Australian Government
Department of Defence
Science and Technology

Speeding Up Phi*: Refined Foundations for Dynamic Path Planning at Any Angle

Patrick Chisan Hew

Joint and Operations Analysis Division
Defence Science and Technology Group

DST-Group–TN–1805

ABSTRACT

‘Any-angled’ planning has emerged as a promising approach to finding short paths through a terrain that has obstacles. Phi* (2009) is the foundation of Incremental Phi* which handles terrains where obstacles are not known ahead of time and therefore have to be discovered during the path planning process (dynamic single pair shortest path). This technical note explores the following refinements to speed up Phi*: Bounds Known, Integer operations only, Expensive Last testing, Lazy evaluation and Angle Propagation. Expensive Last reduces runtime by roughly 5 percent. Expensive Last with Angle Propagation performs line-of-sight testing in constant time and reduces runtime by roughly 15–30 percent. Integer avoids floating point errors that could compromise the paths found by Phi*. The findings will be useful to developers and users of dynamic path planning algorithms in two-dimensional terrains.

RELEASE LIMITATION

Approved for Public Release

Produced by

Joint and Operations Analysis Division
506 Lorimer St,
Fishermans Bend, Victoria 3207, Australia

Telephone: *1300 333 362*

August, 2018

APPROVED FOR PUBLIC RELEASE

Speeding Up Phi*: Refined Foundations for Dynamic Path Planning at Any Angle

Executive Summary

The work in this report is part of an investigation into a new approach to finding short paths through a terrain that has obstacles. These so-called *any-angled* pathfinding algorithms could be used to guide aircraft or watercraft through anti-access/area-denied environments. While the algorithms typically do not obtain the shortest (optimal) path, they can obtain good (near-optimal) results, quite quickly, on computers that are cheap and widely available. Moreover the entry cost to learning and developing the algorithms is low: when undergraduate students first study pathfinding it is through the A* algorithm, and the first any-angled algorithm differs from A* by less than 10 lines of code. Finally the algorithms are still in their early phases of development and thus there is considerable potential for growth. The nearer term applications of any-angled path finding include algorithms for navigating entities in simulated environments, and for navigation of robotic and uninhabited vehicles.

The report focusses on improvements to the algorithm Phi*. The heritage of Phi* is as follows: A* finds a shortest path from a start to a goal through a terrain in which the obstacles are known ahead of time. The terrain is typically modelled as a two-dimensional array of cells that are either accessible or blocked as this model is supported by readily-available data sets. But the textbook application of A* under this terrain model causes 'digitization bias' in which the path is artificially constrained to following the edges of a cell or stepping diagonally across them. Basic Theta* introduces a clever trick that significantly reduces 'digitization bias' with at-most a small increase in runtime. Phi* extends from Basic Theta* by recording information so that if new obstacles are discovered then the previously-planned paths can be reused to find paths around those obstacles. The overall framework for planning the paths as the terrain changes is Incremental Phi* where Phi* is invoked whenever obstacles are discovered.

The extensions to Phi* that were explored are Bounds Known, Integer operations only, Expensive Last testing, Lazy evaluation and Angle Propagation. Prior to the investigation, each extension was hypothesized to improve the performance of Phi* but whether this was true, and the magnitude of improvement, were both unknown. The investigation found that Expensive Last reduces runtime by roughly 5 percent. Expensive Last with Angle Propagation performs line-of-sight testing in constant time and reduces runtime by roughly 15–30 percent. Integer avoids floating point errors that could compromise the paths found by Phi*.

Overall the performance gain from the proposed improvements is disappointing, and not of the magnitude that was anticipated or that would make new implementations of Phi* worthwhile. The report is being released so that the approach is documented and the results are available to aid future research into dynamic path planning, even if the results are negative.

This page is intentionally blank

Contents

Figures

Tables

This page is intentionally blank

1. Introduction

We study the problem of finding a short path that a vehicle can traverse from a start to a goal when motion is constrained to a two-dimensional surface and there are obstacles that must be discovered and avoided (a subset of dynamic single pair shortest path). A military example is to extricate a watercraft from a minefield when the mines have to be localized as the watercraft moves. Similar problems arise in mobile robotics and in the realistic-looking routing of entities in computer games. We assume the following: first, that the terrain has been discretized into a two-dimensional array of accessible and blocked cells (a *binary occupancy grid* – Figure 1.a into Figure 1.b). The cells' corners are declared to be the vehicle's *feasible locations*. Second, that if a line-of-sight between feasible locations passes only through accessible cells then the vehicle can traverse that line-of-sight in the original terrain. Third, that when travelling in accessible terrain, the vehicle can move at the same speed in any direction (motion is isotropic). Finally, we assume that the vehicle's dimensions and turning circle are small compared to the cells so it can 'squeeze through the diagonal' between cells that touch at corners but not faces.

Unfortunately, while it is easy to represent terrain as a two-dimensional array of accessible and blocked cells, the search within this representation for a shortest path from start to goal is not trivial. The *true shortest paths* between any two locations are the ones that minimize the travelling cost between them (this holds for any locations in the terrain, not just the feasible ones). In isotropic terrain, travelling cost is directly proportional to path length. If the locations have line-of-sight then the true shortest path between them is along that line-of-sight. So to approximate a true shortest path from a start to a goal, it is sufficient to find a feasible location near to the start, a feasible location near the goal and obtain a *shortest vertex path* between those two locations. Here, a *vertex path* is a path through the visibility graph on the feasible locations; by definition, vertices are adjacent in the *visibility graph* if they have line-of-sight to each other (Figure 1.c). Thus in principle, to obtain a shortest vertex path we need only search the visibility graph of feasible locations, but in practice, the graph is expensive to compute and too large to search by brute force.

It is therefore common to consider the paths through the *grid graph* on the feasible locations, declaring vertices as adjacent if they share an accessible cell (same edge or diagonally across). Hence in a *grid path*, each step is along the edge of an accessible cell or diagonally across one; this is equivalent to 8-connected motion (Figure 1.d). Grid paths suffer from 'digitization bias' [Tsitsiklis 1995]: even if two locations have line-of-sight, the minimum travelling cost between them can still be greater than the distance along the line-of-sight.

We thus consider approaches that can find good approximations to the true shortest paths. Of particular interest are *any-angled* approaches that look for short vertex paths. Nash, Koenig & Likhachev (2009) proposed Incremental Phi* for dynamic planning of short paths at any angle from a start to a goal. Incremental Phi* is built on Phi* which plans a short path in the terrain as perceived at some time. Incremental Phi* iteratively calls Phi* when the knowledge about the terrain is updated. Meanwhile Phi* is designed so that previously-planned paths can be used when planning new ones, rather than replanning afresh on each update. Phi* does not guarantee a shortest vertex path as it does not search the entire visibility graph. However it does consider all possible grid paths. Consequently the path it finds will be no longer than the shortest grid path and could well be shorter.

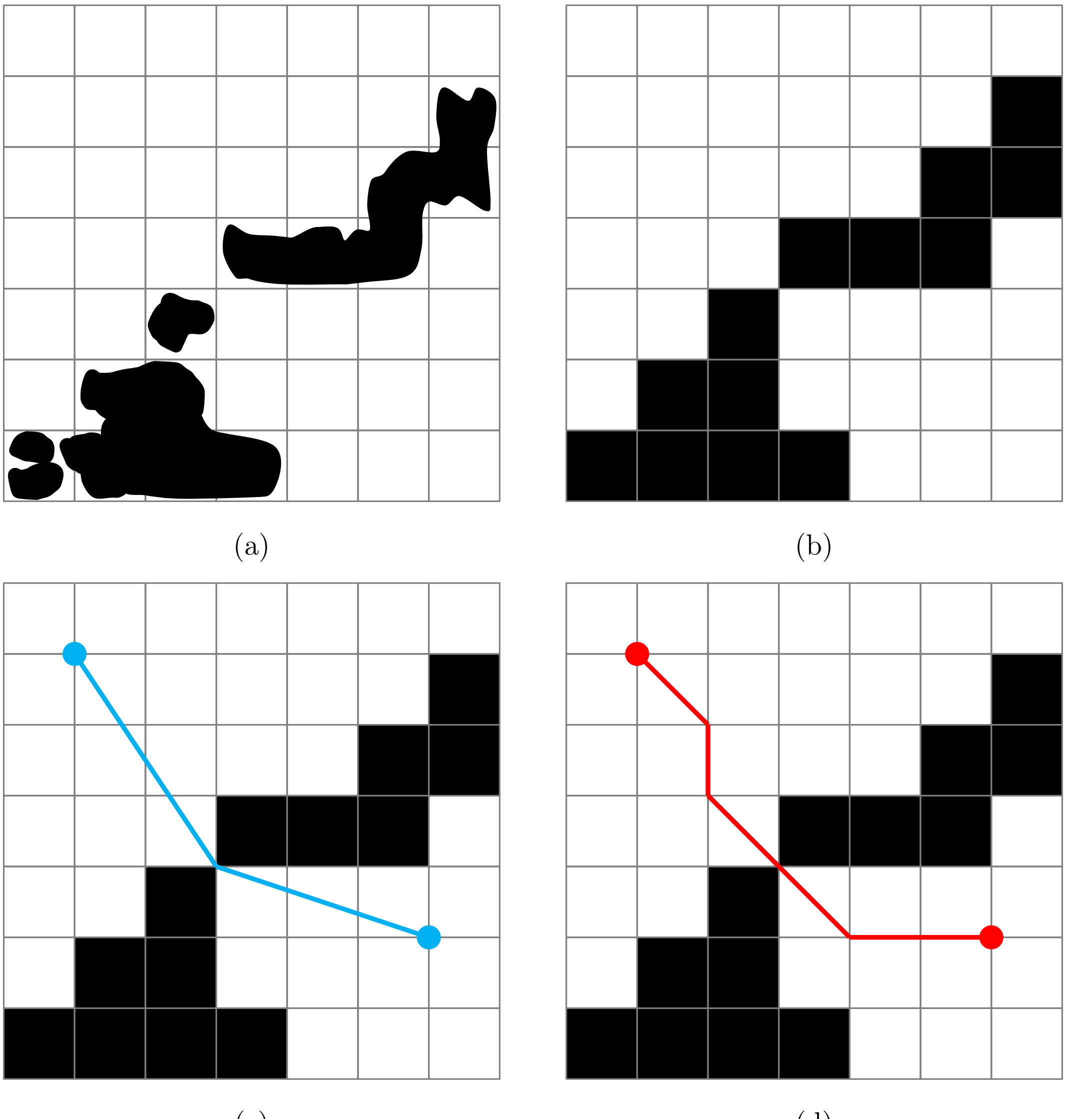

Figure 1: Representing the terrain to find short paths:
a) Locations are either habitable or not (white is accessible, black is forbidden).
b) Discretize into a two-dimensional array of accessible and blocked cells.
c) Vertex paths proceed along feasible locations that have line-of-sight.
d) Grid paths suffer from 'digitization bias'.

This article examines a number of refinements to speed up Phi*. The proposed extensions are novel or implement ideas that have been mentioned as possibilities but have yet to be tested (in these latter cases, we provide references to the original proposals). This article will be useful to readers who are considering Incremental Phi* as a potential solution to their dynamic path planning problems.

Notation: s_x, s_y denote the x, y coordinates of location s with respect to whichever coordinate system is in use. Unless stated otherwise, we will choose coordinate systems that have their origin at a cell corner, with axes that are parallel with the cells' edges and that make the cells of unit size. $\overline{uv}$ denotes the line segment connecting u with v, $\overrightarrow{uv}$ denotes the vector from u to v. We use short-circuit evaluation: If x is false then x **and** y evaluates to false without evaluating y. Likewise if x is true then x **or** y evaluates to true without evaluating y. Function and procedure names are presented as `FunctionName`, variable names as `variableName`.

2. Original Formulations

2.1. Basic Theta*

Basic Theta* derives from A*. They are framed inside Algorithm 1, for planning a path from a start to a goal across a static terrain that has been discretized into a rectilinear grid of accessible and blocked cells. Recall that A* solves the single-pair shortest path problem for any given (static) graph. We focus on the grid graph of 8-connectedness. We make this specialization by declaring v as adjacent to u if it is visible and 8-connected to u (line 22). In this context, we have A* for grid paths at Algorithm 2. As noted above, grid paths suffer from 'digitization bias' (Figure 1). In principle we could reduce the 'digization bias' by applying A* to the visibility graph, but said approach is impractical.

Basic Theta* diverges from A* by changing `ComputeCost`, as seen at Algorithm 3. When we call `ComputeCost` in A*, we know that we can reach vertex v from vertex u; indeed u is the predecessor of v in a path to v. The question is whether u is the predecessor of v in a *shortest* path to v. If going via u yields a shorter path to v than via the currently-assigned predecessor then we will accept that path.

In Basic Theta*, `ComputeCost` tests for whether we can bypass u and come directly from its predecessor p. This situation arises when p has line-of-sight to v (the *line-of-sight test*) and going via p yields a shorter path than via the currently-assigned predecessor (the *shortening test*). If we can bypass u then 'digitization bias' will be reduced.

Algorithm 1: Static path planning in 8-connected grids (Definitions at Table 1).

```
Main()
    Initialize()
    ComputeShortestPath()
    if g($s_{\mathrm{Goal}}$) $\neq \infty$ then
        return 'Path found'
    else
        return 'No path found'
Initialize()
    $open \leftarrow \varnothing$
    $closed \leftarrow \varnothing$
    InitializeVertex($s_{\mathrm{Start}}$)
    InitializeVertex($s_{\mathrm{Goal}}$)
    g($s_{\mathrm{Start}}$) $\leftarrow 0$
    cameFrom($s_{\mathrm{Start}}$) $\leftarrow s_{\mathrm{Start}}$
    $open$.InsertWithPriority($s_{\mathrm{Start}}$, g($s_{\mathrm{Start}}$) + h($s_{\mathrm{Start}}$))
ComputeShortestPath()
    while $open$.SmallestPriority() < g($s_{\mathrm{Goal}}$) + h($s_{\mathrm{Goal}}$) do
        $u \leftarrow open$.PopMinimumElement()
        $closed$.Add($u$)
        ExpandVertex($u$)
ExpandVertex($u$)
    foreach $v \in$ VerticesVisibleAnd8ConnectedTo($u$) do          ˙ 'Expand' $u$ into $\{v\}$.
        if $v \notin closed$ then
            if $v \notin open$ then
                InitializeVertex($v$)
            UpdateVertex($v$, $u$)                                    ˙ Update $v$ given $u$.
UpdateVertex($v$, $u$)
    $\mathrm{g}_{\mathrm{Old}} \leftarrow$ g($v$)
    ComputeCost($v$, $u$)
    if g($v$) < $\mathrm{g}_{\mathrm{Old}}$ then
        if $v \in open$ then
            $open$.Remove($v$)
        $open$.InsertWithPriority($v$, g($v$) + h($v$))
```

Algorithm 2: A*.

```
InitializeVertex($s$)
    $\mathtt{g}(s) \leftarrow \infty$
    $\mathtt{cameFrom}(s) \leftarrow \varnothing$
ComputeCost($v$, $u$)
    if $\mathtt{g}(u) + \mathtt{c}(u, v) < \mathtt{g}(v)$ then                ˙Go via $u$.
        $\mathtt{g}(v) \leftarrow \mathtt{g}(u) + \mathtt{c}(u, v)$
        $\mathtt{cameFrom}(v) \leftarrow u$
```

Algorithm 3: Basic Theta*.

```
ComputeCost($v$, $u$)
    $p \leftarrow \mathtt{cameFrom}(u)$
    if LineOfSight($p$, $v$) then
        if $\mathtt{g}(p) + \mathtt{c}(p, v) < \mathtt{g}(v)$ then            ˙Path 2: Bypass $u$.
            $\mathtt{g}(v) \leftarrow \mathtt{g}(p) + \mathtt{c}(p, v)$
            $\mathtt{cameFrom}(v) \leftarrow p$
    else
        if $\mathtt{g}(u) + \mathtt{c}(u, v) < \mathtt{g}(v)$ then            ˙Path 1: Go via $u$.
            $\mathtt{g}(v) \leftarrow \mathtt{g}(u) + \mathtt{c}(u, v)$
            $\mathtt{cameFrom}(v) \leftarrow u$
```

Table 1: Definitions for algorithms.

Inputs	
s_{Start}	Start vertex
s_{Goal}	Goal vertex
$\texttt{c}(u, v)$	Distance from u to v along their line-of-sight
$\texttt{h}(v)$	Heuristic estimate of cost to travel from v to s_{Goal}
Outputs	
$\texttt{g}(\cdot)$	Travelling costs from s_{Start} to u, for each u
$\texttt{cameFrom}(\cdot)$	Predecessor of u in the shortest path that was found from s_{Start} to u, for each u
Working variables	
open	Priority queue of vertices that are open
closed	Set of vertices that are closed
Methods for sets	
$\texttt{Add}(x)$	Store x
Methods for priority queues	
$\texttt{InsertWithPriority}(x, \mathit{priority})$	Store x with the specified *priority*
$\texttt{PopMinimumElement}()$	Retrieve the item that has the lowest priority and remove it from the queue
$\texttt{Remove}(x)$	Remove x from the queue
$\texttt{SmallestPriority}()$	Retrieve the value of the lowest priority
$\texttt{Enqueue}(x)$	Store x at the tail of the queue
$\texttt{Dequeue}()$	Retrieve the item at the head of the queue and remove it from the queue
Methods for geometry	
$\texttt{VerticesVisibleAnd8ConnectedTo}(u)$	Locations that are visible and 8-connected to u
$\texttt{LineOfSight}(p,\ v)$	True if and only if p has line-of-sight to v

2.2. Phi*

Now suppose that the terrain is dynamic: cells can change from being accessible to blocked (we leave aside the case of cells changing from blocked to accessible). The changes could invalidate the paths that were planned from the start to the goal. While we could replan the paths from afresh whenever the terrain changes, our intuition is that paths need only be replanned from the places where they changed.

Phi* extends Basic Theta* to handle dynamic terrain. Phi* is shown at Algorithm 4 and Figure 2 shows an example of Phi* in use. When framed inside Algorithm 5 for dynamic path planning, Phi* becomes Incremental Phi*. As Incremental Phi* only plans 'incrementally', it can be much faster than the naive approach of replanning afresh [Nash, Koenig & Likhachev 2009]. For this article, we take the framework as given and focus on Phi*.

The insight behind Phi* is that disruptions to grid paths are easily detected. Moreover by constraining its search, Phi* can use those disruptions to replan its vertex paths.

Algorithm 4: Phi*.

```
InitializeVertex($s$)
    g($s$) $\leftarrow \infty$
    cameFrom($s$) $\leftarrow \varnothing$
    localFrom($s$) $\leftarrow \varnothing$
    antic($s$) $\leftarrow -\infty$
    clock($s$) $\leftarrow \infty$
ComputeCost($v$, $u$)
    $p \leftarrow$ cameFrom($u$)
    $\phi \leftarrow$ AngleToFrom($\overrightarrow{pv}$, $\overrightarrow{pu}$)
    if $\phi \in$ [antic($u$), clock($u$)] and OffGrid($\overrightarrow{pv}$) and LineOfSight($p$, $v$) then
        if g($p$) + c($p, v$) < g($v$) then                          ·Path 2: Bypass u.
            g($v$) $\leftarrow$ g($p$) + c($p, v$)
            cameFrom($v$) $\leftarrow p$
            localFrom($v$) $\leftarrow u$
            $C \leftarrow$ Vertices4ConnectedTo($v$)
            $\alpha \leftarrow \min_{s \in C}$ AngleToFrom($\overrightarrow{ps}$, $\overrightarrow{pv}$)
            $\omega \leftarrow \max_{s \in C}$ AngleToFrom($\overrightarrow{ps}$, $\overrightarrow{pv}$)
            antic($v$) $\leftarrow \max(\alpha$, antic($u$) $- \phi)$
            clock($v$) $\leftarrow \min(\omega$, clock($u$) $- \phi)$
    else
        if g($u$) + c($u, v$) < g($v$) then                          ·Path 1: Go via u.
            g($v$) $\leftarrow$ g($u$) + c($u, v$)
            cameFrom($v$) $\leftarrow u$
            localFrom($v$) $\leftarrow u$
            antic($v$) $\leftarrow -45°$
            clock($v$) $\leftarrow 45°$
```

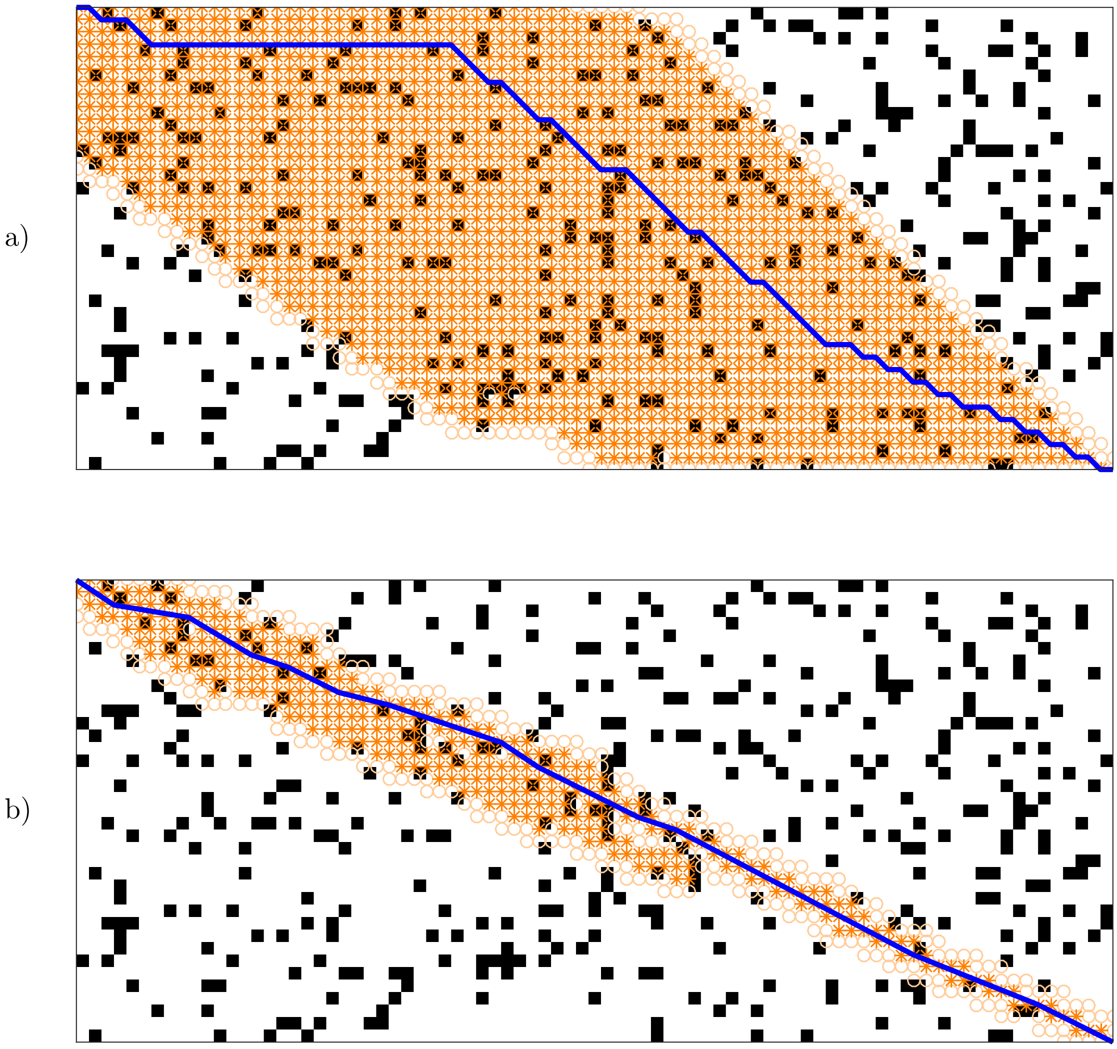

Figure 2: Pathfinding with Phi: The start is at the top-left corner, the goal is at the bottom-right corner, accessible cells are white, blocked cells are black. a) A shortest grid path found by A* searching the grid graph. b) A vertex path found by Phi*. The pictures show the path and algorithm end states where '*' marks a location that was closed and 'o' marks a location that was still open. The vertex path has less 'digitization bias' than the grid path.*

Algorithm 5: Dynamic path planning (Incremental Phi*). The presentation here uses recursion; see Nash et al. (2009) for the original formulation using while loops.

```
Main()
    Initialize()
    while true do
        ComputeShortestPath()
        Wait for cells to become blocked
        foreach newly blocked cell $c$ do
            foreach corner $s$ of $c$ do
                if ($s \in open$ or $s \in closed$) and $s \neq s_{\text{Start}}$ then
                    ClearSubtree($s$)

ClearSubtree($s$)
    $over \leftarrow \varnothing$                    · Vertices where g-values are over-estimated.
    ReinitializeNodes($s$, $over$)
    FinishExpansions($over$)

ReinitializeNodes($s$, $over$)
    $over$.Enqueue($s$)
    InitializeVertex($s$)
    if $s \in open$ then
        $open$.Remove($s$)
    if $s \in closed$ then
        $closed$.Remove($s$)
    foreach $v \in$ VerticesVisibleAnd8ConnectedTo($s$) do
        if localFrom($v$) $= s$ then
            ReinitializeNodes($v$, $over$)

FinishExpansions($over$)
    if $over \neq \varnothing$ then
        $v \leftarrow over$.Dequeue()
        foreach $u \in$ VerticesVisibleAnd8ConnectedTo($v$) do
            if $u \in closed$ then
                UpdateVertex($v$, $u$)
        FinishExpansions($over$)
```

2.2.1. Detecting Disruptions to Paths

Suppose that a cell in the terrain becomes blocked. To detect whether a vertex path has been disrupted, we have to ascertain whether the cell is on a line-of-sight between two points on that path. This test is likely to be expensive, so we wish to avoid it. But for grid paths, detecting a disruption is easy – we need only look at whether the corners of the cell are on the path. For by definition, if a grid path passes through a cell then at least one of the cell's corners is on that path.

In detail: Recall that at any given time, $s \in \mathit{closed}$ if A* has found a shortest path to s. Likewise $s \in \mathit{open}$ if it is adjacent to a vertex in *closed* (s is one step away from a closed vertex). Thus $s \in \mathit{open} \cup \mathit{closed}$ if and only if it is on a path from start to goal. Meanwhile for each s, $\mathtt{g}(s)$ is the distance from the start to s along the shortest known path. Now suppose that A* is applied to a grid graph. If cell c changes from/to being blocked then the paths that are affected can be identified by retrieving $s \in c \cap (\mathit{open} \cup \mathit{closed})$. We reset the computations for those locations (remove s from both *open* and *closed*, then reinitialize $\mathtt{g}(s)$) *and* for all locations on paths leading away from those locations.

2.2.2. Constraining the Search

Recall that Basic Theta* uses $\mathtt{cameFrom}(s)$ to store a vertex path to s. Phi* does the same, and then adds $\mathtt{localFrom}(s)$ to store a *local parent path* to s. The local parent path is constructed as a grid path (line 14, noting that v is adjacent to u in the grid graph).

Phi* constrains its search so that if a local parent path is disrupted then the resetting of computations (see above) will cause an appropriate replanning of the vertex paths. The constraint is enforced by *angle bounds* that restrict the directions taken by segments in the vertex paths, thereby keeping the vertex paths close to the associated local parent paths. Formally, let $\mathcal{C}(s)$ denote the cells traversed by the ray into s from its predecessor (the ray from $\mathtt{cameFrom}(s)$ to s). Then Phi* guarantees that for each cell in $\mathcal{C}(s)$, at least one corner of the cell is contained by the local parent path to s.

How the angle bounds are applied: Let p be the predecessor of u. If we are proposing to travel directly from u to v then Phi* requires that the angle from $\overrightarrow{pu}$ to $\overrightarrow{pv}$ is within bounds (it is contained by the interval $[\mathtt{antic}(u), \mathtt{clock}(u)]$). This *angle-in-bounds test* sits alongside the line-of-sight test that Basic Theta* imposed. The ray from p to v must also be *off the grid*, defined as not being a multiple of 45°. The off-grid test saves time that is otherwise wasted by line-of-sight tests [Nash, Koenig & Likhachev 2009]; in effect, we are checking that the ray from p to v is not a grid path. The tests use

$$\mathtt{AngleToFrom}(\vec{V},\, \vec{U}) \triangleq \arcsin\left(\frac{\vec{U}_x\vec{V}_y - \vec{V}_x\vec{U}_y}{\sqrt{\vec{U}_x^2 + \vec{U}_y^2}\sqrt{\vec{V}_x^2 + \vec{V}_y^2}}\right)$$

$$\mathtt{OffGrid}(\vec{V}) \triangleq \left|\vec{V}_x\right| \neq \left|\vec{V}_y\right| \neq 0$$

In our presentation of Phi* we test for angle-in-bounds, and then for off-grid, and then for line-of-sight. Doing so executes the tests in order of increasing cost (we will take this idea further in one of our refinements).

How the angle bounds are calculated: To complete Phi*, we must calculate the angle bounds for v (lines 15–19). The bounds must ultimately ensure that if a line segment is allowed from p to some location s then for each cell in $\mathcal{C}(s)$, at least one corner of the cell is contained by the local parent path to s. We present a new proof of correctness that will underpin one of the refinements to Phi* and that is perhaps of interest in its own right.

Suppose that we have reached line 15. Then $\overrightarrow{pv}$ is off the grid so we may choose a right-handed coordinate system with origin at p such that $0 < \frac{|v_y|}{v_x} < 1$. We have C as the vertices that are 4-connected to v, a so-called *crossbar* centred on v. Write $C = \{v^{(N)}, v^{(S)}, v^{(E)}, v^{(W)}\}$ for North, South, East and West of v in the assumed coordinate system (where North aligns with the positive y axis and East with the positive x axis). We first show that the calculations for α and ω need only consider $v^{(N)}$ and $v^{(S)}$.

Lemma 1. $\omega = \texttt{AngleToFrom}(\overrightarrow{pv^{(N)}}, \overrightarrow{pv})$ and $\alpha = \texttt{AngleToFrom}(\overrightarrow{pv^{(S)}}, \overrightarrow{pv})$.

Proof. We have that for any s

$$\texttt{AngleToFrom}(\overrightarrow{ps}, \overrightarrow{pv}) = \texttt{AngleToFrom}(\overrightarrow{ps}, x\ axis) - \texttt{AngleToFrom}(\overrightarrow{pv}, x\ axis)$$

So to calculate ω, it is sufficient to find $s \in C$ that maximizes $\texttt{AngleToFrom}(\overrightarrow{ps}, x\ axis)$. Indeed we need only consider the gradient of $\overrightarrow{ps}$, namely $\frac{s_y}{s_x}$. Now

$$\begin{aligned}
&\arg\max \left\{ \frac{v_y^{(N)}}{v_x^{(N)}}, \frac{v_y^{(S)}}{v_x^{(S)}}, \frac{v_y^{(E)}}{v_x^{(E)}}, \frac{v_y^{(W)}}{v_x^{(W)}} \right\} \\
&= \arg\max \left\{ \frac{v_y+1}{v_x}, \frac{v_y-1}{v_x}, \frac{v_y}{v_x+1}, \frac{v_y}{v_x-1} \right\} \\
&= \arg\max \left\{ \frac{v_y}{v_x} + \frac{1}{v_x}, \frac{v_y}{v_x} - \frac{1}{v_x}, \frac{v_y}{v_x}\left(\frac{v_x+1-1}{v_x+1}\right), \frac{v_y}{v_x}\left(\frac{v_x-1+1}{v_x-1}\right) \right\} \\
&= \arg\max \left\{ \frac{v_y}{v_x} + \frac{1}{v_x}, \frac{v_y}{v_x} - \frac{1}{v_x}, \frac{v_y}{v_x} - \frac{1}{v_x}\left(\frac{v_y}{v_x+1}\right), \frac{v_y}{v_x} + \frac{1}{v_x}\left(\frac{v_y}{v_x-1}\right) \right\}
\end{aligned}$$

Then consider the two cases for v_y:

- $v_y > 0$. We have $0 < \frac{|v_y|}{v_x} < 1$ so $v_y \leq v_x - 1$ and thus $\frac{v_y}{v_x-1} \leq 1$.
- $v_y < 0$. Again $0 < \frac{|v_y|}{v_x} < 1$ so $0 < -v_y < v_x < v_x + 1$ and thus $\frac{-v_y}{v_x+1} \leq 1$.

Either way $\frac{v_y^{(N)}}{v_x^{(N)}}$ has the largest gradient and we choose $v^{(N)}$ to calculate ω. The same argument retrieves α when seeking the minimum. $\square$

We now confirm that if a line segment emanates from p then it will pass through a cell that contains v.

Proposition 1 (The calculated angle bounds are correct)**.** Suppose that `antic`$(\cdot)$, `clock`$(\cdot)$ allow a ray from p to some location s. Let $\mathcal{C}(s)$ denote the cells traversed by the line segment from p to s. Then for each cell in $\mathcal{C}(s)$, at least one corner of the cell is contained by the local parent path to s.

Proof. Choose a coordinate system with origin at p such that $s_x > 0$. Let c be any cell in $\mathcal{C}(s)$. Then in our coordinate system, we can take c as having x coordinates from $x-1$ to x for some $0 < x \leq s_x$. The local parent path from p to s is 8-connected so there exists v in the local parent path to s having $v_x = x$. Now Phi* guarantees that $\overline{ps}$ intersects $\overline{v^{(S)}v^{(N)}}$. So c has either $\overline{v^{(S)}v}$ or $\overline{vv^{(N)}}$ as an edge, and consequently has v as a corner. □

3. Refinements Exploiting Geometry

We consider two refinements to Phi* that exploit the geometry of ϕ. We will need

$$\texttt{sign}(x) \triangleq \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \end{cases}$$

3.1. Bounds Known

While Phi* calculates α and ω as the minimum/maximum angle over four points, Lemma 1 provides those angles directly. We use this information in Bounds Known Phi*, as shown at Algorithm 6: `BoundingPoints` infers a right-handed coordinate system with origin at p that makes the gradient to v less than 1. Thus $v^{(N)}$ and $v^{(S)}$ can be located as per Lemma 1. We then use `sign`$(\cdot)$ to convert back to global coordinates.

3.2. Integer Operations Only

Integer Phi* reformulates the angle-in-bounds test to use integer operations only, as shown at Algorithm 7 (the calculations for distance are still floating point). Nash et al. (2009) postulated that doing so would reduce runtime. The key is to keep the vectors that underpin ϕ and test for vectors being anti/clockwise from each other using

$$\texttt{ClockFrom}(\vec{V}, \vec{U}) \triangleq \vec{U}_x\vec{V}_y - \vec{V}_x\vec{U}_y \geq 0$$

$$\texttt{AnticFrom}(\vec{V}, \vec{U}) \triangleq \vec{U}_x\vec{V}_y - \vec{V}_x\vec{U}_y \leq 0$$

Accordingly, we initialize `antic`$(\cdot)$, `clock`$(\cdot)$ as vectors for $-45°, 45°$. Note that $\left(\begin{smallmatrix} -y \\ x \end{smallmatrix}\right)$ is clockwise 90° from $\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$, so $\left(\begin{smallmatrix} x-y \\ y+x \end{smallmatrix}\right)$ is clockwise 45° from $\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$. Likewise for anticlockwise.

Algorithm 6: Bounds Known Phi*.

```
ComputeCost($v$, $u$)
    $p \leftarrow$ cameFrom($u$)
    $\phi \leftarrow$ AngleToFrom($\overrightarrow{pv}$, $\overrightarrow{pu}$)
    if $\phi \in$ [antic($u$), clock($u$)] and OffGrid($\overrightarrow{pv}$) and LineOfSight($p$, $v$) then
        if g($p$) + c($p, v$) < g($v$) then                              ˙Path 2: Bypass u.
            g($v$) $\leftarrow$ g($p$) + c($p, v$)
            cameFrom($v$) $\leftarrow p$
            localFrom($v$) $\leftarrow u$
            $[v^{(A)}, v^{(C)}] \leftarrow$ BoundingPoints($v$, $p$)
            $\alpha \leftarrow$ AngleToFrom($\overrightarrow{pv^{(A)}}$, $\overrightarrow{pv}$)
            $\omega \leftarrow$ AngleToFrom($\overrightarrow{pv^{(C)}}$, $\overrightarrow{pv}$)
            antic($v$) $\leftarrow$ max($\alpha$, antic($u$) $- \phi$)
            clock($v$) $\leftarrow$ min($\omega$, clock($u$) $- \phi$)
    else
        if g($u$) + c($u, v$) < g($v$) then                              ˙Path 1: Go via u.
            g($v$) $\leftarrow$ g($u$) + c($u, v$)
            cameFrom($v$) $\leftarrow u$
            localFrom($v$) $\leftarrow u$
            antic($v$) $\leftarrow -45^\circ$
            clock($v$) $\leftarrow \ \ 45^\circ$

BoundingPoints($v$, $p$)
    $\sigma_x \leftarrow$ sign($v_x - p_x$)
    $\sigma_y \leftarrow$ sign($v_y - p_y$)
    if $|v_y - p_y| < |v_x - p_x|$ then
        $v^{(A)} \leftarrow v - \left(\begin{smallmatrix}0\\ \sigma_x\end{smallmatrix}\right)$
        $v^{(C)} \leftarrow v + \left(\begin{smallmatrix}0\\ \sigma_x\end{smallmatrix}\right)$
    else if $|v_y - p_y| > |v_x - p_x|$ then
        $v^{(A)} \leftarrow v + \left(\begin{smallmatrix}\sigma_y\\ 0\end{smallmatrix}\right)$
        $v^{(C)} \leftarrow v - \left(\begin{smallmatrix}\sigma_y\\ 0\end{smallmatrix}\right)$
    else
        ˙OffGrid guarantees that we do not reach this point.
    return $[v^{(A)}, v^{(C)}]$
```

Algorithm 7: Integer Phi*.

```
ComputeCost($v$, $u$)
    $p \leftarrow$ cameFrom($u$)
    if ClockFrom($\overrightarrow{pv}$, antic($u$)) and AnticFrom($\overrightarrow{pv}$, clock($u$)) and OffGrid($\overrightarrow{pv}$) and
      LineOfSight($p$, $v$) then
        if $g(p) + c(p, v) < g(v)$ then                                   · Path 2: Bypass u.
            $g(v) \leftarrow g(p) + c(p, v)$
            cameFrom($v$) $\leftarrow p$
            localFrom($v$) $\leftarrow u$
            $[v^{(A)}, v^{(C)}] \leftarrow$ BoundingPoints($v$, $p$)
            $\alpha \leftarrow \overrightarrow{pv^{(A)}}$
            $\omega \leftarrow \overrightarrow{pv^{(C)}}$
            if ClockFrom($\alpha$, antic($u$)) then
                antic($v$) $\leftarrow \alpha$
            else
                antic($v$) $\leftarrow$ antic($u$)
            if AnticFrom($\omega$, clock($u$)) then
                clock($v$) $\leftarrow \omega$
            else
                clock($v$) $\leftarrow$ clock($u$)
    else
        if $g(u) + c(u, v) < g(v)$ then                                   · Path 1: Go via u.
            $g(v) \leftarrow g(u) + c(u, v)$
            cameFrom($v$) $\leftarrow u$
            localFrom($v$) $\leftarrow u$
            antic($v$) $\leftarrow \begin{pmatrix} x + y \\ y - x \end{pmatrix}$
            clock($v$) $\leftarrow \begin{pmatrix} x - y \\ y + x \end{pmatrix}$
```

4. Refinements Focussed on Line-of-Sight Testing

We look at three ways of reducing the number of line-of-sight tests or their cost. While they can be applied to any of the algorithms given above, we apply them to Bounds Known Phi* for definiteness and for reasons that arose during experiments (Section 5).

4.1. Expensive Testing Last

Expensive Last Phi* performs the shortening test before the line-of-sight test, as shown at Algorithm 8. The following result proves that we will get the same paths, faster.

Proposition 2. Under the same conditions, Algorithm 8 will execute the same code as Algorithm 3 but with potentially fewer tests for line-of-sight.

Proof of Proposition 2. In three steps:

1. *If* $\mathtt{g}(p) + \mathtt{c}(p, v) \geq \mathtt{g}(v)$ *then* $\mathtt{g}(u) + \mathtt{c}(u, v) \geq \mathtt{g}(v)$.

   Proof. $\mathtt{g}(u) + \mathtt{c}(u, v) = \mathtt{g}(p) + \mathtt{c}(p, u) + \mathtt{c}(u, v) \geq \mathtt{g}(p) + \mathtt{c}(p, v) \geq \mathtt{g}(v)$. ☐

2. *Under the same conditions, Algorithm 8 will execute the same code as Algorithm 6.*

   Proof. Lines 6–13 and 16–20 are the same in both algorithms. Then Table 2 shows that under the same conditions, the algorithms call the same lines. ☐

3. *Algorithm 8 can potentially make fewer tests for line-of-sight:*

   Proof. Line-of-sight testing only occurs if the other tests succeed. ☐

4.2. Lazy Evaluation

Lazy Phi* applies *lazy evaluation*. Nash et al. (2009) proposed lazy evaluation as an idea for future research into Phi* and subsequently developed it as an extension to Basic Theta* [Nash, Koenig & Tovey 2010]. Recall that in Bounds Known Phi*, `ComputeCost` tests for a path to v directly from p, bypassing u. The tests are applied immediately after v has been expanded from u. Lazy evaluation *assumes* that the direct path exists, and checks this assumption as late as possible – namely, just before u is expanded, as shown at Algorithm 9.

If the assumption was invalid then the path to u is repaired. The repairs infer a path to u by considering the vertices that are 8-connected to u and closed. In general there will exist at least one such vertex, namely the one(s) that opened u when they were expanded (the exception is when s_{Start} is being expanded).

Algorithm 8: Expensive Last Phi*.

```
1  ComputeCost($v$, $u$)
2      $p \leftarrow$ cameFrom($u$)
3      $\phi \leftarrow$ AngleToFrom($\overrightarrow{pv}$, $\overrightarrow{pu}$)
4      if $\phi \in$ [antic($u$), clock($u$)] and OffGrid($\overrightarrow{pv}$) and g($p$) + c($p, v$) < g($v$) and
        LineOfSight($p$, $v$) then                                    · Path 2: Bypass u.
5          · Blank line inserted to align with Bounds Known Phi*.
6
7
8
9          · Lines 6–13 are as for Bounds Known Phi*.
10
11
12
13
14         return
15     if g($u$) + c($u, v$) < g($v$) then                              · Path 1: Go via u.
16
17
18         · Lines 16–20 are as for Bounds Known Phi*.
19
20
```

Table 2: Logic tree for Algorithm 6 and Algorithm 8.

p has line-of-sight to v, ϕ is in bounds and $\overrightarrow{pv}$ is off grid	$\mathsf{g}(p) + \mathsf{c}(p, v) < \mathsf{g}(v)$	$\mathsf{g}(u) + \mathsf{c}(u, v) < \mathsf{g}(v)$	Lines executed
T	T	T	6–13
		F	6–13
	F	F	
F	T	T	16–20
		F	
	F	F	

Note: If $\mathsf{g}(p) + \mathsf{c}(p, v) \geq \mathsf{g}(v)$ then $\mathsf{g}(u) + \mathsf{c}(u, v) \geq \mathsf{g}(v)$ (see text).

After repairs, u may no longer be the vertex with the smallest priority. This motivates the Lazy-R variant in which u is only expanded if it is a vertex with the smallest priority.

4.3. Angle Propagation

Angle Propagating Phi* uses *angle propagation* to perform line-of-sight testing in constant time. Nash et al. (2007) introduced angle propagation to make Basic Theta* faster, then Nash et al. (2009) proposed that it be used in Phi*. Our development here is novel. The key is to use induction: suppose that a given line segment has line-of-sight. Then suppose that a new line segment has the same start as that line segment and is in the same direction 'up to suitable bounds'. To test for line-of-sight along the new segment, we only have to check the cells that are different. As 'suitable bounds', we use the anticlockwise and clockwise angles that we are already maintaining for Phi*.

Angle Propagating Phi* is shown at Algorithm 10: Define $\mathcal{V}(s,p)$ as the *last cell(s) traversed by* $\overrightarrow{ps}$ (if s and p have the same x coordinate or the same y coordinate then there are two such cells, otherwise there is one). Construct

$$\texttt{LastAccessible}(s,\ p) \triangleq \text{true if and only if } \mathcal{V}(s,p) \text{ is accessible}$$

$$\texttt{LastBlocked}(s,\ p) \triangleq \textbf{not } \texttt{LastAccessible}(s,\ p)$$

Then Angle Propagating Phi* diverges from Bounds Known Phi* as follows:

- *Tests that* $\mathcal{V}(v,p)$ *is accessible at line 4.* Where previous algorithms used `LineOfSight`, we now have `LastAccessible`. This implements the idea of only testing the new cells on a line-of-sight.

- *Modifies* `BoundingPoints` *(compare with original at Algorithm 6).* Recall the guarantee made by Bounds Known Phi*: if $\texttt{cameFrom}(v) = p$ for some location v then rays emanating from p will intersect $\overline{v^{(A)}v^{(C)}}$. Now for line-of-sight purposes, if we are to allow line segments to intersect $\overline{vv^{(A)}}$ then the cells abutting $\overline{vv^{(A)}}$ must both be accessible. If either cell is inaccessible then we must constrain the rays. The same holds for $\overline{vv^{(C)}}$.

- *New initial values for the angle bounds, replacing the original* ±45°. We are proposing to allow rays to emanate from u towards v. The intervening cell must be accessible if these rays are to be valid.

Algorithm 9: Lazy Phi*.

```
1  ComputeShortestPath()
2      while $open$.SmallestPriority() $< g(s_{Goal}) + h(s_{Goal})$ do
3          $u \leftarrow open$.PopMinimumElement()
4          $p \leftarrow$ cameFrom($u$)
5          $s \leftarrow$ localFrom($u$)
6          $\phi \leftarrow$ AngleToFrom($\overrightarrow{pu}$, $\overrightarrow{ps}$)
7          if $\phi \notin$ [antic($s$), clock($s$)] or not OffGrid($\overrightarrow{pu}$) or not LineOfSight($p$, $u$) then
8              RepairCameFrom($u$)
9              { continue }                                            ▷ Lazy-R.
10         $closed$.Add($u$)
11         ExpandVertex($u$)
12 ComputeCost($v$, $u$)
13     if $g(p) + c(p, v) < g(v)$ then                                 ▷ Path 2: Bypass u.
14
15
16
17         ▷ Lines 14–21 are as for Bounds Known Phi*, lines 6–13.
18
19
20
21
22 RepairCameFrom($u$)
23     if $closed = \varnothing$ then
24         $p \leftarrow s_{Start}$      ▷ $\phi$ is ill-defined when $s_{Start}$ is opened, causes a false alarm.
25     else
26         $C \leftarrow closed \cap$ VerticesVisibleAnd8ConnectedTo($u$)
27         $p \leftarrow \arg\min_{s \in C} g(s) + c(s, u)$                 ▷ Path 1: Infer p.
28     $g(u) \leftarrow g(p) + c(p, u)$
29     cameFrom($u$) $\leftarrow p$
30     localFrom($u$) $\leftarrow p$
31     antic($u$) $\leftarrow -45°$
32     clock($u$) $\leftarrow$ 45°
```

Algorithm 10: Angle Propagating Phi* (based on Expensive Last Phi*).

```
1  ComputeCost($v$, $u$)
2      $p \leftarrow$ cameFrom($u$)
3      $\phi \leftarrow$ AngleToFrom($\overrightarrow{pv}$, $\overrightarrow{pu}$)
4      if $\phi \in$ [antic($u$), clock($u$)] and OffGrid($\overrightarrow{pv}$) and g($p$) + c($p, v$) < g($v$) and
        LastAccessible($v$, $p$) then                                   ˙Path 2: Bypass u.
5          ˙Blank line inserted to align with Bounds Known Phi*.
6
7
8
9          ˙Lines 6–13 are as for Bounds Known Phi*.
10
11
12
13
14         return
15     if g($u$) + c($u, v$) < g($v$) then                                  ˙Path 1: Go via u.
16
17         ˙Lines 16–18 are as for Bounds Known Phi*.
18
19         [antic($v$), clock($v$)] $\leftarrow$ InitialAngleBounds($v$, $u$)
```

Algorithm 10: Angle Propagating Phi* (continued).

```
BoundingPoints($v$, $p$)
    $\sigma_x \leftarrow$ sign($v_x - p_x$)
    $\sigma_y \leftarrow$ sign($v_y - p_y$)
    if $|v_y - p_y| < |v_x - p_x|$ then
        $v^{(A)} \leftarrow v + \left(\begin{smallmatrix}0\\-\sigma_x\end{smallmatrix}\right)$
        $v^{(C)} \leftarrow v + \left(\begin{smallmatrix}0\\\sigma_x\end{smallmatrix}\right)$
        if $\sigma_x = \sigma_y$ then
            if LastBlocked($v + \left(\begin{smallmatrix}\sigma_x\\0\end{smallmatrix}\right)$, $p$) then
                $v_y^{(A)} \leftarrow v_y$
            if LastBlocked($v + \left(\begin{smallmatrix}0\\\sigma_x\end{smallmatrix}\right)$, $p$) or LastBlocked($v + \left(\begin{smallmatrix}\sigma_x\\\sigma_x\end{smallmatrix}\right)$, $p$) then
                $v_y^{(C)} \leftarrow v_y$
        else if $\sigma_x = -\sigma_y$ then
            if LastBlocked($v + \left(\begin{smallmatrix}\sigma_x\\0\end{smallmatrix}\right)$, $p$) then
                $v_y^{(C)} \leftarrow v_y$
            if LastBlocked($v + \left(\begin{smallmatrix}0\\-\sigma_x\end{smallmatrix}\right)$, $p$) or LastBlocked($v + \left(\begin{smallmatrix}\sigma_x\\-\sigma_x\end{smallmatrix}\right)$, $p$) then
                $v_y^{(A)} \leftarrow v_y$
    else if $|v_y - p_y| > |v_x - p_x|$ then
        $v^{(A)} \leftarrow v + \left(\begin{smallmatrix}\sigma_y\\0\end{smallmatrix}\right)$
        $v^{(C)} \leftarrow v + \left(\begin{smallmatrix}-\sigma_y\\0\end{smallmatrix}\right)$
        if $\sigma_x = \sigma_y$ then
            if LastBlocked($v + \left(\begin{smallmatrix}0\\\sigma_y\end{smallmatrix}\right)$, $p$) then
                $v_x^{(C)} \leftarrow v_x$
            if LastBlocked($v + \left(\begin{smallmatrix}\sigma_y\\0\end{smallmatrix}\right)$, $p$) or LastBlocked($v + \left(\begin{smallmatrix}\sigma_y\\\sigma_y\end{smallmatrix}\right)$, $p$) then
                $v_x^{(A)} \leftarrow v_x$
        else if $\sigma_x = -\sigma_y$ then
            if LastBlocked($v + \left(\begin{smallmatrix}0\\\sigma_y\end{smallmatrix}\right)$, $p$) then
                $v_x^{(A)} \leftarrow v_x$
            if LastBlocked($v + \left(\begin{smallmatrix}-\sigma_y\\0\end{smallmatrix}\right)$, $p$) or LastBlocked($v + \left(\begin{smallmatrix}-\sigma_y\\\sigma_y\end{smallmatrix}\right)$, $p$) then
                $v_x^{(C)} \leftarrow v_x$
    else
        OffGrid guarantees that we do not reach this point.
    return $[v^{(A)}, v^{(C)}]$
```

Algorithm 10: Angle Propagating Phi* (continued).

```
InitialAngleBounds($v$, $u$)
    $\sigma_x \leftarrow$ sign($v_x - u_x$)
    $\sigma_y \leftarrow$ sign($v_y - u_y$)
    if $\sigma_x = 0$ then
        if LastAccessible($u + \binom{\sigma_y}{\sigma_y}$, $u$) then
            antic $\leftarrow -45°$
        else
            antic $\leftarrow 0°$
        if LastAccessible($u + \binom{-\sigma_y}{\sigma_y}$, $u$) then
            clock $\leftarrow$  $45°$
        else
            clock $\leftarrow 0°$
    else if $\sigma_y = 0$ then
        if LastAccessible($u + \binom{\sigma_x}{-\sigma_x}$, $u$) then
            antic $\leftarrow -45°$
        else
            antic $\leftarrow 0°$
        if LastAccessible($u + \binom{\sigma_x}{\sigma_x}$, $u$) then
            clock $\leftarrow$  $45°$
        else
            clock $\leftarrow 0°$
    else
        if LastAccessible($v$, $u$) then
            antic $\leftarrow -45°$
            clock $\leftarrow$  $45°$
        else
            antic $\leftarrow 0°$
            clock $\leftarrow 0°$
    return [antic, clock]
```

5. Experiments

The refinements were tested on the following set of problems: the terrain was N cells × N cells where $N = 100, 200, 300, 400, 500$. A set fraction of cells was blocked, and then the cells' locations were randomized by allowing the permutations of cells to have equal probability. The start location was one of the terrain's four corners, and the goal location was chosen from the opposing edges. Each algorithm was applied to 50 sample problems. The algorithms were implemented in MATLAB using vector storage of the vertices, so retrieval of the most-promising vertex (`PopMinimumElement()`) occurs in linear time. If binary heap storage were used then retrieval would be in log time, but we note that at any given time, the number of open vertices is quite small (on the order of 100 or so) hence log times and linear times are approximately equal. Runtimes were measured as elapsed times using `timeit` following recommended practices [McKeeman 2016].

As the algorithms all yielded paths that were the same length as the original Phi*, we concentrate on the reductions in runtime: Figures 3–7 plot 95 percent confidence intervals for the mean reductions in runtime (bootstrapped from 2000 draws). Figures 8–12 plot the reductions in the vertices that are touched vs the reductions in runtime where a vertex is *touched* if it is opened or closed when an algorithm executes. We investigate how the runtime is affected by vertices being touched by calculating the Pearson correlation coefficient R.

Bounds Known did not demonstrate a reduction in runtime. We speculate that in our implementation, calculating two angles is not much cheaper than calculating four angles and finding their minimum and maximum. Indeed MATLAB can calculate multiple angles in a single call, and `max` and `min` are fast, built-in functions.

The results for Integer are very surprising: the runtime increased, and Integer touched a different number of vertices than the original Phi*. The cause was traced to floating point errors in Phi*, with Phi* classifying vertices as out-of-bounds when Integer had them in-bounds (or vice versa). As the reductions in runtime were moderately-well correlated with reductions in vertices being touched (R ranging from 0.75 through 0.88), we conclude that it is the change in vertices being touched that dominates the change in runtime.

Expensive Last demonstrated a small reduction in runtime, roughly 5 percent. The reductions decreased as blockages increased. This decrease is to be expected as more blockages leads to quicker line-of-sight testing and hence smaller savings from avoiding those tests. Likewise the reductions decreased as the terrain size increased. This decrease is consistent with line-of-sight testing being conducted between points that were further apart.

Lazy showed a 10 percent reduction in runtime on small terrains but the runtimes became bigger as the terrain became larger. Runtime also increased as the blockages increased. If the blockages are low then runtime is correlated with vertices being touched; conversely when blockages are high then there is no change to the vertices being touched (but poorer runtime anyway). We speculate to the cause as follows: for the original Basic Theta* to Lazy Theta*, it was safe to assume that direct paths exist and that repairs are the exception. But Phi* imposes angle bounds, making repairs more likely. Lazy-R had results that were qualitatively the same as Lazy but with poorer runtimes.

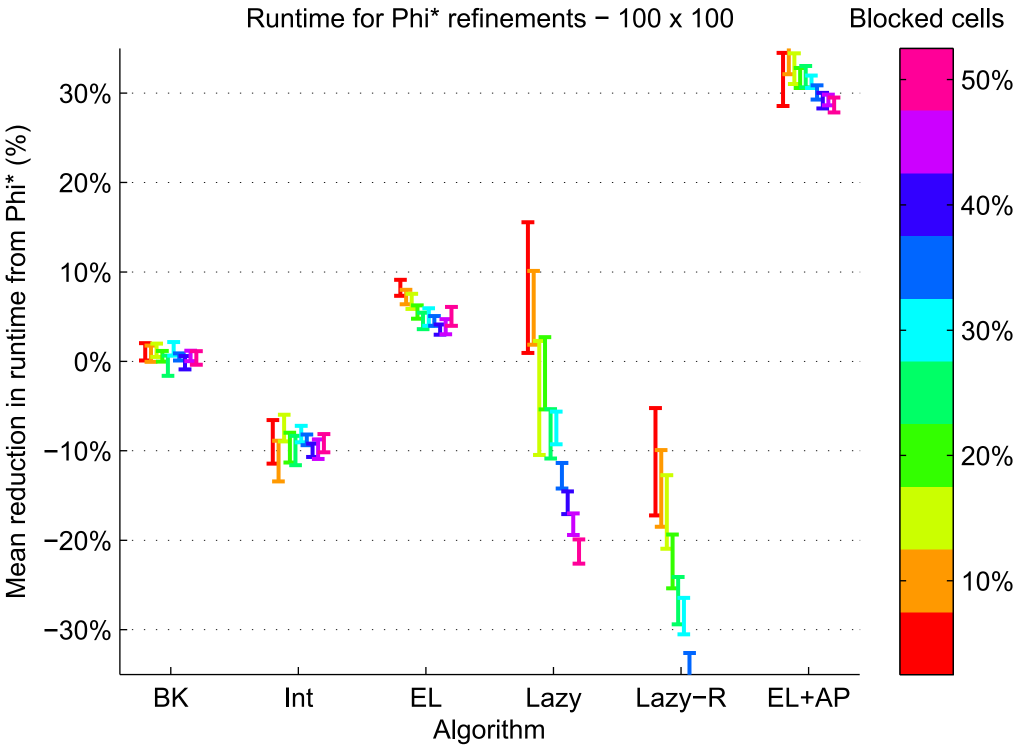

Figure 3: Runtimes for each algorithm (100 cells × 100 cells): Bounds Known (BK), Integer (Int), Expensive Last (EL), Lazy, Lazy-R and Expensive Last with Angle Propagation (EL+AP). The chart shows a 95 percent bootstrap confidence interval for the mean reduction in runtime from Phi. Colours denote the fraction of cells that were blocked.*

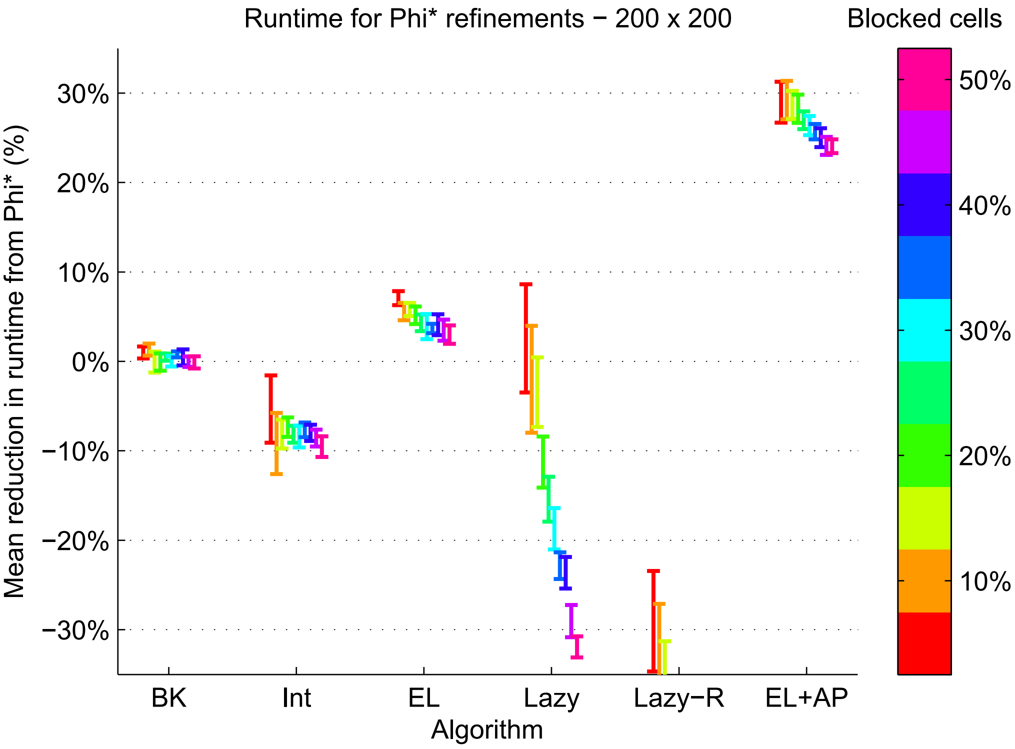

Figure 4: Runtimes for each algorithm (200 cells × 200 cells): Bounds Known (BK), Integer (Int), Expensive Last (EL), Lazy, Lazy-R and Expensive Last with Angle Propagation (EL+AP). The chart shows a 95 percent bootstrap confidence interval for the mean reduction in runtime from Phi. Colours denote the fraction of cells that were blocked.*

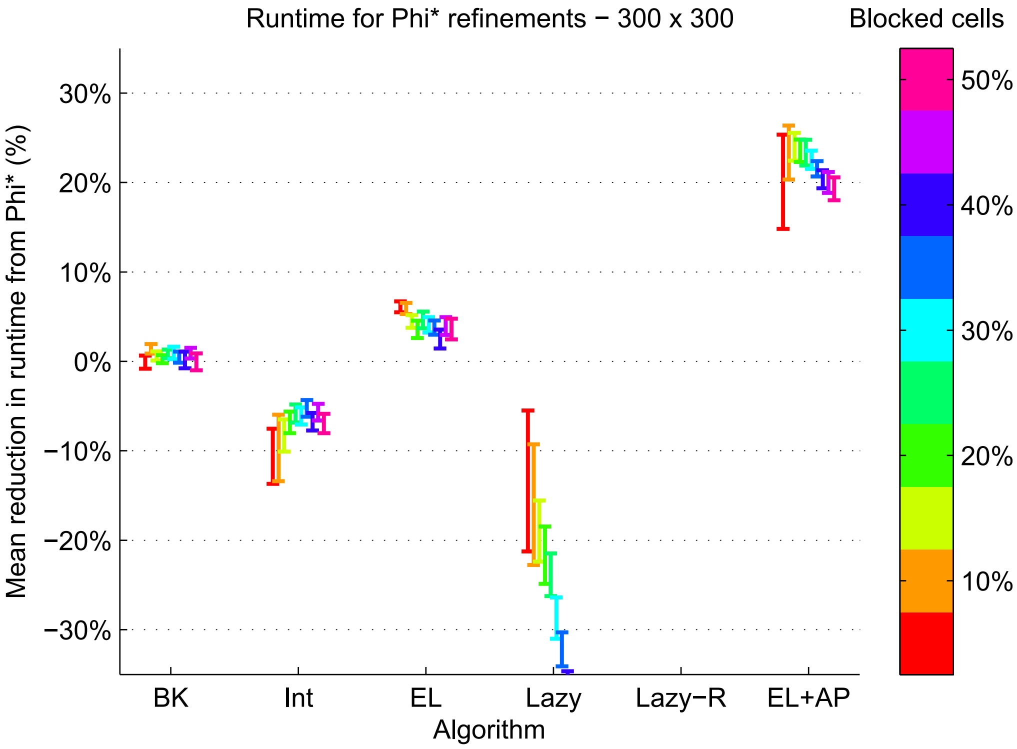

Figure 5: Runtimes for each algorithm (300 cells × 300 cells): Bounds Known (BK), Integer (Int), Expensive Last (EL), Lazy, Lazy-R and Expensive Last with Angle Propagation (EL+AP). The chart shows a 95 percent bootstrap confidence interval for the mean reduction in runtime from Phi. Colours denote the fraction of cells that were blocked.*

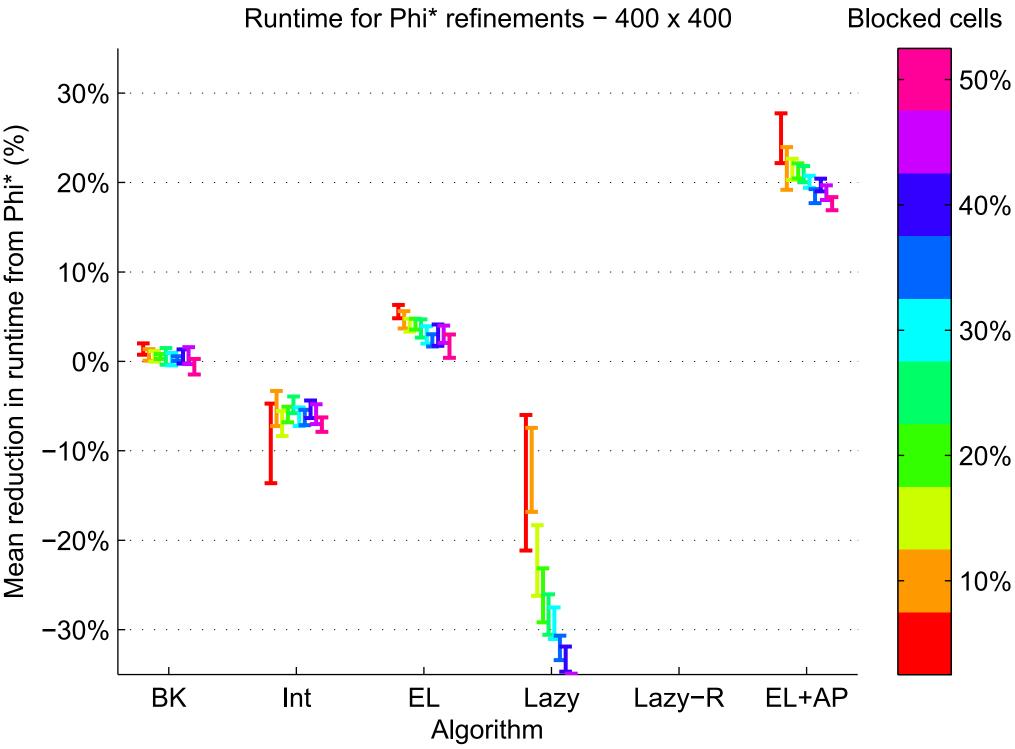

Figure 6: Runtimes for each algorithm (400 cells × 400 cells): Bounds Known (BK), Integer (Int), Expensive Last (EL), Lazy, Lazy-R and Expensive Last with Angle Propagation (EL+AP). The chart shows a 95 percent bootstrap confidence interval for the mean reduction in runtime from Phi. Colours denote the fraction of cells that were blocked.*

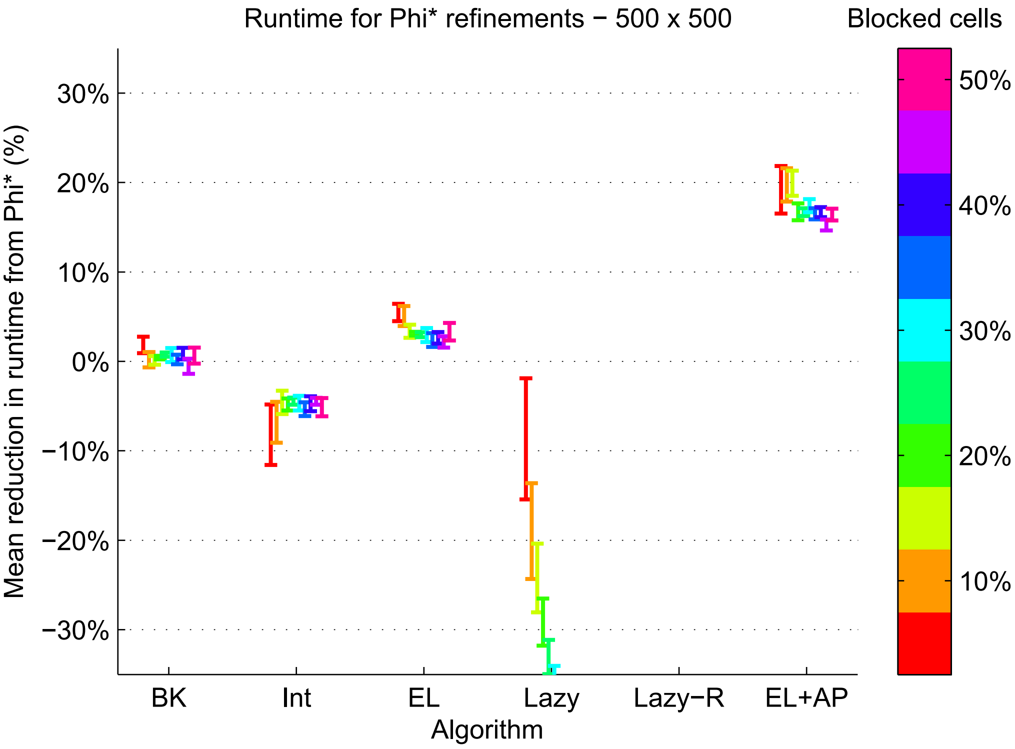

Figure 7: Runtimes for each algorithm (500 cells × 500 cells): Bounds Known (BK), Integer (Int), Expensive Last (EL), Lazy, Lazy-R and Expensive Last with Angle Propagation (EL+AP). The chart shows a 95 percent bootstrap confidence interval for the mean reduction in runtime from Phi. Colours denote the fraction of cells that were blocked.*

Figure 8: Runtimes vs Vertices Touched for each algorithm (100 cells × 100 cells. Colours denote the fraction of cells that were blocked.

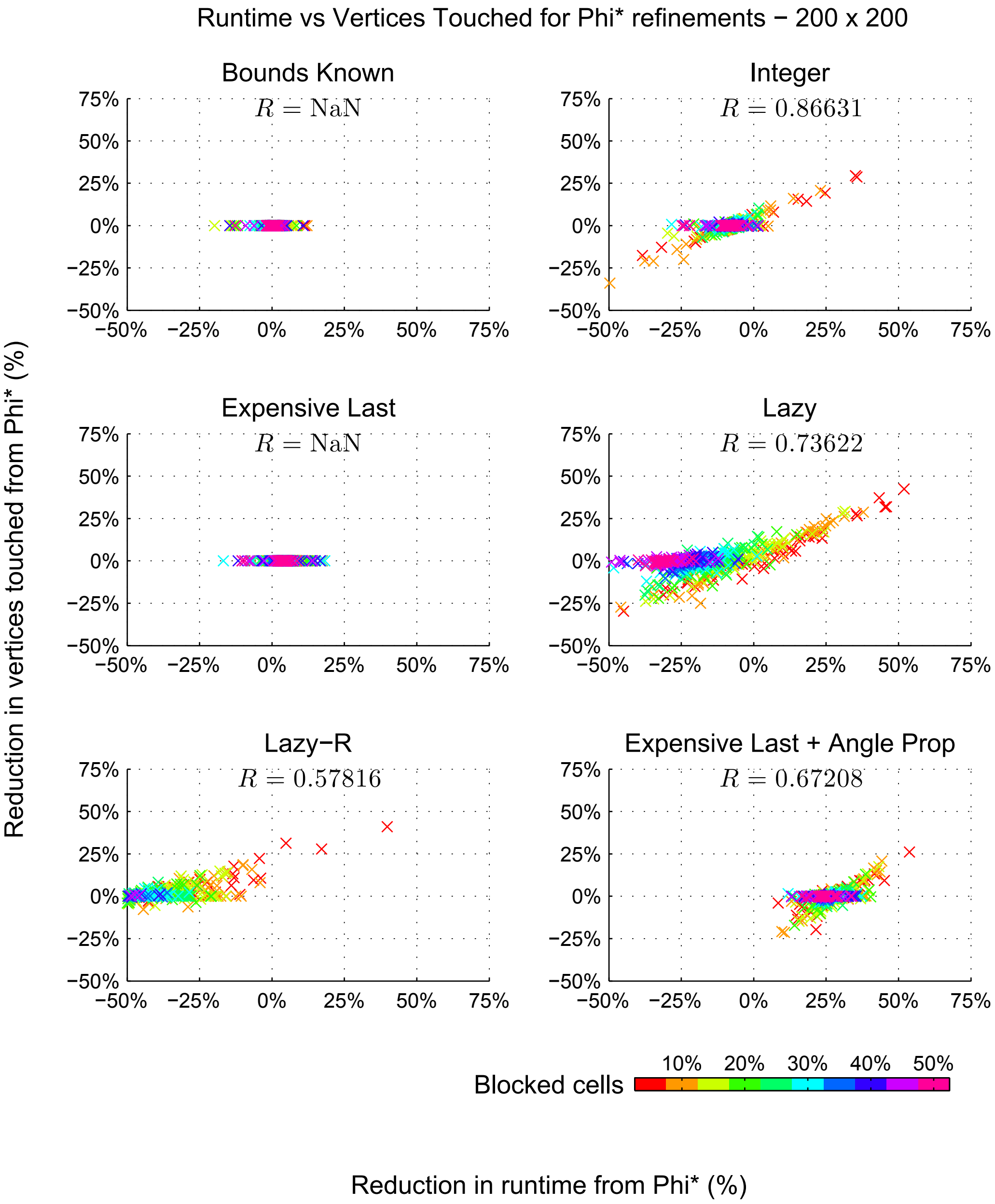

Figure 9: Runtimes vs Vertices Touched for each algorithm (200 cells × 200 cells. Colours denote the fraction of cells that were blocked.

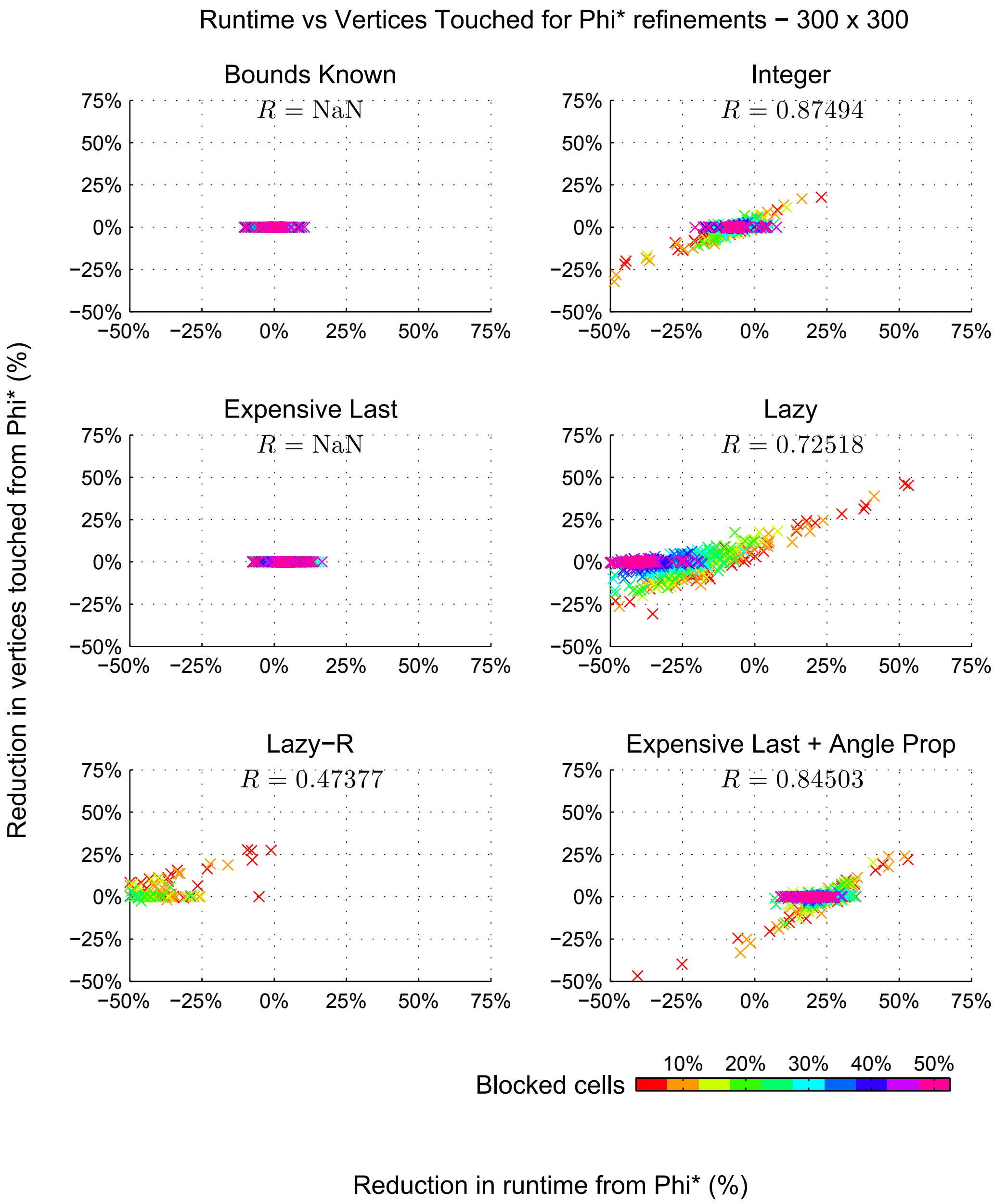

Figure 10: Runtimes vs Vertices Touched for each algorithm (300 cells × 300 cells. Colours denote the fraction of cells that were blocked.

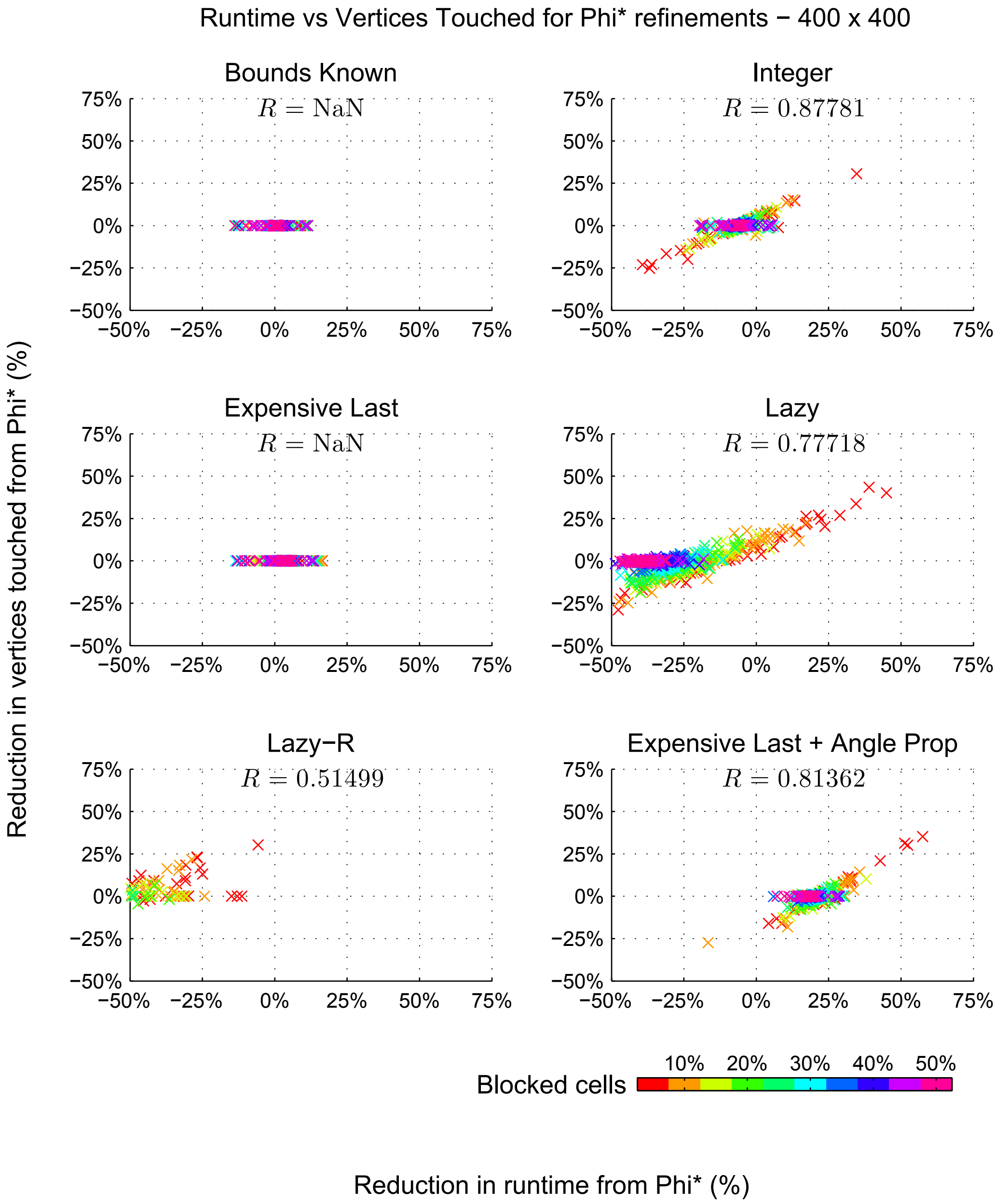

Figure 11: Runtimes vs Vertices Touched for each algorithm (400 cells × 400 cells. Colours denote the fraction of cells that were blocked.

Figure 12: Runtimes vs Vertices Touched for each algorithm (500 cells × 500 cells. Colours denote the fraction of cells that were blocked.

Angle Propagation (with Expensive Last) performed best overall: it showed a 30 percent reduction in runtime on 100 × 100 terrains, reducing to 15 percent on 500 × 500 terrains. The reduction can be accounted for as follows: Angle Propagation has a consistent reduction in runtime over Expensive Last. Then for terrains with few blockages, there can be substantial variation in the vertices that are touched, with a corresponding change to runtime.

6. Conclusion

Phi* can be sped up by Expensive Last evaluation with Angle Propagation. The resulting algorithm performs its line-of-sight testing in constant time, and demonstrated a 15–30 percent reduction in runtime over the original Phi* (in MATLAB). The runtime appeared to be dominated by the number of vertices that were touched during execution (that is, the total number of vertices that were opened over the entire runtime, *not* the number that were open at any given time). The Integer variant otherwise retains interest, in avoiding floating point errors that could cause paths to be over-constrained or under-constrained.

7. Acknowledgments

The author appreciates the feedback and improvements from Jez Gray and Jijoong Kim.

8. References

McKeeman, B. (2016) MATLAB Performance Measurement, http://au.mathworks.com/matlabcentral/fileexchange/18510-matlab-performance-measurement. (Accessed 2017-02-21).

Nash, A., Daniel, K., Koenig, S. & Felner, A. (2007) Theta*: Any-angle path planning on grids, *in Proceedings of the National Conference on Artificial Intelligence*, Vol. 22, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press, pp. 1177–1183.

Nash, A., Koenig, S. & Likhachev, M. (2009) Incremental Phi*: Incremental Any-Angle Path Planning on Grids, *in Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1824–1830.

Nash, A., Koenig, S. & Tovey, C. (2010) Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D, *in Proceedings of the AAAI Conference on Artificial Intelligence AAAI.*

Tsitsiklis, J. (1995) Efficient algorithms for globally optimal trajectories, *Automatic Control, IEEE Transactions on* **40**(9), 1528–1538.

This page is intentionally blank

DEFENCE SCIENCE AND TECHNOLOGY GROUP DOCUMENT CONTROL DATA	1. DLM/CAVEAT (OF DOCUMENT)
2. TITLE Speeding Up Phi*: Refined Foundations for Dynamic Path Planning at Any Angle	3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)
4. AUTHORS Patrick Chisan Hew	5. CORPORATE AUTHOR Defence Science and Technology Group 506 Lorimer St, Fishermans Bend, Victoria 3207, Australia

6a. DST GROUP NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DST-Group–TN–1805	AR-017-268	Technical Note	August, 2018

8. OBJECTIVE ID	9. TASK NUMBER	10. TASK SPONSOR
qAV22839	NAV 17/525	Director General SEA1000

11. MSTC	12. STC
Maritime Capability Analysis	Maritime Systems Analysis
13. DOWNGRADING/DELIMITING INSTRUCTIONS	14. RELEASE AUTHORITY
(nil)	Chief, Joint and Operations Analysis Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. RESEARCH LIBRARY THESAURUS

trajectory optimisation, algorithms, autonomous navigation

19. ABSTRACT

‘Any-angled’ planning has emerged as a promising approach to finding short paths through a terrain that has obstacles. Phi* (2009) is the foundation of Incremental Phi* which handles terrains where obstacles are not known ahead of time and therefore have to be discovered during the path planning process (dynamic single pair shortest path). This technical note explores the following refinements to speed up Phi*: Bounds Known, Integer operations only, Expensive Last testing, Lazy evaluation and Angle Propagation. Expensive Last reduces runtime by roughly 5 percent. Expensive Last with Angle Propagation performs line-of-sight testing in constant time and reduces runtime by roughly 15–30 percent. Integer avoids floating point errors that could compromise the paths found by Phi*. The findings will be useful to developers and users of dynamic path planning algorithms in two-dimensional terrains.