C8 is the parent of C9, depending on how Theta* breaks ties when Paths 1 and 2 are equally long. It is C7 for the pseudo code of Theta* in Figure 5(b). Therefore, B10, B9, C9, D9, D10, E1 and C7 are the only possible parents of C10. The vertex that minimizes the length of a shortest path from start vertex E1 via the vertex to C10 is C7, resulting in the dashed red path [E1, C7, C10]. This path is longer than the shortest path [E1, D8, C10] but still within 0.2 percent of the length of the shortest path.

There exist several variants of Theta* that result in other trade-offs between the runtimes of their searches and the lengths of the resulting paths (Figure 15) in addition to applying it to grids with different numbers of neighbors (such as to 4-neighbor instead of 8-neighbor square grids). We mention the variants in shaded boxes only briefly:

- Theta* has runtimes per vertex expansion that can be linear in the number of grid cells due to





Arrows point to the parent of a vertex
Vertex currently being expanded

Figure 13: Execution Trace of Theta* (Daniel *et al.* 2010)

the visibility checks. On the other hand, Angle-Propagation Theta* (Nash *et al.* 2007) achieves runtimes per vertex expansion that are, in the worst case, only constant in the number of grid cells by propagating not only g-values and parents but also angle ranges along grid edges. However, Angle-Propagation Theta* currently applies only to 2D 8-neighbor square grids with vertices placed at the corners of grid cells and is more involved, is experimentally not as fast and finds slightly longer paths than Theta*.

- There exist variants of Theta* that find shorter paths (at the expense of being slower), typically by using strategies that A* cannot use.
  - Theta* might be able to find shorter paths by re-expanding vertices or expanding additional vertices. A* expands vertices at most once if the h-values are consistent, while Theta* maintains the closed list to prevent it from expanding vertices multiple times. However, there exist variants of Theta* that do not maintain a closed list and thus can reexpand vertices whose f-values have decreased (Daniel *et al.* 2010). There also exist variants of Theta* that expand more vertices by breaking ties among vertices with the same f-value in the open list in favor of vertices with smaller g-values or by calculating the f-values (like Weighted A* (Pohl 1973)) as $f(s) = g(s) + w \times h(s)$, where (unlike Weighted A*) $0 < w < 1$ (sic!) is a user-given constant, because this typically focuses the search less and increases the number of vertex expansions (Daniel *et al.* 2010).
  - Theta* might also be able to find shorter paths by examining more paths. There exist variants of Theta that consider setting the parent of an unexpanded visible neighbor of the vertex currently being expanded to additional vertices other than the vertex and its parent, such as the parent of its parent (Path 3) or cached vertices encountered earlier during the search (Key Vertices) (Daniel *et al.* 2010).
- There exist variants of Theta* that apply to grids whose grid cells have non-uniform traversal costs (Daniel *et al.* 2010; Choi & Yu 2011), in which case shortest paths on 2D grids with vertices placed at the corners of grid cells can have heading changes at the borders of grid cells with different traversal costs but never at the borders of grid cells with identical traversal costs other than at vertices. In particular, Theta*-T (Daniel *et al.* 2010) (which was so far unnamed)

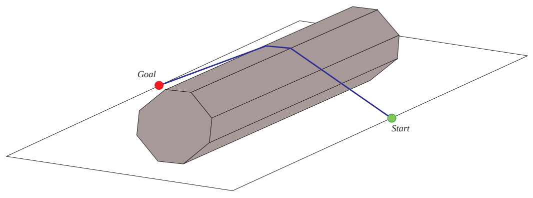
does not produce heading changes at the borders of grid cells other than at vertices. Field D*, on the other hand, applies unchanged to grids whose grid cells have non-uniform traversal costs since it was designed for this case. It can produce heading changes at the borders of grid cells with different traversal costs but also at the borders of grid cells with identical traversal costs.

- Finally, Accelerated A* (Sislak, Volf, & Pechoucek 2009b; 2009a) can be understood as a variant of Theta* with two innovations:
  - First, Accelerated A* uses an adaptive step size to determine the neighbors of a vertex s. When s is far away from blocked grid cells, Accelerated A* chooses vertices as neighbors of s that are further away from s than when s is close to blocked grid cells. On 2D square grids with vertices placed at the corners of grid cells, it uses a maximum unblocked square to determine the neighbors of s, which it constructs by expanding a square centered on s until one side of the square touches either the goal vertex or a blocked grid cell. It then chooses the neighbors of vertex s from the vertices on the sides of the maximum unblocked square, such as the four vertices that are in the middle of the four sides.
  - Second, Accelerated A* basically considers setting the parent of an unexpanded visible neighbor of the vertex that is currently being expanded to additional vertices other than the vertex and its parent, namely all expanded vertices, and uses a sufficiently large ellipse to prune those expanded vertices that cannot possibly be chosen as the parent. Accelerated A* considers setting the parent of an unexpanded visible neighbor of the vertex that is currently being expanded to all expanded vertices within this ellipse.

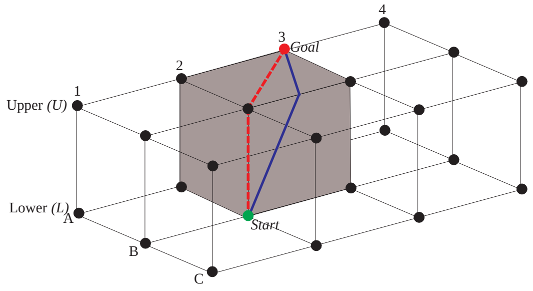
Experimental Comparisons

Yap et al. (Yap *et al.* 2011b) compared Block A* whose lookup table stores the (lengths of the) shortest paths, A* on Grid Graphs and Theta* on known 2D game maps and known 2D square grids with randomly blocked grid cells, see also (Yap *et al.* 2011a). A* on Grid Graphs and Theta* were approximately 2.6 and 7.5 times slower than Block A*, respectively. The paths of A* on Grid Graphs and Theta* were approximately 4.2 percent longer and less than one percent shorter than those of Block A*, respectively.

Nash et al. compared A* on Grid Graphs, Theta*, Field D*, A* with Post Smoothing and A* on Visibility Graphs on known 2D game maps and known 2D square grids with randomly blocked grid cells (Nash 2012), see also (Daniel *et al.* 2010). Theta*, Field D*, A* with Post Smoothing and A* on Visibility Graphs were approximately 3.2, 8.6, 10.6 and 118.6 times slower than A* on Grid Graphs, respectively. The paths of Theta*, Field D*, A* with Post Smoothing and A* on Visibility Graphs were approximately 4.6, 4.4, 3.9 and 4.7 percent shorter than those of A* on Grid Graphs, respectively.



(a)



(b)

Figure 16: Nonoptimality of A* on Visibility Graphs in 3D Terrain (a) and 3D Grids (b) (Nash, Koenig, & Tovey 2010)

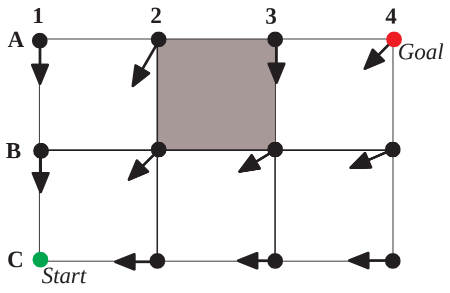
Sislak et al. compared Theta*, Accelerated A* and A* on Visibility Graphs on known 2D square grids with randomly blocked grid cells and arranged blocked grid cells to simulate path-planning problems from robotics (Sislak, Volf, & Pechoucek 2009b). Accelerated A* and A* on Visibility Graphs were approximately 1.7 and 1630.0 times slower than Theta*, respectively. The paths of Accelerated A* and A* on Visibility Graphs were approximately 1.0 percent shorter than those of Theta*. Accelerated A* always found shortest paths although no theoretical argument was made that it is optimal.

Known 3D Terrain

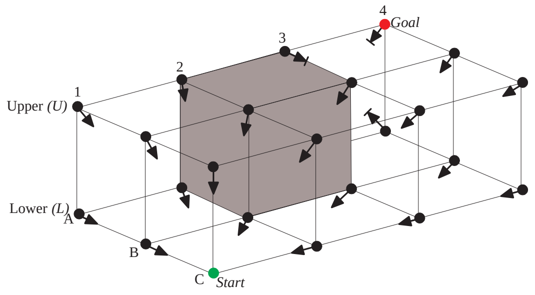
Agents operate not only in known 2D terrain but also in known 3D terrain, such as in the video game James Cameron's Avatar: The Game by Ubisoft Montreal. Path planning in known 3D terrain can be more difficult than in known 2D terrain. For example, A* on Grid Graphs can find grid paths that are at most approximately eight percent longer than shortest paths on 2D 8-neighbor square grids with vertices placed at the corners of grid cells rather than at least approximately 13 percent longer than shortest paths on 3D 26-neighbor cubic grids with vertices placed at the corners of grid cells, as explained under "Path-Length Analysis." A* on Visibility Graphs finds shortest paths in 2D terrain with polygonal obstacles but is not guaranteed to find shortest paths in 3D terrain with polyhedral obstacles (Choset *et al.* 2005), as shown in Figure 16(a) where the heading changes of the only shortest path from the start vertex to the goal vertex are not at the corners of the polyhedral obstacle. Figure 16(b) demonstrates that this property also holds for 3D grids by showing a path-planning example where the terrain is discretized into a 3D 26-neighbor cubic grid with vertices placed at the corners of grid cells. The start vertex is B2L, and the goal vertex is A3U. The dashed red path [B2L, B2U, A3U] is the shortest vertex path, and the solid blue path is a shortest path. Thus, it is neither guaranteed that a shortest path from the start vertex to the goal vertex is part of the visibility graph nor that it is a vertex path on 3D grids (even with vertices placed at the corners of grid cells). Shortest vertex paths can thus be longer than shortest paths on 3D grids. In fact, shortest paths in 2D terrain with polygonal obstacles can be found in polynomial time, while finding shortest paths in 3D terrain with polyhedral obstacles is NP-hard (Canny & Reif 1987). One can modify Field D* so that it applies to 3D cubic grids and 3D tetrahedral meshes rather than 2D square grids and 2D triangular meshes by changing the optimization equations (Carsten, Ferguson, & Stentz 2006; Sapronov & Lacaze 2008; Perkins *et al.* 2012). The resulting variants of Field D* are more involved than Field D* and typically use additional approximations. For example, the optimization equations for 2D square grids (where the perimeter consists of edges) can be solved in closed form but the ones for 3D cubic grids (where the perimeter consists of faces) cannot. Block A* could be extended without problems to all types of 3D grids although its lookup table can consume much more memory for 3D grids than 2D grids.

Theta* applies unchanged to every graph embedded in 2D or 3D terrain. However, it performs one visibility check per generated vertex (namely, one visibility check for every unexpanded visible neighbor of the vertex that is currently being expanded). The number of visibility checks thus increases with the number of neighbors (even if the pseudo code of Theta* in Figure 5(b) is optimized to perform a visibility check only if the length $g(parent(s)) + c(parent(s), s')$ of Path 2 is smaller than the length $g(s)$ of the shortest path from the start vertex to s' that Theta* has found so far). Figure 17(a) shows a path-planning example where the terrain is discretized into a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is C1, and the goal vertex is A4. Theta* performs 3+6+6=15 visibility checks on line 28. (The visible neighbors on line 11 can be determined without visibility checks.) On the other hand, Figure 17(b) shows a similar path-planning example where the terrain is discretized into a 3D 26-neighbor cubic grid with vertices placed at the corners of grid cells. The start vertex is C1L, and the goal vertex is A4U. Theta* now performs many more than 15 visibility checks, namely 7+15+15=37 visibility checks. The runtimes per vertex expansion of Theta* can be linear in the number of grid cells due to the visibility checks, even though visibility checks on 2D square grids and 3D cubic grids can be performed with fast line-drawing algorithms from computer graphics (Daniel *et al.* 2010), such as the standard Bresenham line-drawing algorithm (Bresenham 1965), and be optimized further for the task (Choi, Lee, & Yu 2010). Visibility checks on other types of graphs, such as navigation meshes, can be slower. It is thus important to decrease the number of visibility checks per vertex expansion in 3D terrain.

Lazy Theta* (Nash, Koenig, & Tovey 2010) is a variant of Theta* that can speed up Theta* both when it generates many more vertices than it expands and when its visibility checks are slow. It uses lazy evaluation to perform only one visibility check per expanded vertex instead of one visibility check per generated vertex but increases the number of vertex expansions and potentially the length of the resulting path. There exist several variants of Lazy Theta*. For example, the main variant of Lazy Theta* delays visibility checks by optimistically assuming that the parent of the vertex that is currently being expanded is visible from every unexpanded visible neighbor of the expanded vertex, while Lazy Theta*-P (Nash, Koenig, & Tovey 2010) delays visibility checks by pessimistically assuming that the parent is not visible. We describe the main variant. Due to its optimistic assumption, Lazy Theta* can update the g-value and parent of an unexpanded visible neighbor of the expanded vertex according to Path 2 even if the parent of the expanded vertex is not visible from the neighbor. It revisits this assumption when it expands the neighbor and, if it does not hold, corrects the g-value and parent of the neighbor. Figure 5(c)



(a) 3+6+6=15 Visibility Checks



(b) 7+15+15=37 Visibility Checks

Figure 17: Visibility Checks of Theta* in 2D Terrain (a) and 3D Terrain (b)

shows the pseudo code of Lazy Theta* (the differences between the pseudo code of Theta* and the pseudo code of Lazy Theta* are highlighted in red, namely line 8 and lines 34-39). The difference between Lazy Theta* and Theta* when they update the g-value and parent of an unexpanded visible neighbor s' of the vertex s that is currently being expanded, is the following (in procedure ComputeCost): Without checking whether the parent of s is visible from s', Lazy Theta* considers setting the parent of s' to the parent of s [lines 31-32], resulting in a (potentially blocked) path of length $g(parent(s)) + c(parent(s), s')$ from the start vertex to the parent of s and from there to s' in a straight line (Path 2). It updates the g-value and parent of s' if the length of this path is smaller than the length $g(s')$ of the shortest path from the start vertex to s' that it has found so far. Lazy Theta* performs one visibility check (in procedure SetVertex) immediately before it expands vertex s'. If the new parent of s' is visible from s', then Lazy Theta* does not change the g-value and parent of s'. Otherwise, Lazy Theta* updates the g-value and parent of s' according to Path 1 by setting the parent of s' to the expanded visible neighbor s'' of s' that minimizes the length $g(s'') + c(s'', s')$ of the path from the start vertex to s'' and from there to s' in a straight line [lines 37-38]. (This path is well-defined and of finite length since s'' is an expanded visible neighbor of s'.) Lazy Theta*-R (Nash, Koenig, & Tovey 2010) is a variant of Lazy Theta* that, at this point, re-inserts s' into the open list with an updated key instead of expanding it. This gives Lazy Theta*-R an opportunity to discover shorter paths from the start vertex to s' before it expands s'.

Lazy Theta* applies to all graphs that Theta* applies to. For example, it applies not only to 3D but also to 2D terrain. We explain Lazy Theta* on 2D 8-neighbor square grids because they are easier to visualize than 3D 26-neighbor cubic grids. Therefore, Figure 18 shows Lazy Theta* in operation on a 2D 8-neighbor square grid with vertices placed at the corners of grid cells. The start vertex is A4, and the goal vertex is C1. The h-values are the straight-line distances. Lazy Theta* expands start vertex A4, followed by B3 and B2. When Lazy Theta* expands B3 with parent A4, B2 is an example of an unexpanded visible neighbor of B3. Lazy Theta* optimistically assumes that start vertex A4 is visible from B2 and sets the parent of B2 to start vertex A4 (Figure 18(c)). Lazy Theta* expands B2 next. Since start vertex A4 is not visible from B2, Lazy Theta* updates the g-value and parent of B2 according to Path 1 by considering the paths from the start vertex A4 to each expanded visible neighbor of B2 and from there to B2 in a straight line. Lazy Theta* sets the parent of B2 to B3 since the path from start vertex A4 to B3 and from there to B2 in a straight line is the only such path and thus it is also the shortest such path (Figure 18(d)). Lazy Theta* terminates when it is about to expand goal vertex C1 after it has checked that the parent of goal vertex C1, namely B3, is indeed visible from goal vertex C1. Path extraction then retrieves the shortest path [A4, B3, C1] from start vertex A4 to goal vertex C1.

Lazy Theta* can find the same paths as Theta* (Figure 18). In the execution trace depicted in Figure 18, Lazy Theta* performs only four visibility checks, while Theta* performs 5+6+6=17 visibility checks. However, this is not guaranteed. Lazy Theta* typically finds slightly longer paths than Theta* but performs many fewer visibility checks and is thus faster. One can typically decrease the number of visibility checks even more, using a strategy that A* can use for the same purpose, namely weighting the h-values. Weighted Lazy Theta* (Nash 2012) calculates the f-values (like Weighted A* (Pohl 1973)) as $f(s) = g(s) + w \times h(s)$, where

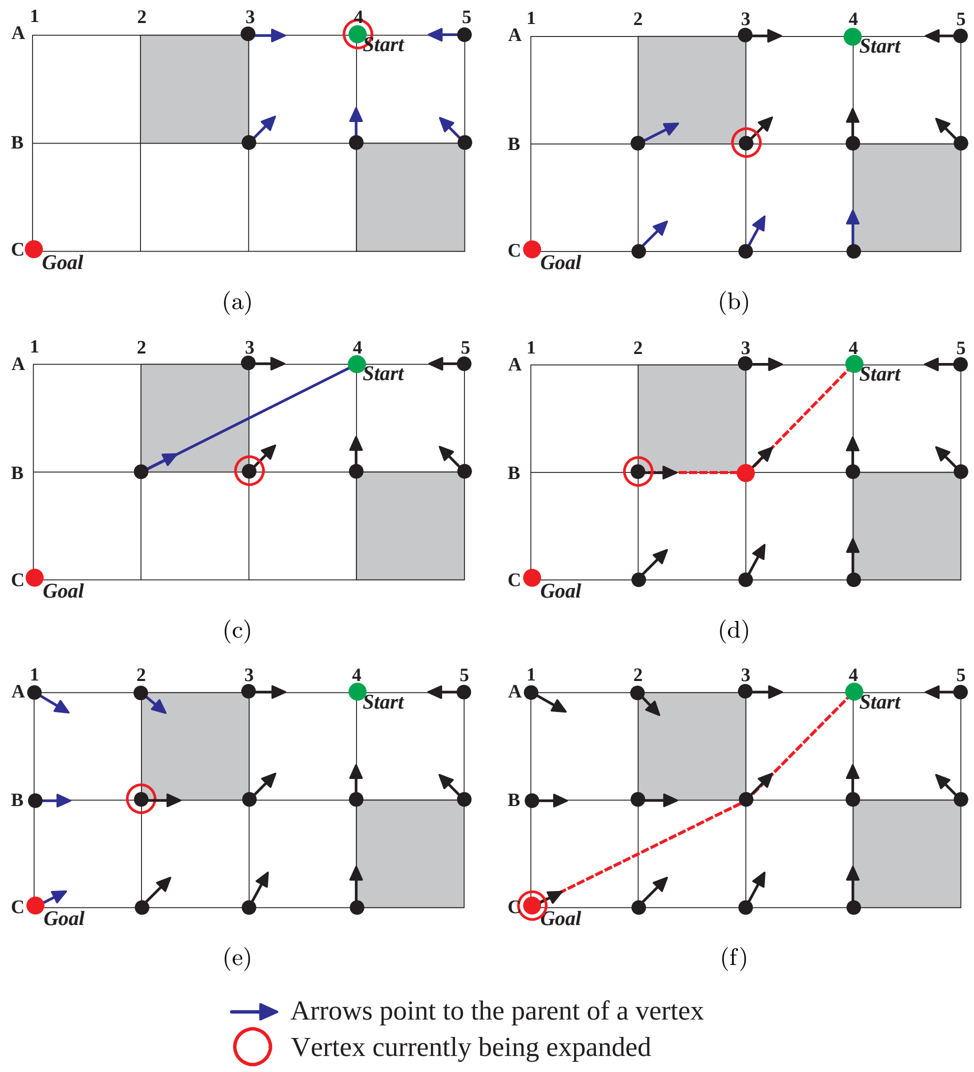

1
2
3
4
5
A
B
C
Start
Goal
(a)
(b)
(c)
(d)
(e)
(f)
Arrows point to the parent of a vertex
Vertex currently being expanded

Figure 18: Execution Trace of Lazy Theta*

$w > 1$ is a user-given constant, because this typically focuses the search better and decreases the number of vertex expansions. Therefore, it also decreases the number of visibility checks. Both reductions make it faster. On the other hand, the path length increases, just as it does for A*, although not necessarily as much as it does for A*, for the following reason: Lazy Theta* can set the parent of vertex s to vertex s' according to Path 2 only if there exists a grid path of expanded vertices from s' to s such that s' is the parent of every vertex on the grid path (except for s' itself). Even if Lazy Theta* expands few vertices, it can still set the parent of s to s' as long as there still exists a grid path with these properties, in which case the path length does not increase.

Experimental Comparisons

Nash et al. (Nash 2012) compared A* on Grid Graphs, Lazy Theta*, Theta* and A* with Post Smoothing on known 3D cubic grids with randomly blocked grid cells, see also (Nash, Koenig, & Tovey 2010). Lazy Theta*, Theta* and A* with Post Smoothing were approximately 4.0, 6.7 and 46.5 times slower than A* on Grid Graphs, respectively. The paths of Lazy Theta*, Theta* and A* with Post Smoothing were approximately 7.1, 7.2 and 5.7 percent shorter than those of A* on Grid Graphs, respectively.

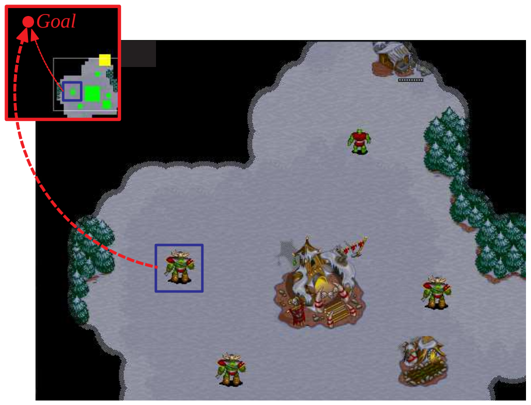


Figure 19: Screenshot of Warcraft II: Tides of Darkness by Blizzard Entertainment and Climax Studios

Unknown 2D Terrain

Agents operate not only in 2D terrain with grid cells of known blockage status but also in 2D terrain with grid cells of unknown blockage status, for example, because the blockage status of grid cells is initially unknown (in unknown terrain) or changes over time (in dynamic terrain). One way of navigating in initially unknown terrain is to interleave path planning with movement, which requires agents to find paths repeatedly. Consider, for example, an agent that has to move from its current vertex to a given goal vertex in initially unknown terrain, such as a game character that has to move to coordinates specified by a user despite the "fog of war" (that is, blacked-out areas) (Figure 19). The agent initially does not know which grid cells are blocked but always observes the blockage status of grid cells within its sensor radius and adds them to its map. The agent can use goal-directed navigation with the freespace assumption (Koenig, Smirnov, & Tovey 2003) to reach the goal vertex or determine that this is impossible: It finds a short path from its current vertex to the goal vertex, taking into account its current knowledge of the blockage status of grid cells and making the freespace assumption (that is, optimistically assuming that grid cells with unknown blockage status are unblocked). If no such path exists, it stops unsuccessfully. Otherwise, it follows the path until it either reaches the goal vertex, in which case it stops successfully, or observes the path to be blocked, in which case it repeats the procedure, taking into account its revised knowledge of the blockage status of grid cells and still optimistically assuming that grid cells with unknown blockage status are unblocked. Therefore, the agent has to find a new short path every time it observes its current path to be blocked.

Figure 20 shows goal-directed navigation with the freespace assumption in operation on a 2D 8-neighbor square grid with vertices placed at the corners of grid cells, using A* on Grid Graphs rather than any-angle path planning. The agent always observes the blockage status of all grid cells that have its current vertex as a corner. The start vertex is C2, and the goal vertex is D6. The agent starts at C2 and finds a shortest grid path from its current vertex C2 to goal vertex D6 assuming that all grid cells are unblocked. It follows the path to C3, where it observes two grid cells that block its path. It finds a shortest grid path from its current vertex C3 to goal vertex D6 taking the two blocked grid cells into account. It then follows the path via D3 to D4, where it observes another two grid cells that block its path, and repeats the procedure.

The agent thus has to solve a series of similar path-planning problems. It has to solve them quickly so that it can move without stopping. Incremental path-planning algorithms (Koenig *et al.* 2004) solve a series of similar path-planning problems quickly by reusing information from previous searches to speed up their current search, which typically makes them faster than repeated A* searches from scratch. The main difference between this approach and most other replanning and plan-reuse algorithms (such as planning by analogy) is that incremental path-planning algorithms are guaranteed to find paths that are no longer than those found by repeated A* searches from scratch.

Some any-angle path-planning algorithms, including Field D* and Theta*, can use incremental path-planning techniques to replan faster than repeated searches from scratch. Field D* was designed for this case by extending the incremental heuristic path-planning algorithm D* Lite (Koenig & Likhachev 2005) or D* (Stentz 1995). On the other hand, Theta* cannot easily extend D* Lite because the parent of a vertex is not guaranteed to be its neighbor and its f-values are not guaranteed to be non-decreasing. Incremental Phi* (Nash, Koenig, & Likhachev 2009) is an incremental variant of Theta* that can currently handle only the case where the blockage status of grid cells is initially unknown (which is equivalent to the case where the costs of grid edges can increase to infinity), while Field D* can also handle the case where the blockage status of grid cells changes over time (or, more generally, the case where the costs of grid edges can increase and decrease by arbitrary amounts). Incremental Phi* applies only to 2D 8-neighbor square grids with vertices placed at the corners of grid cells, while Field D* can be modi-

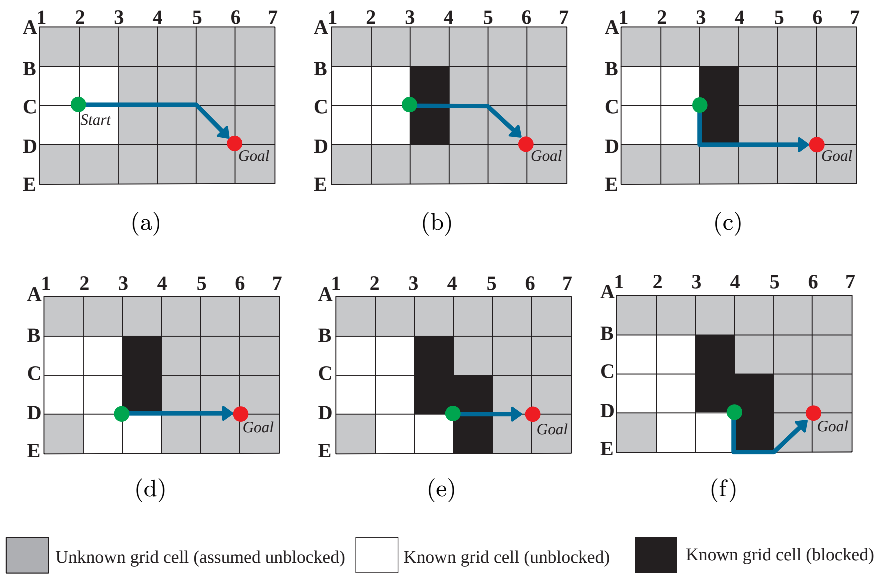

A
B
C
D
E
1
2
3
4
5
6
7
Start
Goal
(a)
(b)
(c)
(d)
(e)
(f)
Unknown grid cell (assumed unblocked)
Known grid cell (unblocked)
Known grid cell (blocked)

Figure 20: Execution Trace of Goal-Directed Navigation with the Freespace Assumption

fied so that it applies to additional types of graphs embedded in 2D or 3D terrain, as explained under "Known 2D Terrain" and "Known 3D Terrain." However, neither Incremental Phi* nor Field D* apply to every graph embedded in 2D or 3D terrain.

Experimental Comparisons

Ferguson et al. (Ferguson & Stentz 2006) compared D* Lite and Field D* on unknown 2D grids with randomly assigned non-uniform traversal costs, see also (Ferguson 2006). Field D* was approximately 1.7 times slower than D* Lite. Its paths were approximately four percent less costly than those of Field D*.

Nash et al. (Nash, Koenig, & Likhachev 2009) compared Incremental Phi* and repeated Theta* searches on unknown 2D game maps and unknown 2D square grids with randomly blocked grid cells, see also (Nash 2012). Repeated Theta* searches were approximately 6.0 times slower than Incremental Phi*. Their paths were less than one percent shorter than those of Incremental Phi*.

Conclusions

We provided a sketch of an analysis of how much longer shortest grid paths can be than shortest paths. The results suggested that it might be necessary to find shorter paths than shortest grid paths. Any-angle path-planning algorithms are variants of A* that find short paths in (continuous) terrain by propagating information along grid edges (like A* on Grid Graphs, to be fast) without constraining the resulting paths to grid edges (like A* on Visibility Graphs, to find short paths). We surveyed the state-of-the-art in any-angle path-planning algorithms, including variants of Block A*, Field D* and Theta* in known 2D terrain, known 3D terrain and unknown 2D terrain. Future research should be dedicated to understanding the full power of any-angle path-planning, to broaden it from a few isolated path-planning algorithms to a well-understood framework, to extend its applicability (for example, to motion planning) and to understand its properties better, including the influence of design decisions on the trade-off with respect to its memory consumption, the runtimes of its searches and the lengths of the resulting paths as well as the guarantees it is able to provide. For example, no tight bounds are known on the ratio of the lengths of the paths found by specific any-angle path-planning algorithms and shortest paths. Analyses of any-angle path-planning algorithms are complicated by the fact that even some of the basic properties of A* do not hold for any-angle path-planning algorithms. For example, A* has the property that the f-values of all vertices along all branches of its search trees are non-decreasing if the h-values are consistent. Theta* does not have this property. Overall, any-angle path planning appears to be a promising way of trading off between the runtimes of the searches and the lengths of the resulting paths in robotics and video games.

With respect to efficiency, any-angle path-planning algorithms are typically faster than A* on Visibility Graphs and find shorter paths than A* on Grid Graphs. With respect to simplicity, any-angle path-planning algorithms are typically simple to understand, implement, debug and extend since they extend A*, which has these properties. For example, Theta* and Lazy Theta* are similar to A* and can easily be taught to game developers and undergraduate students, see the tutorials and class project listed under "Recent Resources." Finally, with respect to generality, some any-angle path-planning algorithms, such as Theta*, apply to every graph embedded in 2D or 3D terrain.

Recent Resources

- **Dissertations** (Ferguson 2006), (Nash 2012).
- **Accelerated A* Publications** (Sislak, Volf, & Pechoucek 2009a), (Sislak, Volf, & Pechoucek 2009b).
- **Anya Publications** (Harabor & Grastien 2013)
- **Block A* Publications** (Yap *et al.* 2011b), (Yap *et al.* 2011a).
- **Field D* Publications** (Carsten, Ferguson, & Stentz 2006), (Ferguson & Stentz 2006), (Sapronov & Lacaze 2008), (Carsten *et al.* 2009).
- **Theta* Publications** (Nash *et al.* 2007), (Koenig, Daniel, & Nash 2008), (Nash, Koenig, & Likhachev 2009), (Daniel *et al.* 2010), (Nash, Koenig, & Tovey 2010), (Choi & Yu 2011), (Choi & Yu 2011).
- **Web Pages**
  - idm-lab.org/project-o.html (information on Theta* and its variants)
  - aigamedev.com/open/tutorials/theta-star-any-angle-paths/ (on-line tutorial on Theta*)
  - aigamedev.com/open/tutorial/lazy-theta-star/ (on-line tutorial on Lazy Theta*)
- **Class Project** The following class project has successfully been used at the University of Nevada at Reno, New Mexico State University and the University of Southern California and was chosen as a "Model Artificial Intelligence Assignment" by the Symposium on Educational Advances in Artificial Intelligence 2010 (Koenig, Daniel, & Nash 2008):

  http://idm-lab.org/project-m/project2.html

  This stand-alone 14-page path-planning project for an undergraduate or graduate artificial intelligence class is part of an effort to use video games as a motivator in projects without the students having to use game engines. In this project, the students code A* and then extend it to Theta* to find paths for game characters in known grid worlds. The students have to develop an understanding of A* to answer questions that are not yet covered in textbooks. The project lists 18 possible project choices, both easy and difficult ones, that cover theoretical and implementation aspects of heuristic search.

Acknowledgments

Our overview article is based on a JAIR article (Daniel *et al.* 2010), several conference papers (Nash *et al.* 2007; Nash, Koenig, & Likhachev 2009; Nash, Koenig, & Tovey 2010) and Sven Koenig's part of the Tutorial on Search-Based Planning: Toward High Dimensionality and Differential Constraints at the AAAI Conference on Artificial Intelligence 2012 (with M. Pivtoraiko and M. Likhachev). We re-use small parts of (Daniel *et al.* 2010) and (Nash, Koenig, & Tovey 2010) verbatim. We thank K. Daniel, A. Felner, M. Likhachev, S. Sun, C. Tovey, W. Yeoh and X. Zheng for their contributions to joint research that resulted in the publications that form the basis of our overview article. We thank R. Holte, J. Schaeffer and P. Yap for their input on our research, N. Sturtevant for providing the video game maps at movingai.com that we used as test cases and both D. Ferguson and T. Uras and the reviewers for comments on our draft manuscript. Our research has been supported by NSF under grant number IIS-1319966, ARO under grant number W911NF-08-1-0468, ONR under grant number N00014-09-1-1031 and Northrop Grumman via a fellowship to Alex Nash. 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, companies or the U.S. government.

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*, 1511–1512.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *Journal of Game Development* 1(1):7–28.

Bresenham, J. 1965. Algorithm for computer control of a digital plotter. *IBM Systems Journal* 4(1):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*, 49–60.

Carsten, J.; Rankin, A.; Ferguson, D.; and Stentz, A. 2009. Global planning on the Mars exploration rovers: Software integration and surface testing. *Journal of Field Robotics* 26(4):337–357.

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*, 3381–3386.

Champandard, A. 2010. Personal Communication.

Choi, S., and Yu, W. 2011. Any-angle path planning on non-uniform costmaps. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 5615–5621.

Choi, S.; Lee, J.-Y.; and Yu, W. 2010. Fast any-angle path planning on grid maps with non-collision pruning. In *Proceedings of the IEEE International Conference on Robotics and Biomimetics*, 1051–1056.

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.

Chrpa, L., and Komenda, A. 2011. Smoothed hexgrid trajectory planning using helicopter dynamics. In *Proceedings of the International Conference on Agents and Artificial Intelligence*, 629–632.

Daniel, K.; Nash, A.; Koenig, S.; and Felner, A. 2010. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research* 39:533–579.

Ferguson, D., and Stentz, A. 2006. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics* 23(2):79–101.

Ferguson, D. 2006. *Single Agent and Multi Agent Path Planning in Unknown and Dynamic Environments*. Ph.D. Dissertation, Carnegie Mellon University.

Ferguson, D. 2013. Personal Communication.

Harabor, D., and Grastien, A. 2013. An optimal any-angle pathfinding algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 308–311.

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(2):100–107.

Hershberger, J., and Suri, S. 1999. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing* 28(6):2215–2256.

Kavraki, L.; Svestka, P.; Latombe, J.; and Overmars, M. 1996. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation* 12(4):566–580.

Koenig, S., and Likhachev, M. 2005. Fast replanning for navigation in unknown terrain. *Transactions on Robotics* 21(3):354–363.

Koenig, S.; Likhachev, M.; Liu, Y.; and Furcy, D. 2004. Incremental heuristic search in artificial intelligence. *Artificial Intelligence Magazine* 25(2):99–112.

Koenig, S.; Daniel, K.; and Nash, A. 2008. A project on any-angle path planning for computer games for introduction to artificial intelligence classes. Technical report, University of Southern California. http://idm-lab.org/project-m/project2.html.

Koenig, S.; Smirnov, Y.; and Tovey, C. 2003. Performance bounds for planning in unknown terrain. *Artificial Intelligence Journal* 147(1–2):253–279.

LaValle, S., and Kuffner, J. 2001. Rapidly-exploring random trees: Progress and prospects. In Donald, B.; Lynch, K.; and Rus, D., eds., *Algorithmic and Computational Robotics: New Directions*. A K Peters. 293–308.

LaValle, S. 2006. *Planning Algorithms*. Cambridge University Press.

Lee, D.-T. 1978. *Proximity and Reachability in the Plane*. Ph.D. Dissertation, University of Illinois at Urbana-Champaign.

Lindemann, S., and LaValle, S. 2004. Steps toward derandomizing RRTs. In *Proceedings of the International Workshop on Robot Motion and Control*, 271–277.

Liu, Y.-H., and Arimoto, S. 1992. Path planning using a tangent graph for mobile robots among polygonal and curved obstacles. *International Journal of Robotics Research* 11(4):376–382.

Lozano-Pérez, T., and Wesley, M. 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM* 22(10):560–570.

Millington, I., and Funge, J. 2009. *Artificial Intelligence for Games*. Morgan Kaufmann, second edition.

Mitchell, J., and Papadimitriou, C. 1991. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of the ACM* 38(1):18–73.

Mitchell, J.; Mount, D.; and Papadimitriou, C. 1987. The discrete geodesic problem. *SIAM Journal on Computing* 16(4):647–668.

Nagy, B. 2003. Shortest paths in triangular grids with neighbourhood sequences. *Journal of Computing and Information Technology* 11(2):111–122.

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*, 1177–1183.

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*, 1824–1830.

Nash, A.; Koenig, S.; and Tovey, C. 2010. Lazy Theta*: Any-angle path planning and path length analysis in 3D. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Nash, A. 2012. *Any-Angle Path Planning*. Ph.D. Dissertation, University of Southern California. http://idm-lab.org/project-o.html.

Pearl, J. 1985. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Perkins, S.; Marais, P.; Gain, J.; and Berman, M. 2012. Field D* path-finding on weighted triangulated and tetrahedral meshes. *Autonomous Agents and Multi-Agent Systems* 1–35.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 12–17.

Rabin, S. 2000. A* speed optimizations. In Deloura, M., ed., *Game Programming Gems*. Charles River Media. 272–287.

Sapronov, L., and Lacaze, A. 2008. Path planning for robotic vehicles using Generalized Field D*. In *Proceedings of the SPIE: Unmanned Systems Technology X*, volume 6962, 69621C–1–69621C–12.

Sislak, D.; Volf, P.; and Pechoucek, M. 2009a. Accelerated A* path planning. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 1133–1134.

Sislak, D.; Volf, P.; and Pechoucek, M. 2009b. Accelerated A* trajectory planning: Grid-based path planning comparison. In *Proceedings of the Workshop on Planning and Plan Execution for Real-World Systems at the International Conference on Automated Planning and Scheduling*, 74–81.

Stentz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1652–1659.

Thorpe, C. 1984. Path relaxation: Path planning for a mobile robot. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 318–321.

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

Tozour, P. 2008. Fixing pathfinding once and for all. www.ai-blog.net/archives/000152.html.

Yap, P.; Burch, N.; Holte, R.; and Schaeffer, J. 2011a. Any-angle path planning for computer games. In *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Yap, P.; Burch, N.; Holte, R.; and Schaeffer, J. 2011b. Block A*: Database-driven search with applications in any-angle path-planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Yap, P. 2002. Grid-based path-finding. In *Proceedings of the Canadian Conference on Artificial Intelligence*, 44–55.