University of Pennsylvania
ScholarlyCommons

Lab Papers (GRASP)

General Robotics, Automation, Sensing and Perception Laboratory

7-11-2009

Incremental Phi*: Incremental Any-Angle Path Planning on Grids

Alex Nash
University of Southern California

Sven Koenig
University of Southern California

Maxim Likhachev
University of Pennsylvania, maximl@seas.upenn.edu

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

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/grasp_papers/21
For more information, please contact repository@pobox.upenn.edu.

Incremental Phi*: Incremental Any-Angle Path Planning on Grids

Abstract
We study path planning on grids with blocked and unblocked cells. Any-angle path-planning algorithms find short paths fast because they propagate information along grid edges without constraining the resulting paths to grid edges. Incremental path-planning algorithms solve a series of similar path-planning problems faster than repeated single-shot searches because they reuse information from the previous search to speed up the next one. In this paper, we combine these ideas by making the any-angle path-planning algorithm Basic Theta* incremental. This is non-trivial because Basic Theta* does not fit the standard assumption that the parent of a vertex in the search tree must also be its neighbor. We present Incremental Phi* and show experimentally that it can speed up Basic Theta* by about one order of magnitude for path planning with the freespace assumption.

Comments
A. Nash, S. Koenig and M. Likhachev. Incremental Phi*: Incremental Any-Angle Path Planning on Grids. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pages (in print), 2009.

This conference paper is available at ScholarlyCommons: http://repository.upenn.edu/grasp_papers/21

Incremental Phi*: Incremental Any-Angle Path Planning on Grids*

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

Maxim Likhachev
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
maximl@seas.upenn.edu

Abstract

We study path planning on grids with blocked and unblocked cells. Any-angle path-planning algorithms find short paths fast because they propagate information along grid edges without constraining the resulting paths to grid edges. Incremental path-planning algorithms solve a series of similar path-planning problems faster than repeated single-shot searches because they reuse information from the previous search to speed up the next one. In this paper, we combine these ideas by making the any-angle path-planning algorithm Basic Theta* incremental. This is non-trivial because Basic Theta* does not fit the standard assumption that the parent of a vertex in the search tree must also be its neighbor. We present Incremental Phi* and show experimentally that it can speed up Basic Theta* by about one order of magnitude for path planning with the freespace assumption.

1 Introduction

We study path planning where a two-dimensional terrain is discretized into square cells that are either blocked (grey) or unblocked (white) and each corner of a cell represents a grid vertex on an eight-neighbor grid [Choset *et al.*, 2005]. A robot can use **path planning with the freespace assumption** to move from a given start vertex to a given goal vertex without knowing the blockage status of all cells a priori [Koenig *et al.*, 2003]. It finds a short path from its current vertex to the goal vertex given its current knowledge of the blockage status of the cells in the grid. The robot then repeatedly moves one unit along this path, observes the blockage status of cells within its sensor radius, finds a new short path from its current vertex to the goal vertex given its revised knowledge of the blockage status of the cells in the grid and then repeats this process. Thus, the robot must at least find a new short path each time it observes its existing path to be blocked, which can be slow on large grids. Incremental path-planning algorithms based on A* [Hart *et al.*, 1968], such as D* [Stentz, 1995] and D* Lite [Koenig and Likhachev, 2002], solve a series of similar path-planning problems faster than repeated single-shot A* searches because they reuse information from the previous search to speed up the next one. However, they constrain the resulting paths to grid edges (just like A*), see Figure 1 (left). Any-angle path-planning algorithms based on A* find much shorter paths because they propagate information along grid edges without constraining the resulting path to grid edges. For example, the any-angle path in Figure 1 (right) avoids the unnecessary heading change at C2 in Figure 1 (left). It therefore makes sense to study combinations of incremental and any-angle path-planning algorithms. Field D* [Ferguson and Stentz, 2006] is one such combination, but it suffers from unnecessary heading changes as a result of linear interpolation. Basic Theta* [Nash *et al.*, 2007] is an any-angle path-planning algorithm that finds shorter paths than Field D* for our path-planning problems, but cannot easily be made incremental because it does not fit a standard assumption that holds for A*, namely that the parent of a vertex in the search tree must also be its neighbor. We present Incremental Phi*, an incremental version of Basic Theta*, and show experimentally that it can speed up Basic Theta* by about one order of magnitude for path planning with the freespace assumption.

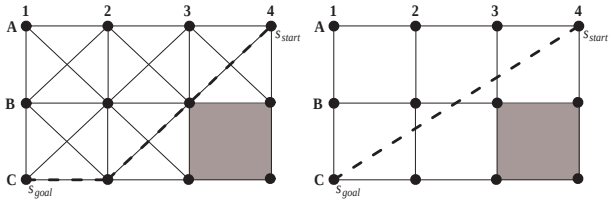

Figure 1: Grid Edge Path (left) vs. Any-Angle Path (right)

2 Notation and Definitions

We assume an eight-neighbor grid, where S is the set of all grid vertices, $s_{start} \in S$ is the start vertex of the search and $s_{goal} \in S$ 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 its length. *lineofsight*(s, s') is true iff vertex s has line-of-sight to vertex s', that is, the line segment $\mathcal{L}(s, s')$ neither traverses blocked cells nor passes between blocked cells that share a grid edge. $\mathcal{L}(s, s')$ traverses a cell if it crosses into its interior. For sim-

*We thank Vadim Bulitko from the University of Alberta for making maps from Baldur's Gate II available to us and Willow Garage for making its robotics maps available to us. Alex Nash was supported by the Northrop Grumman Corporation. This material is also based upon work supported by, or in part by, the U.S. Army Research Laboratory and the U.S. Army Research Office under contract/grant number W911NF-08-1-0468 and by NSF under contract 0413196. 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.

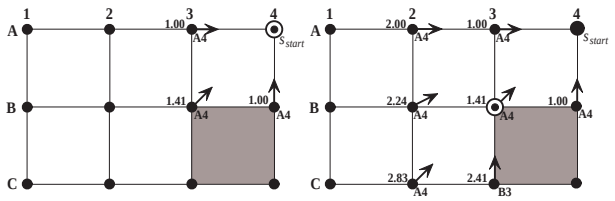

Figure 2: Example Trace of Basic Theta*

plicity, we allow the line segment $\mathcal{L}(s, s')$ to pass between diagonally touching blocked cells. $nghbr_8(s) \subseteq S$ is the set of eight neighbors of vertex s. $nghbr_{vis}(s) \subseteq nghbr_8(s)$ is the set of visible neighbors of vertex s, that is, neighbors of s that have line-of-sight to s. $nghbr_4(s) \subseteq nghbr_8(s)$ is the set of four neighbors to the immediate north, east, south and west of vertex s. We refer to $nghbr_4(s)$ as the crossbar neighbors of s since the four line segments that connect them to s form a crossbar.

3 Basic Theta*

All path-planning algorithms in this paper build on Basic Theta* and use the h-values $h(s) = c(s, s_{goal})$ to focus their search. The h-values correspond to an approximation of the distance from vertex s to the goal vertex. Basic Theta* is the A* variant shown in Algorithm 1 which becomes A* if Lines 43-49 are removed. [Statements in curly braces are to be ignored for now.]

Like A*, Basic Theta* maintains two values for every vertex s: **(1)** its g-value $g(s)$, which is the length of the shortest path from the start vertex to s found so far; and **(2)** its parent *parent*(s) in the search tree, which is used to extract paths after the search terminates. In particular, the path from the start vertex to the goal vertex is extracted by repeatedly following the parent pointers from the goal vertex to the start vertex. We allow Basic Theta* to maintain one additional value for every vertex s which will be required by Incremental Phi*, namely its local parent *local*(s). *local*(s) is the vertex that was expanded when the g-value and parent of s were set. Notice that if Lines 43-49 are removed the local parent and parent are always the same. The local parent path $G_p(s)$ consists of the vertices encountered by repeatedly following the local parent pointers from s to its parent (inclusive of the endpoints). Like A*, Basic Theta* also maintains two global data structures: **(1)** the open-list *open*, which is a priority queue that contains all vertices that it considers for expansion; and **(2)** the closed-list *closed*, which is a set that contains all vertices that it has already expanded.[1] When Basic Theta* expands a vertex s in procedure ComputeShortestPath, it updates the g-value, parent and local parent of each unexpanded visible neighbor s' of s by considering two paths in procedure ComputeCost: **Path 1** is the path considered by A*. It goes from the start vertex to s and from s to s' in a straight-line (Line 51). **Path 2** is

```
1  Main()
2      Initialize();
3      ComputeShortestPath();
4      if g(s_goal) ≠ ∞ then
5          return "path found";
6      else
7          return "no path found";
8  end
9  Initialize()
10     open := closed := ∅;
11     InitializeVertex(s_start);
12     InitializeVertex(s_goal);
13     g(s_start) := 0;
14     parent(s_start) := s_start;
15     open.Insert(s_start, g(s_start) + h(s_start));
16 end
17 InitializeVertex(s)
18     g(s) := ∞;
19     parent(s) := NULL;
20     {local(s) := NULL};
21     {lb(s) := −∞};
22     {ub(s) := ∞};
23 end
24 ComputeShortestPath()
25     while open.TopKey() < g(s_goal) + h(s_goal) do
26         s := open.Pop();
27         closed := closed ∪ {s};
28         foreach s' ∈ nghbr_vis(s) do
29             if s' ∉ closed then
30                 if s' ∉ open then
31                     InitializeVertex(s');
32                 UpdateVertex(s, s');
33 end
34 UpdateVertex(s, s')
35     g_old := g(s');
36     ComputeCost(s, s');
37     if g(s') < g_old then
38         if s' ∈ open then
39             open.Remove(s');
40         open.Insert(s', g(s') + h(s'));
41 end
42 ComputeCost(s, s')
43     if lineofsight(parent(s), s') then
44         /* Path 2 */
45         if g(parent(s)) + c(parent(s), s') < g(s') then
46             parent(s') := parent(s);
47             g(s') := g(parent(s)) + c(parent(s), s');
48             {local(s') := s};
49     else
50         /* Path 1 */
51         if g(s) + c(s, s') < g(s') then
52             parent(s') := s;
53             g(s') := g(s) + c(s, s');
54             {local(s') := s};
55 end
```

Algorithm 1: Basic Theta*

the path that allows for any-angle paths. It goes from the start vertex to the parent of s and from the parent of s to s' in a straight-line (Line 45). Since Path 2 is no longer than Path 1 due to the triangle inequality, Basic Theta* uses Path 2 if the parent of s has line-of-sight to s' (Line 43) and then sets the g-value of s' to the length of Path 2, the parent of s' to the parent of s and the local parent of s' to s. Otherwise, Basic Theta* uses Path 1 and sets the g-value of s' to the length of

[1] *open* is a priority queue. *open.Insert*(s, x) inserts vertex s with key x into the priority queue *open*, *open.Remove*(s) removes vertex s from the priority queue *open*, and *open.Pop*$()$ removes a vertex with the smallest key from priority queue *open* and returns it. *open.TopKey*$()$ returns the smallest key of all vertices in the priority queue *open* unless *open* is empty in which case it returns ∞.

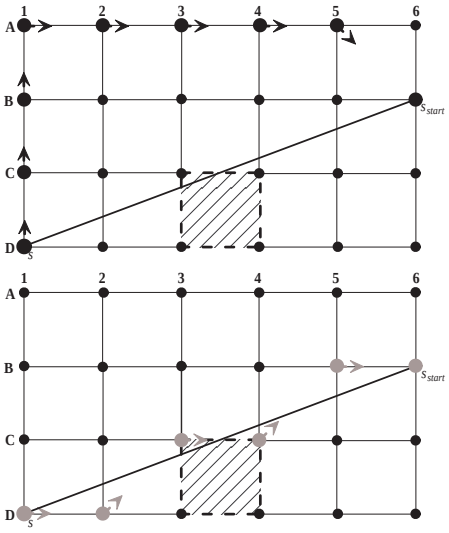

Figure 3: Path 2 Parent Problem

Path 1 and both the parent and local parent of s' to s. The key idea behind Basic Theta* and Path 2 is that unlike A*, where the parent of a vertex must be a visible neighbor of that vertex, Basic Theta* allows the parent of a vertex to be any vertex.

Figure 2 shows a trace of Basic Theta*. The vertices are labeled with their g-values and parents. The hollow circle indicates the vertex that is currently being expanded, the larger black circles indicate vertices that have already been expanded, and the arrows point to the parent of a vertex. The start vertex A4 is expanded first (left) and B3 is expanded next (right). Consider Figure 2 (right), where B3 with parent A4 is expanded. C3 is an unexpanded visible neighbor of B3 that does not have line-of-sight to A4 and thus is updated according to Path 1. Basic Theta* therefore sets both its parent and local parent to B3. On the other hand, B2 is an unexpanded visible neighbor of B3 that does have line-of-sight to A4 and thus is updated according to Path 2. Basic Theta* therefore sets its parent to A4 and its local parent to B3.

While Basic Theta* is not guaranteed to find shortest paths it finds much shorter paths than A* [Nash *et al.*, 2007]. When applied to planning with the freespace assumption, the robot must at least find a new short path each time it observes its existing path to be blocked. This problem could certainly be solved by repeatedly performing single shot Basic Theta* searches, but this would be too slow on large grids. Alternatively the robot could reuse information from the previous search when finding a new short path. Incremental path-planning algorithms, such as D* [Stentz, 1995] and D* Lite [Koenig and Likhachev, 2002] extend A* to planning with the freespace assumption by efficiently reusing information from the previous search to speed up the next one. Despite the similarities between Basic Theta* and A*, these methods do not apply to Basic Theta* because they assume that the parent of a vertex in the search tree must also be its neighbor and Basic Theta* allows the parent of a vertex in the search tree to be any vertex. We refer to this as the **Path 2 Parent Problem**. Reusing information from the previous search requires that all vertices that no longer have line-of-sight to their parent due to a newly blocked cell be identified. The fact that the paths found by A* are constrained to grid edges, makes this easy because any vertex that no longer has line-of-sight to its parent must be a corner of a newly blocked cell. For example, assume that cell (B2-B3-C3-C2) becomes blocked in Figure 1 (left). C2 no longer has line-of-sight to its parent B3 which can easily be identified because C2 and B3 are both corners of the newly blocked cell. The paths found by Basic Theta* are not constrained to grid edges, which makes this more difficult because vertices that no longer have line-of-sight to their parent are not necessarily corners of a newly blocked cell. For example, assume that cell (C3-C4-D4-D3) becomes blocked in Figure 3 (top), which is *not* an actual trace of Basic Theta*. The larger black circles and black arrows indicate the local parent path of D1: G_p(D1) = (D1, C1, B1, A1, A2, A3, A4, A5, B6). D1 no longer has line-of-sight to its parent B6, but neither it nor any vertex on its local parent path is a corner of the newly blocked cell.

```
ComputeCost(s, s')
    if lineofsight(parent(s), s') and
    Φ(s, parent(s), s') ∈ [lb(s), ub(s)] and
    ∠(s', parent(s)) is not a multiple of 45° then
        /* Path 2 */
        if g(parent(s)) + c(parent(s), s') < g(s') then
            parent(s') := parent(s);
            g(s') := g(parent(s)) + c(parent(s), s');
            local(s') := s;
            l := min_{s''∈nghbr4(s')}(Φ(s', parent(s), s''));
            h := max_{s''∈nghbr4(s')}(Φ(s', parent(s), s''));
            δ = Φ(s, parent(s), s');
            lb(s') := max(l, lb(s) − δ);
            ub(s') := min(h, ub(s) − δ);
    else
        /* Path 1 */
        if g(s) + c(s, s') < g(s') then
            parent(s') := s;
            g(s') := g(s) + c(s, s');
            local(s') := s;
            lb(s') := −45°;
            ub(s') := 45°;
end
```

Algorithm 2: Phi*

4 Phi*

Phi* is a version of Basic Theta* that can be made incremental because it addresses the Path 2 Parent Problem by maintaining **Property 1**: *The local parent path of any vertex that no longer has line-of-sight to its parent after cells become blocked must contain some corner of a newly blocked cell.*[2] Phi* is identical to Algorithm 1 except for procedure ComputeCost shown in Algorithm 2. [Statements in curly braces are to be executed from now on.] Phi* is complete and correct. We constructed it so that we can prove the following lemma which implies Property 1:

Lemma 1. *The local parent path $G_p(s)$ of any vertex s contains at least one corner of each cell that the line segment $\mathcal{L}(s, \text{parent}(s))$ traverses.*

[2]Statements like these only apply to vertices that are in either *open* or *closed*, but we do not mention this to improve readability.

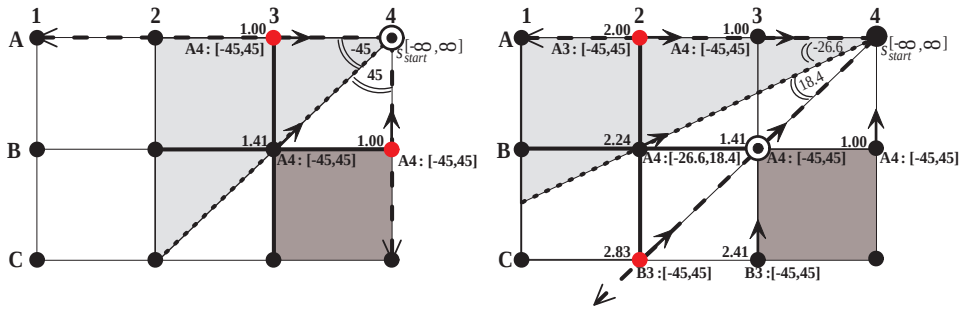

Figure 4: Example Trace of Phi*

If the line segment from a vertex to its parent traverses a blocked cell, then it must touch at least one of the four grid edges (two horizontal and two vertical) that form the perimeter of the traversed blocked cell. We constructed Phi* so that the local parent path of any vertex contains at least one of the two end points of each horizontal and vertical grid edge that the line segment from the vertex to its parent touches. Phi* satisfies this property by maintaining an angle range for each vertex that restricts which of its unexpanded visible neighbors can be updated according to Path 2. To illustrate this assume that cell (C3-C4-D4-D3) becomes blocked in Figure 3 (bottom), which is *not* an actual trace of Phi*. The larger grey circles and grey arrows indicate the local parent path of D1: G_p(D1) = (D1, D2, C3, C4, B5, B6). D1 no longer has line-of-sight to its parent B6, but unlike Figure 3 (top) this can easily be identified because a vertex on its local parent path is a corner of the newly blocked cell. In fact, the local parent path of D1 contains at least one of the two end points of each horizontal and vertical grid edge that the line segment from D1 to its parent B6 touches. Figure 4 shows a trace of Phi*. The vertices are now labeled with their angle ranges in addition to their g-values and parents.

4.1 Angle Ranges

Phi* maintains two additional values for every vertex s: a lower angle bound $lb(s)$ and an upper angle bound $ub(s)$ that together form the angle range $[lb(s), ub(s)]$. We define $\Phi(s, p, s')$ to explain their meaning. $\Phi(s, p, s')$ is the (smaller) angle $\angle(s, p, s') \in [-180°, 180°)$ between the ray from p through s' and the ray from p through s. It is positive iff the former ray is counterclockwise of the latter ray. Assume that Phi* expands a vertex s and considers a Path 2 update for an unexpanded visible neighbor s' of s. Phi* constrained the angle range of s so that the local parent path of s' contains at least one corner of each cell that the line segment $\mathcal{L}(s', parent(s))$ traverses if $\Phi(s, parent(s), s') \in [lb(s), ub(s)]$. In Figure 4 (right) the angle range of B2 with parent A4 is denoted by the two dashed rays emanating from A4. The dotted line within this angle range is Φ(B2, A4, B2) = 0, which is the reference point of the angle range.

4.2 Updating Angle Ranges

When Phi* expands a vertex s, it updates the g-value and parent of each unexpanded visible neighbor s' of s by considering Path 1 and Path 2. It updates the g-value, parent and local parent of s' in the same way as Basic Theta*, but also updates the angle range of s'.

- **Path 1:** When Phi* expands A4 in Figure 4 (left), it updates its unexpanded visible neighbor B3 according to Path 1 (Line 70). Phi* sets the angle range of B3 to the angle range defined by the crossbar centered at B3, as follows: It computes $\Phi(s', parent(s), s'') = \Phi$(B3, A4, s'') for each crossbar neighbor s'' of B3, namely A3, B4, C3 and B2. (This is independent of the blocked cell with corner B3.) It sets the upper angle bound of B3 to the maximum of the four computed angles, namely 45.0° due to B4. It sets the lower angle bound of B3 to the minimum of the four computed angles, namely -45.0° due to A3. Phi* does not need to compute these angle bounds since the upper angle bound is always 45.0° and the lower angle bound is always -45.0° in the Path 1 case (Lines 74 and 75).
- **Path 2:** When Phi* expands B3 in Figure 4 (right), it updates its unexpanded visible neighbor B2 according to Path 2 (Line 59). Phi* then sets the angle range of B2 to the intersection of the angle range defined by the crossbar centered at B2 and the angle range of its local parent B3, as follows: It computes $\Phi(s', parent(s), s'') = \Phi$(B2, A4, s'') for each crossbar neighbor s'' of B2, namely A2, B3, C2 and B1. Phi* computes the maximum of the four computed angles (Line 64), namely 18.4° due to B3 and C2. It sets the upper angle bound of B2 to the minimum of that angle and the upper angle bound of B3 shifted so that the reference point is now B2 rather than B3 (Line 67), resulting in 18.4°. Phi* also computes the minimum of the four computed angles (Line 63), namely -26.6° due to A2. It sets the lower angle bound of B2 to the maximum of that angle and the lower angle bound of B3 shifted so that the reference point is now B2 rather than B3 (Line 66), resulting in -26.6°. Thus, it calculates the angle range of vertex s' as for Path 1, but then intersects it with the angle range of vertex s.

Assume that Phi* expands a vertex s. If it updates the unexpanded visible neighbor s' of s according to Path 2, then it sets the angle range of s' to the intersection of the angle range defined by the crossbar centered at s' and the angle range of the local parent of s'. Thus, it sets the angle range of s' to the intersection of the angle ranges defined by all crossbars centered on vertices that are members of the local parent path

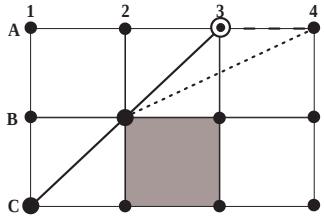

Figure 5: Tie Breaking and the Triangle Inequality

of s', including s', but not the parent of s'. We refer to this as the **Crossbar Property**. If Phi* updates the unexpanded visible neighbor s' of s according to Path 1, then it sets the angle range of s' to the angle range defined by the crossbar centered at s'. Thus, the Crossbar Property holds in this case as well.

4.3 Using Angle Ranges

Due to the Crossbar Property any ray within the angle range of a vertex s will touch all crossbars centered on vertices that are members of the local parent path of s. Therefore any unexpanded visible neighbor s' of s that satisfies $\Phi(s, \mathit{parent}(s), s') \in [\mathit{lb}(s), \mathit{ub}(s)]$ (Line 57) will be such that the line segment $\mathcal{L}(s', \mathit{parent}(s))$ touches all crossbars centered on vertices that are members of the local parent path of s'. Thus the local parent path of s' contains at least one corner of each cell that the line segment traverses, which is how Phi* satisfies Lemma 1.

4.4 Tie Breaking and the Triangle Inequality

When Phi* expands a vertex s, it updates each unexpanded visible neighbor s' of s by considering Path 1 and Path 2. $\angle(s, p)$ is the smaller angle formed by the ray from p through s and the vertical line through p. Path 2 is no longer than Path 1 due to the triangle inequality. They are equally long if $\angle(s', \mathit{parent}(s))$ is a multiple of 45°, that is, if the vertices lie on a horizontal, vertical or 45° line. In this case, it is better to update s' according to Path 1, which explains the third condition on Line 57. The reason is that an update according to Path 1 results in faster line-of-sight checks and often slightly reduces path lengths (although this is not guaranteed). For example in Figure 5, if A3 has parent C1 when it is expanded by Phi*, then it updates its unexpanded visible neighbor A4 according to Path 1 (dashed line segment). However, if A3 has parent B2 when it is expanded by Phi*, then it updates its unexpanded visible neighbor A4 according to Path 2 (dotted line segment), resulting in a shorter path.

5 Incremental Phi*

Incremental Phi* is an incremental version of Phi* that recomputes any-angle paths faster than repeated single shot Basic Theta* or Phi* searches when cells become blocked. Incremental Phi* is identical to Algorithm 2 except for procedures Main and ClearSubtree shown in Algorithm 3.[3] Incremental Phi* is complete and correct, but can find paths

```
77  Main()
78      Initialize();
79      while true do
80          ComputeShortestPath();
81          Wait for cells to become blocked ;
82          foreach newly blocked cell c do
83              Update blockage status of cell c to blocked ;
84              foreach s' ∈ corners(c) do
85                  if ((s' ∈ closed or s' ∈ open) and s' ≠ s_start)
                    then
86                      ClearSubtree(s');
87  end
88  ClearSubtree(s)
89      under := over := ∅;
90      under.Enqueue(s);
91      while under ≠ ∅ do
92          u := under.Dequeue();
93          over.Enqueue(u);
94          InitializeVertex(u);
95          if u ∈ open then
96              open.Remove(u);
97          if u ∈ closed then
98              closed.Remove(u);
99          foreach s' ∈ nghbr_8(u) do
100             if local(s') = u then
101                 under.Enqueue(s');
102     while over ≠ ∅ do
103         v := over.Dequeue();
104         foreach s' ∈ nghbr_vis(v) do
105             if s' ∈ closed then
106                 UpdateVertex(s', v);
107 end
```

Algorithm 3: Incremental Phi*

that are longer than those found by repeated single shot Basic Theta* or Phi* searches, especially if cells can become unblocked (which was not permitted in our experiments). We constructed Phi* so that all vertices that no longer have line-of-sight to their parent due to newly blocked cells can be identified and removed quickly. Incremental Phi* uses a preprocessing technique that does this in a way that is similar in spirit to the preprocessing technique used by Differential A* [Trovato and Dorst, 2002] (Lines 81-86). We constructed this preprocessing technique so that we can prove the following lemma:

Lemma 2. *Every vertex in either* open *or* closed *has line-of-sight to its parent each time procedure ComputeShortestPath is called on Line 80.*

If every vertex in either *open* or *closed* has line-of-sight to its parent immediately prior to procedure ComputeShortestPath then every vertex in either *open* or *closed* has line-of-sight to its parent immediately afterwards as well. If cells become blocked (Line 83), Incremental Phi* uses the preprocessing technique to maintain Lemma 2 prior to the next call to procedure ComputeShortestPath. The preprocessing technique uses Property 1 to identify all vertices that no longer have line-of-sight to their parent. This can be done easily because Property 1 ensures that the local parent path of any vertex that no longer has line-of-sight to its parent after cells become blocked must contain some corner of a newly blocked cell. The preprocessing technique thus calls procedure Clear-

[3] *under* is a FIFO queue. *under.Enqueue*(s) inserts vertex s at the end of the FIFO queue *under* and *under.Dequeue* removes a vertex from the front of the FIFO queue *under*. *over* is a FIFO queue similar to *under*.

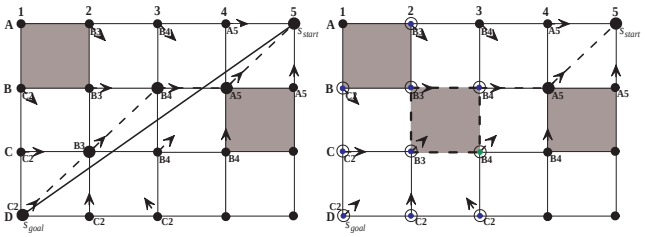

Figure 6: Example Trace of Incremental Phi*

Subtree on each corner of each newly blocked cell (Lines 82 and Line 84), where each call performs a breadth-first search by following local parent pointers backwards. Procedure ClearSubtree removes all encountered vertices from both *open* and *closed* and reinitializes their values as if they had never been visited by procedure ComputeShortestPath (Lines 94-98). It then updates *open* by reinserting all vertices with a visible neighbor in *closed* back into *open* (Line 106), which allows the next call to procedure ComputeShortestPath to be a standard Phi* search that reuses information from the previous search. Notice that ClearSubtree is fast because it uses FIFO queues. Consider Figure 6 in which black arrows point to the local parent of a vertex. Figure 6 (left) shows a snap shot after the first call to procedure ComputeShortestPath. Assume that the cell (B2-B3-C3-C2) becomes blocked in Figure 6 (right). The preprocessing technique would iterate over the four corners of the newly blocked cell, calling procedure ClearSubtree on each corner that is in either *open* or *closed* (Line 85). If procedure ClearSubtree is applied to B3 first, the hollow blue vertices B1, C1, D1, A2, B2, C2, D2, B3 and D3 would be reinitialized after which B3 would be reinserted into *open* with parent A5 and local parent B4. If procedure ClearSubtree were then applied to C3 the hollow green vertex C3 would be reinitialized and reinserted into *open* with parent and local parent B4. Procedure ClearSubtree would not be applied to C2 and B2 as neither corner would be in *open* or *closed* after ClearSubtree was applied to B3 and C3.

6 Moving Robot

Incremental Phi* needs to handle a moving robot to apply to planning with the freespace assumption. We proceed similarly to D* [Stentz, 1995] and D* Lite [Koenig and Likhachev, 2002]. We let Incremental Phi* search from the goal vertex to the current vertex of the robot. The h-values then correspond to an approximation of the distance from a vertex to the current vertex of the robot. The keys of the vertices in the priority queue *open* are thus no longer valid when the robot moves. Incremental Phi* uses the heap reordering technique of D* [Stentz, 1995] and D* Lite [Koenig and Likhachev, 2002] instead of repeatedly reordering the priority queue by recomputing all keys.

7 Experimental Results

We now compare the path length, number of vertex expansions and runtime of Phi* and Incremental Phi* against those of Basic Theta*. Unlike A*, we break ties among vertices with the same key in favor of the vertex with the smaller g-value in all of our experiments since this tie-breaking scheme finds shorter paths for variants of Basic Theta*[Nash *et al.*, 2007]. We report ratios such that values larger than 1.0 indicate that Basic Theta* did worse than Phi* or Incremental Phi*. We use grids with a given percentage of randomly blocked cells (random grids) and scaled indoor and outdoor maps from robotics and the real-time game Baldur's Gate II [Bulitko *et al.*, 2005] (non-random grids). We averaged over 500 path-planning problems for random grids and 650 path-planning problems for non-random grids (50 path-planning problems were run on each non-random grid of which there were twelve game maps and one robotics map). For random grids, the start vertex is always in the bottom left corner and the goal vertex is randomly selected on the right border. For non-random grids, the start and goal vertices are randomly selected with the constraint that the distance between them is at least 250.

% Blocked Cells	Path Length	Vertex Expansions	Runtime
0	1.0000	3.4788	2.2412
5	1.0000	1.1923	0.9900
10	1.0002	1.1155	0.9167
20	1.0004	1.0669	0.9012

Table 1: Single-Shot Searches on Random Grids

7.1 Phi*

Table 1 compares Phi* and Basic Theta* for single-shot searches on random grids of size 500×500, where both path-planning algorithms search from the start vertex to the goal vertex. Phi* finds paths of essentially the same length as Basic Theta*, which is important because Basic Theta* typically finds shorter paths than Field D* [Nash *et al.*, 2007]. Phi* runs faster than Basic Theta* on grids with no blocked cells and slightly slower on grids with larger percentages of blocked cells, for the following reason: Phi* expands far fewer vertices than Basic Theta* on grids with no blocked cells because it updates fewer vertices according to Path 2 than Basic Theta* due to its additional angle range constraint. When Phi* updates a vertex according to Path 1 and Basic Theta* updates that same vertex according to Path 2, Phi* inserts it into *open* with a larger key than Basic Theta*. Thus, Phi* is more likely to finish the search without expanding that vertex than Basic Theta*. Phi* expands more of these vertices as the percentage of blocked cells increases.

7.2 Incremental Phi*

Table 2 compares Incremental Phi* and Basic Theta* for planning with the freespace assumption. For these experiments we maintained two grids: one represented the terrain (terrain grid) and the other represented the knowledge that the robot has of the terrain (knowledge grid). The robot finds a short path from its current vertex, initially the start vertex, to the goal vertex in its knowledge grid and then repeatedly moves one unit along this path. After each move, it scans all cells within a given sensor radius around its current point and updates the scanned cells in its knowledge grid to match the blockage status of those same cells in the terrain grid. The robot then finds a new short path from its current vertex to the goal vertex in its knowledge grid and repeats this process until it reaches the goal vertex. Unblocked cells in the

% Blocked Cells	Path Length		
	100×100	250×250	500×500
0	0.9902	0.9902	0.9902
5	0.9925	0.9922	0.9919
10	0.9947	0.9937	0.9944
20	0.9995	0.9970	0.9971

% Blocked Cells	Vertex Expansions		
	100×100	250×250	500×500
0	3.2551	7.8718	15.3940
5	4.4492	11.5943	25.7134
10	5.1763	13.7031	31.1268
20	5.7573	15.4465	33.8728

% Blocked Cells	Runtime		
	100×100	250×250	500×500
0	1.2870	3.3653	7.0040
5	1.5056	3.9075	8.0542
10	1.7062	4.5855	10.0422
20	1.9680	5.6277	11.7307

(a) Grid Sizes and Percentages of Blocked Cells on Random Grids

Sensor Radius	Path Length	Vertex Expansions	Runtime
5	0.9954	31.9153	9.6181
10	0.9961	28.3299	8.1223
20	0.9966	19.8923	5.5846

(b) Sensor Radius on Random Grids

Path Length	Vertex Expansions	Runtime
1.0037	25.1068	12.0073

(c) Non-Random Grids

Table 2: Planning with the Freespace Assumption

knowledge grid can become blocked in the terrain grid but not vice versa. For random grids, the knowledge grid contained a given percentage of randomly blocked cells and the terrain grid corresponded to the knowledge grid except that twenty percent of randomly chosen cells were made blocked in addition to the blocked cells in the knowledge grid. For non-random grids, the knowledge grid contained only unblocked cells and the terrain grid corresponded to the given map. Incremental Phi* operated as described earlier. Basic Theta* always searched from the start vertex to the goal vertex if the existing path in the knowledge grid became blocked. The number of vertex expansions includes all vertices that were removed from *open* and updated their unexpanded visible neighbors. The runtime includes the entire time from the start of the search until the robot reached the goal vertex. Table 2(a) reports on random grids with a sensor radius of three. As the grid size and the percentage of blocked cells in the knowledge grid increases, Incremental Phi* expands many fewer vertices than Basic Theta* and its speedup relative to Basic Theta* increases. Table 2(b) reports on random grids of size 500×500 with ten percent blocked cells in the knowledge grid. As the sensor radius decreases, Incremental Phi* expands many fewer vertices than Basic Theta* and its speedup relative to Basic Theta* increases. Table 2(c) reports on non-random grids of size 500×500 and a sensor radius of three, where the path-planning algorithms often needed to replan more frequently than on random grids. Across all tables, Incremental Phi* finds paths of essentially the same length as Basic Theta* (although both path-planning algorithms search in opposite directions and thus the paths of the robot quickly diverge). However, Incremental Phi* does so up to 12 times faster than Basic Theta*.

8 Conclusions

In this paper, we made the any-angle path-planning algorithm Basic Theta* incremental. We presented Phi*, a version of Basic Theta* that can be made incremental, and then extended it to Incremental Phi*. Both Phi* and Incremental Phi* are complete and correct. We demonstrated that Incremental Phi* is simple, that it finds paths of essentially the same length as Basic Theta* and that it can provide a speedup of approximately one order of magnitude for path planning with the freespace assumption. Future research will be directed toward making Incremental Phi* even faster by **(1)** integrating it with Angle Propagation Theta* [Nash *et al.*, 2007] so that in the worst case its runtime per vertex expansion is constant and not linear in the number of cells **(2)** using quantities that are not as computationally intensive as angles and **(3)** identifying and repairing vertices lazily rather than eagerly. Future research should also be directed toward obtaining analytical bounds on the lengths of the paths found by Incremental Phi* and comparing these lengths with the lengths of the paths found by Basic Theta* and the lengths of the truly shortest any-angle paths.

References

[Bulitko *et al.*, 2005] V. Bulitko, N. Sturtevant, and M. Kazakevich. Speeding up learning in real-time search via automatic state abstraction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1349–1354, 2005.

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

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

[Hart *et al.*, 1968] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SCC-4(2):100–107, 1968.

[Koenig and Likhachev, 2002] S. Koenig and M. Likhachev. D* Lite. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 476–483, 2002.

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

[Nash *et al.*, 2007] A. Nash, K. Daniel, S. Koenig, and A. Felner. Theta*: Any-angle path planning on grids. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1177–1183, 2007.

[Stentz, 1995] A. Stentz. The focussed D* algorithm for real-time replanning. In *International Joint Conference on Ariticial Intelligence*, pages 1652–1659, 1995.

[Trovato and Dorst, 2002] K. Trovato and L. Dorst. Differential A*. In *IEEE Transaction on Knowledge and Data Engineering*, pages 1218–1229, 2002.