Journal of Artificial Intelligence Research xx (2015) xx Submitted 10/2015; published xx/2015

Optimal Any-Angle Pathfinding In Practice

Daniel Harabor DANIEL.HARABOR@NICTA.COM.AU
National ICT Australia, Victoria Laboratory, Australia

Alban Grastien ALBAN.GRASTIEN@NICTA.COM.AU
National ICT Australia, Canberra Laboratory, Australia

Dindar Öz DINDAR.OZ@YASAR.EDU.TR
Yasar University, Bornova, Izmir, 35100 Turkey

Vural Aksakalli AKSAKALLI@SEHIR.EDU.TR
Istanbul Sehir University, Altunizade, Istanbul, 34662 Turkey

Abstract

Any-angle pathfinding is a fundamental problem in robotics and computer games. The goal is to find a shortest path between a pair of points on a grid map such that the path is not artificially constrained to the points of the grid. Prior research has focused on approximate online solutions. A number of exact methods exist but they all require supra-linear space and pre-processing time. In this study, we describe Anya: a new and optimal any-angle pathfinding algorithm. Where other works find approximate any-angle paths by searching over individual points from the grid, Anya finds exact paths by searching over sets of states represented as intervals. Each interval is identified on-the-fly. From each interval Anya selects a single representative point which it uses to compute an admissible cost estimate for the entire set. Anya always returns an optimal path if one exists. Moreover it does so without any offline pre-processing or the introduction of additional memory overheads. In a range of empirical comparisons we show that Anya is competitive with a range of recent sub-optimal and pre-processing based techniques and is up to an order of magnitude faster than the most common benchmark algorithm, a grid-based implementation of A*.

1. Introduction

Any-angle pathfinding is a commonly encountered navigation problem in robotics and computer games. It takes as input a pair of points from a uniform two-dimensional grid and asks for a shortest path between them which is not artificially constrained to the points of the grid. Such any-angle paths are desirable to compute as they are typically shorter than their grid-constrained counterparts and because following such a trajectory can give the appearance of realism and intelligence; e.g. to the player of a computer game. Despite its apparent simplicity any-angle pathfinding is surprisingly challenging. So far many successful and popular methods have been proposed, yet they all involve trade-offs of some kind. We begin with a few examples that highlight, in broad strokes, the main research trends and their limitations, to date.

In the communities of Artificial Intelligence and Game Development the any-angle pathfinding problem is often solved efficiently using a technique known as *string pulling*. The idea is to compute a grid-optimal path in the first instance and smooth the result; either as part of a post-processing step (e.g. (Pinter, 2001; Botea, Müller, & Schaeffer, 2004)) or by interleaving string pulling with online search (e.g. (Ferguson & Stentz, 2005; Nash, Daniel, Koenig, & Felner, 2007)). Regardless of the particular approach, all string pulling techniques suffer from the same disadvantages: (i) they require more computation than just finding a path and; (ii) they only yield approximately shortest paths.

In the communities of Robotics and Computational Geometry a related and more general problem exists that has been well-studied: finding Euclidean shortest paths between polygonal obstacles in the plane. *Visibility Graphs* (Lozano-Pérez & Wesley, 1979) and the *Continuous Dijkstra* paradigm (Mitchell, Mount, & Papadimitriou, 1987) are among the best known and most influential techniques that originate from this line of research. Even though both of these methods are optimal and efficient in practice they nevertheless suffer from having often undesirable properties: (i) the search graph[1] must be pre-computed during an offline pre-processing step; (ii) if the map changes at any point the search graph is invalidated and must be recomputed, usually from scratch.

To date, it is not clear if there exists an any-angle pathfinding algorithm that is simultaneously online, optimal and also practically efficient (i.e. at least as fast in practice as grid-based pathfinding using A* search). In this manuscript, we present new work which answers this open question in the affirmative by introducing a new any-angle pathfinding algorithm called **Anya**. Our approach bears some similarity to existing works from the literature, most notably those algorithms based on the Continuous Dijkstra paradigm. In rough overview:

- Where other methods search over the individual nodes of the grid, **Anya** searches over contiguous sets of states that form intervals.
- Each **Anya** interval has a single representative point that is used to derive an admissible cost estimate (i.e f-value) for all points in the set.
- To progress the search process **Anya** projects each interval, from one row of the grid onto another, until the target is reached.

Anya always finds an optimal any-angle path, if one exists. In addition **Anya** does not rely on any pre-computation nor does it introduce any memory overheads (beyond what is required to maintain an open and closed list). A theoretical description of this algorithm has previously appeared in the literature (Harabor & Grastien, 2013). In this study we extend that work in several ways: (i) we give a detailed conceptual description of the **Anya** algorithm and provide an extended theoretical argument for optimality and completeness; (ii) we discuss the practical considerations that arise when implementing the algorithm and we give a technical description for one possible and efficient implementation; (iii) we make detailed empirical comparisons showing that **Anya** is competitive with a range of recent sub-optimal techniques from the literature, including those based on offline pre-processing, and is up to one order of magnitude better than our benchmark grid-based implementation of A*; (iv) we discuss a range of possible extensions for further improving the current results.

1. As distinct from the grid map, which is given as an input.

2. The Optimal Any-Angle Pathfinding Problem

A *grid* is a planar subdivision consisting of $W \times H$ square cells. Each cell is an open set of *interior* points which are all *traversable* or all *non-traversable*. The vertices associated with each cell are called the *discrete* points of the grid. Edges in the grid can be interpreted as open intervals of *intermediate* points; each one representing a transition between two discrete points. Each type of point $p = (x, y)$ has a unique coordinate where $x \in [0, W]$ and $y = [0, H]$, with discrete points limited to the subset of integer x and y values.

A discrete or intermediate point is traversable if it is adjacent to at least one traversable cell. Otherwise it is non-traversable. A discrete point which is common to exactly four adjacent cells is called an *intersection*. Any intersection where three of the adjacent cells are traversable and one is not is called a *corner*. Two points are *visible* from one another if they can be connected by a straight-line path (i.e. a sequence of adjacent points, either intermediate or discrete) that does not: (i) pass through any non-traversable point or (ii) pass through an intersection formed by two diagonally-adjacent non-traversable cells.

An *any-angle path* π is a sequence of points $\langle p_1, \ldots, p_k \rangle$ where each p_i is visible from p_{i-1} and p_{i+1}. The *length* of π is the cumulative distance between every successive pair of points $d(p_1, p_2) + \ldots + d(p_{k-1}, p_k)$. The function $d(p = (x, y), p' = (x', y')) = \sqrt{(x - x')^2 + (y - y')^2}$ is a uniform Euclidean distance metric. We will say $p_i \in \pi$ is a *turning point* if the segments (p_{i-1}, p_i) and (p_i, p_{i+1}) form an angle not equal to 180° [2]. Finally, the *any-angle pathfinding problem* is one that requires as input a pair of discrete points, s and t, and asks for an any-angle path connecting them. The point s designates the source (equivalently, start) location while the point t designates the target (equivalently, goal) location. Such a path is *optimal* if there exists no alternative any-angle path between s and t which is strictly shorter.

Figure 1 provides an example of an optimal any-angle pathfinding problem. As can be seen the source, target and all obstacles have discrete positions however the path itself does not need to follow the grid. Notice also that the trajectory of this path appears much more realistic than any alternative which is restricted to turning at modulo 45 deg or 90 deg.

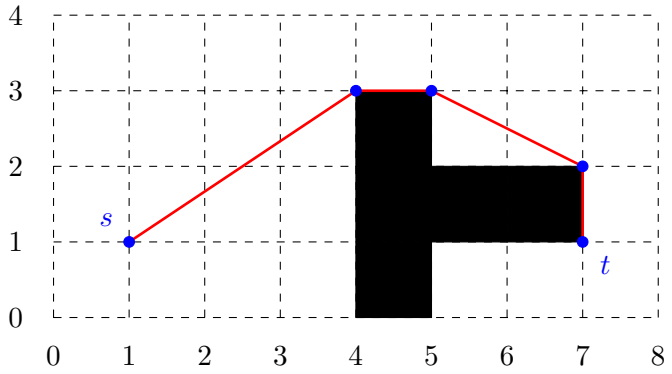

Figure 1: Example of an any-angle pathfinding problem together with its solution.

2. It is well-known that the turning points in optimal any-angle paths are corner points; see e.g. (Mitchell et al., 1987)

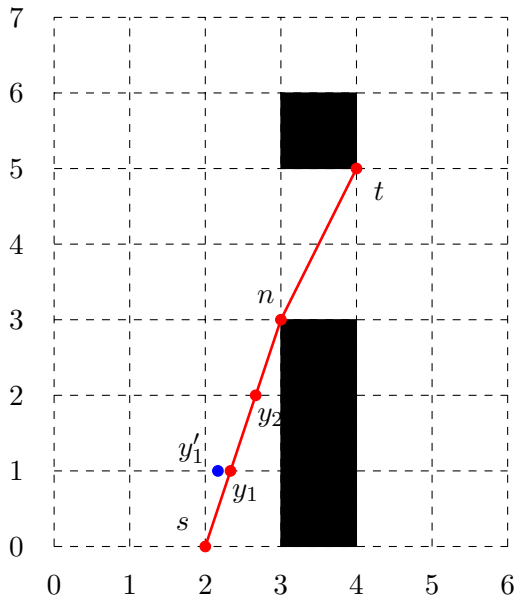

Figure 2: When pathfinding from s to n, online algorithms such as A* and Theta* only expand discrete points from the grid and never intermediate points such as y_i.

3. An Overview of Anya

Consider the any-angle instance shown in Figure 2. In this example the optimal path between s and t needs to first head towards the corner point n and then change direction toward the target t. One possible approach to solving this problem involves computing a visibility graph: i.e., identifying all pairs of corners that are visible from one another, and also visible from the start and target locations, and then searching for a path on this graph. The main drawback in this case is that the visibility graph can be quite large (up to quadratic in the size of the grid) and very expensive to compute.

An alternative approach, which avoids these overheads, is to solve the problem online. Unfortunately online search methods generally consider only the discrete points of the grid and their immediate neighbours. For example, when expanding the point s it is common to only generate the neighbours: $(1, 0)$, $(2, 1)$, and $(3, 0)$ in the example of Figure 2. The A* f-value for each of the three neighbours is, respectively, $1 + \sqrt{34} \simeq 6.83$, $1 + \sqrt{20} \simeq 5.47$, and $1 + \sqrt{26} \simeq 6.1$ (using Euclidean-distance as a heuristic). By comparison the optimal any-angle path has cost of $\sqrt{10} + \sqrt{5} \simeq 5.4$. Immediately we can see that the heuristic at hand does not satisfy one of the essential properties of A* search: that the f-value of each node should always be an underestimate of the actual distance to the goal. Without this property A* is not guaranteed to be optimal.

The issue described above comes from the fact that the optimal path does not go through any of the points $(1, 0)$, $(2, 1)$, or $(3, 0)$. Instead the optimal path crosses row 1 at point y_1, which is not part of the search space. To ensure optimality we should consider all points such as y_1 rather than just the discrete points of the grid. There are however many such

points including e.g., points such as y_1' (leading to $(3, 6)$), which apriori seems a reasonable candidate for expansion, but which does not appear on any optimal path.

In general we need to consider all potential y_i points defined as a fraction $\frac{h}{w}$ where $h \in \{0, \ldots, H\}$ and $w \in \{1, \ldots, W\}$. This is a set whose members are reducible to a Farey Sequence. For any given n (in our case $n = \min(W, H)$) the cardinality of the corresponding set of elements is known to be quadratic in n (Graham, Knuth, & Patashnik, 1989)(Ch. 9).

For this reason we propose to consider, instead of individual points, a set of points that appear together as part of a contiguous interval on the grid. In the example of Figure 2 we would consider all the points lying between $(0, 1)$ and $(3, 1)$, at the same time and as part of a single A* search node. In this framework we need:

- to define formally an **Anya** search node,
- to define the set of successors of a search node,
- to define how to compute the f-value of a search node,
- to prove optimality of the returned path,
- to terminate search when no path is available,
- to ensure the **Anya** algorithm is efficient in practice.

4. Algorithm Description

This section presents in detail the **Anya** algorithm and its properties. Since **Anya** is a variant of A* we first present its search space: the search nodes, the successors of a node and the evaluation function used to rank nodes during search. We then a give pseudo-code description of the algorithm and discuss its properties. Improvements that make **Anya** efficient in practice are presented in the next section.

4.1 Anya Search Nodes

We now define the notion of interval, which is at the core of **Anya**.

Definition 1 *A* grid interval I *is a set of contiguous and pairwise visible points drawn from any discrete row of the grid. Each interval is defined in terms of its endpoints* a *and* b*. With the possible exception of* a *and* b*, each interval contains only intermediate and discrete non-corner points.*

By definition, all points in an interval share the same y position, which is a positive integer. Moreover, the x position of all points in an interval (including that of endpoints a and b) is a rational number[3]. We will use normal parentheses "(" or ")" to indicate an interval endpoint that is open and square brackets "[" or "]" to indicate an interval endpoint that is closed. For example, the interval $I = (a, b]$ is open at (i.e. does not include) a and closed at (i.e. does include) b.

3. As per the problem definition, every point (x, y) appearing on an optimal any-angle path belongs to a Farey Sequence and all such points are rational.

Identifying intervals is simple: any row of the grid can be naturally divided into maximally contiguous sets of traversable and non-traversable points. Each traversable set forms a tentative interval which we can split, repeatedly if necessary, until all corner points are end points of intervals. Intervals can also be identified through an operation called projection. We discuss this procedure in the next sub-section. For now we note only that intervals produced by way of projection can also have non-discrete and non-corner endpoints.

A significant advantage of **Anya** is that we construct intervals on-the-fly. This allows us to start answering queries immediately and for any discrete start-target pair. Similar algorithms (e.g. Continuous Dijkstra (Mitchell et al., 1987)) require a pre-processing step before any queries can be answered and then only from a single fixed start point.

Definition 2 *A* search node (I, r) *is a tuple where* $r \notin I$ *is a point called the* root *and* I *is an interval defined such that each point* $p \in I$ *is visible from* r*. The identity of* r *is always the most recent turning point on an any-angle path, from the start point* s *to any* $p \in I$*. To represent the start node itself, set* $I = [s]$ *and assume* r *is located off the plane and visible only from* s*; the cost from* r *to* s *in this case is zero.*

Besides the start node, which we treat as a special case, there are other two types of search nodes: cone nodes and flat nodes. An example of a cone node is shown in Figure 3. Such nodes are characterised by the fact that the root r is not on the same row as its associated interval I. Notice in the example that although the interval $I = [a, b]$ is maximal, it does not have any endpoints which are obstacles, corners or indeed even discrete points of the grid (here the left endpoint a is $(2.5, 4)$ while the right endpoint b is $(5.5, 4)$). Examples of flat nodes are shown in Figure 4. The two nodes are: $((a_1, b_1], r)$ and $((a_2, b_2], r)$. Flat nodes are characterised by the fact that the root r is on the same row as the interval I. Notice in the examples given that $a_1 = r$ (resp. $a_2 = b_1$) is excluded from the first (resp. second) interval. The semantics of every search node is that the current position is located somewhere in the interval I and we reach that point by an any-angle path whose most recent turning point is r.

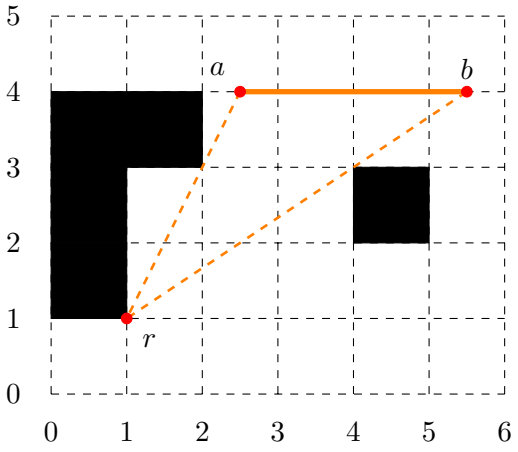

Figure 3: Example of a cone search node.

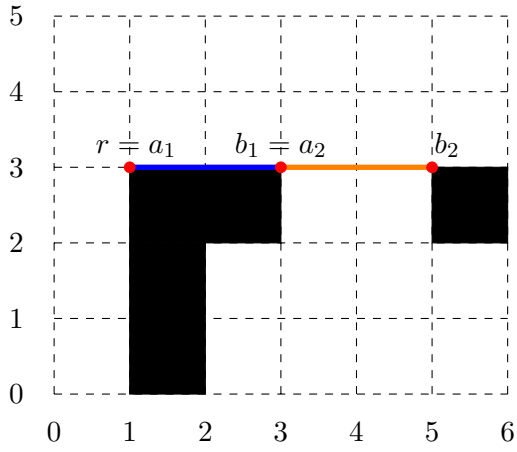

Figure 4: Example of two flat search nodes.

4.2 Searching with Anya: Successors

The successors of a search node n are identified by computing intervals over sets of traversable points; from the same row of the grid as the current node n and from the rows immediately adjacent. We want to guarantee that each point in such a set can be reached from the root of n via a local path which is *taut*. Taut simply means that if we "pull" on the endpoints of the path we cannot make it any shorter. We now provide a formal definition of a successor and then discuss how this definition can be applied in practice.

Definition 3 *A* successor *of a search node (I, r) is a search node (I', r') such that*

1. *for all points $p' \in I'$, there exists a point $p \in I$ such that the local path $\langle r, p, p' \rangle$ is taut;*
2. *r' is the last common point shared by all paths $\langle r, p, p' \rangle$; and*
3. *I' is maximal according to the points above and the definition of a search node.*

The first requirement (tautness) implies that for each successor $p' \in I'$ can be reached from the root of the current node r by a path that is locally optimal. We will use this property in the next subsection to show that **Anya** always finds a globally optimal path if one exists at all. The third property, requiring that each successor have an interval that is maximal, exists for the purpose of practical efficiency: simply put, we do not want to have arbitrarily small and arbitrarily many successors. Instead, we will make each successor interval as large as possible. The second property, involving a common point shared by all local paths, allows us to differentiate between successors that are *observable* from the current root r and successors which are *non-observable*. We will explore this idea in some detail as it has important practical ramifications.

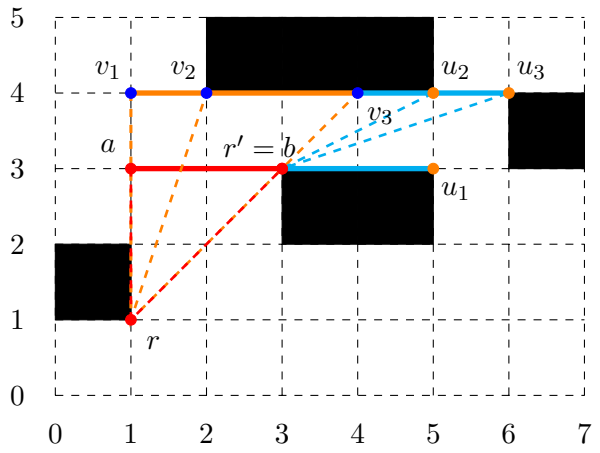

Figure 5: Successors of a cone search node. $([a, b], r)$ has five successors: $([v_1, v_2], r)$ and $((v_2, v_3], r)$ which are observable and $((r', u_1], r')$, $((v_3, u_2), r')$, and $([u_2, u_3], r')$ which are not.

Algorithm 1 Computing the successor set

```
1: function SUCCESSORS(n = (I, r))                          ▷ Takes as input the current node
2:     if n is the start node s then
3:         return generate-start-successors(I = [s])
4:     end if
5:     successors ← ∅
6:     if n is a flat node then
7:         p ← endpoint of I farthest from r                ▷ Successor interval starts from p
8:         successors ← generate-flat-successors(p, r, d)   ▷ Observable successors
9:         if p is a turning point on a taut local path beginning at r then
10:            successors ← successors ∪ generate-cone-successors(p, p, r) ▷ Non-observable successors
11:        end if
12:    else                                                 ▷ If the node is not flat, it must be a cone
13:        a ← left endpoint of I
14:        b ← right endpoint of I
15:        successors ← successors ∪ generate-cone-successors(a, b, r)   ▷ Observable successors
16:        if a is a turning point on a taut local beginning at r then
17:            successors ← successors ∪ generate-flat-successors(a, r)   ▷ Non-observable
18:            successors ← successors ∪ generate-cone-successors(a, a, r) ▷ Non-observable
19:        end if
20:        if b is a turning point on a taut local beginning at r then
21:            successors ← successors ∪ generate-flat-successors(b, r)   ▷ Non-observable
22:            successors ← successors ∪ generate-cone-successors(b, b, r) ▷ Non-observable
23:        end if
24:    end if
25: end function
```

An observable successor is characterised by the fact that all points $p' \in I'$ are visible from the current root point r. In this case the last common point shared by all local paths of the form $\langle r, p, p' \rangle$ is $r = r'$. Observable successors are computed by projecting the current interval on the next row. The projection identifies a maximal interval I_{max} that we will split at each internal corner point point. Each interval produced by the split operation associated with a new observable successor and all such successors share the same root point as the original (parent) node; i.e. each $r' = r$. This process is illustrated in Figure 5 where the interval $I = [a, b]$ is projected onto the next row. The projection identifies a maximal observable interval $I_{max} = [v_1, v_3]$ which is subsequently split to create two observable successors: $([v_1, v_2], r)$ and $((v_2, v_3]), r)$.

By comparison, a non-observable successor is characterised by the fact that all points $p' \in I'$ are not visible from the current root r. In this case all local paths of the form $\langle r, p, p' \rangle$ must pass through a (visibility obstructing) corner point whose identity is $r' = p$. Figure 5 illustrates the process of computing non-observable successors. First, from the non-observable points to the right of the current interval $I = [a, b]$, we construct a single flat successor with $I' = (b, u_1]$ and root $r' = b$. Non-observable points also exist to the left of the current interval I but the local path to each such point (from r through a) is not taut. Other non-observable successors can be found on rows of the grid adjacent to the current interval I. By projecting the corner endpoint b onto the next row of the grid we can construct two further non-observable successors: $((v_3, u_2), b)$ and $([u_2, u_3], b)$.

In Algorithm 1 we give an overview of the procedure that generates the successor set for each search node. An overview of the sub-functions appearing in Algorithm 1 is given in the appendix. The implementation is straightforward, requiring nothing more complicated than grid scanning operations and linear projections. We highlight several important points:

- (Lines 2-4) Observe that the start node is treated as a special case because its root point is located off the grid. The successors of the start node are (i) all non-observable from the root and (ii) can be found to the left and right of the start location, on the row immediately above the start location and on the row immediately below.

- (Lines 6-11) Observe that each flat node yields at most one (observable) flat successor. Each flat node can also yield zero or more (non-observable) cone successors. This occurs when the far-end of the interval (far, relative to the root) is a corner point where the optimal path could potentially turn to reach an adjacent row of the grid.

- (Lines 12-22) Observe that each cone node can yield zero or more observable cone successors. In addition each of the two endpoints of every cone-node interval can also be turning points for the optimal path. In such a case the endpoint becomes a root for up to one flat successor and zero or more cone successors, none of which are observable.

4.3 Evaluating an Anya Search Node

The search procedure of **Anya**, similarly to that of A*, consists in expanding the "most promising" node found so far. It is therefore necessary to evaluate each root and interval pair. This evaluation corresponds to an estimate f of the minimal length of a path from the source to the target through the current interval. An optimality condition of A* is that this estimate is optimistic (i.e. it is never larger than the actual optimal path length). In classical A* where a search node n corresponds to a single point p on the grid the value $f(n)$ is computed as the sum of $g(p)$, the length of the path from the source to p, and $h(p)$, an (under)estimation of the length of the shortest path from p to the target.

As a search node $n = (I, r)$ represents a set of points its f value is the minimum f value of the points in the node:

$$f(n) = \inf_{p \in I} \; f(s, r, p, t)$$

where $f(s, r, p, t)$ is an (under)estimate of the shortest path from s to t through r and p. It should be noted that, because the set of points p is continuous and potentially open, the minimum is replaced by the infimum. Since all points in the interval are visible from r, this value can be broken down as follows:

$$f(s, r, p, t) = g(r) + d(r, p) + h(p)$$

where $d(r, p)$ is the distance between the points r and p.

Finding the point of the interval that minimises the f value may seem like a hard problem since the interval contains a large number of points and we want to avoid to generate them. However the straight-line distance heuristic h ($h(p) = d(p, t)$) makes it easy to isolate the point p that minimises the f value, thanks to the following two simple

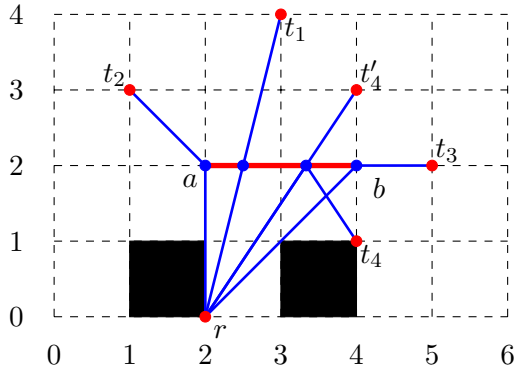

Figure 6: An illustration of Lemmas 1 and 2. The points t_1 and t'_4 correspond to the case where the line intersects the interval; t_2 and t_3 where it does not; t_4 where the mirrored target t'_4 must be used.

geometric observations. More precise heuristics could make it harder to find this optimal point.

Lemma 1 *Let t and r be two points s.t. the interval I is on the same row as t or on a row between the rows of r and t. Then the point p of I with minimum f-value is the point in I closest to the intersection of the straight-line path $\langle t, r\rangle$ with the row of I.*

If the line between r and t intersects the interval then the point p is the intersection. Otherwise this point p is one of the end of the interval. In the event that the precondition of Lemma 1 is not satisfied, it is possible to replace t by its mirrored version t' through I and thus satisfy the precondition. This case is described in Lemma 2.

Lemma 2 *The mirrored point t' of target t through interval I is such that $d(p, t) = d(p, t')$ for all $p \in I$.*

Lemma 2 is a trivial geometrical result. Both lemmas are illustrated on Figure 6.

4.4 Correctness and Optimality

The procedure **Anya** is presented on Algorithm 2. It follows the pattern of A*. It comprises a priority queue, *open*, that stores all the yet-to-be-expanded search nodes ordered by f value. At each step, it extracts one node, verifies whether the corresponding interval contains the target (in which case the returned path is the list of corners in the sequence of search nodes that led to the current node), and expands the node to add the successors of the current node into the priority queue. Some successors may not be inserted in the priority queue (Line 9); we discuss this aspect in the next section: it suffices to know that these successors are part of no optimal path.

To prove correctness and optimality, we show (i) the optimal path appears in the search space, (ii) when the target is expanded we have found an optimal path, and (iii) that each

Algorithm 2 Anya

```
1: input: Grid, source location s, target location t
2: open ← {(I = [s], r_0)}                          ▷ Start node with root r_0 located off the grid
3: while open is not empty do
4:     (I, r) ← pop(open)
5:     if t ∈ I then
6:         return path_to(I)
7:     end if
8:     for all (I', r') ∈ successors(I, r) do
9:         if ¬should_prune(I', r') then                 ▷ Successor pruning
10:            open ← open ∪ {(I', r')}
11:        end if
12:    end for
13: end while
14: return null
```

node in the search space will be reached in a finite number of steps. We assume at this stage that no node is pruned. Note that this version of **Anya** (without any pruning) does not guarantee completeness, i.e., there is no guarantee that the algorithm will terminate if there is no path to the target. We will discuss this aspect of the algorithm in the next section and show that with a suitable pruning policy the algorithm does indeed terminate (the function of the pruning policy is roughly analogous function of the *closed* list in classical A* search). In the same upcoming section we will also discuss the practical ramifications that arise when choosing one pruning policy over another.

In the discussion that follows recall that a search node $n = (I, r)$ represents a set of potential paths (from s to r and from r to each point $p \in I$). Following these semantics we will say that n is a search node of a path π if $r \in \pi$ and I intersects π.

Lemma 3 *If $n = (I, r)$ is a search node of an optimal path π^* then: either n contains the target t or n has at least one successor n' that is also a search node of π^*.*

Proof: We proceed by structural induction over the set of possible successors of n.

`Base case:` n is the start node with $I = [s]$ and r located off the grid. Additionally, n is a search node of π^* (hypothesis). Algorithm 1 (Line 3) scans the traversable points of the grid that are visible from and adjacent to s. These points are located to the left and right of s and they are located on the rows immediately above and immediately below the row of s. Algorithm 1 assigns each of these points to an interval I' and each I' is associated with a successor node that has as its root $r' = s$. Every optimal path must pass through $I = [s]$ and there are no traversable points that can be reached from s without passing through an interval associated with a successor of s. This is sufficient to satisfy the lemma.

`Inductive case:` n is an arbitrary node of π^* and $t \notin I$ (if $t \in I$ there are no further successors and we are done). By definition $r \in \pi^*$ and $p \in I$ is the (apriori unknown) intersection of π^* with the interval I. There are now two possibilities to consider, depending on whether p is a turning point or not. We will show that in both cases there is a successor of n whose interval I' intersects π^*, which is sufficient to satisfy the lemma.

Case 1 $p \in I$ is not a turning point. Algorithm 1 (Lines 8 and 15) scans all points that are adjacent to I and (straight-line) visible from r through I. Each point is assigned to a successor with observable interval I' and root point r. Thus at least one of the successors of n intersects every straight line path from r through p which means at least one successor of n intersects π^*.

Case 2 $p \in I$ is a turning point. In this case p must be a corner endpoint of I, otherwise π^* is not taut and thus cannot be optimal. Algorithm 1 (Lines 10, 17, 18, 21) scans all points that are adjacent to I and reachable from r through p by a taut local path. These points are located on the row of p or on a row that is immediately adjacent. Each such point is assigned to a successor with a non-observable interval I' and root $r = p$. As the process is exhaustive all points reachable by a taut local path, from r though p, must be assigned to an interval. Thus π^* must intersect at least one of the successors of n.

□

Lemma 4 *The first expanded node that contains the target t corresponds to one optimal path to t.*

Proof: Sketch. First we notice that the f-value of a node is indeed the minimal value of all the nodes in the interval, which means that f is an under estimate ($\leq$) of the actual cost to the target. Second we notice that, given a search node (I, r) and its successor (I', r'), for each point $p' \in I'$, the f-value of p' is $\geq$ than the f-value of some point $p \in I$ ($p = r'$ if $r' \neq r$; p is the intersection of I and (r, p') otherwise); the f function is therefore monotonically increasing. Finally, the f function of a search node (I, r) is the length of the path if $t \in I$. Hence the f function of the nodes representing a sub-optimal path to t will eventually exceed the optimal path distance, while the f function of the nodes representing the optimal path will always remain under this value. □

Lemma 5 *If the target is reachable **Anya** will eventually expand a node whose interval includes the target.*

Proof: By contradiction, assume that **Anya** does not expand a node whose interval includes the target. From Lemma 3 we know that failure to expand this node means that **Anya** expand infinitely many nodes. We shall prove that doing so implies that the f value of these nodes is unbounded and, therefore, the target is not reachable.

For most search nodes (I', r'), the interval I' is on a different row than that of its parent (I, r). Therefore, for those nodes, the value $g(p')$ is larger than the value $g(p)$ by 1 or more. This does not happen when the node is flat, but there can be only a bounded number of successive flat nodes.[4] Hence an infinite sequence of successive **Anya** nodes has an infinite length. Finally each **Anya** node has a bounded number of successors, meaning that an

4. And, furthermore, the value $g(p)$ does not increase significantly only for an unobservable flat cone.

infinite number of expansions will have to generate an infinite number of successive nodes. □

5. Improvements to Anya

In this section we discuss several key improvements to the **Anya** algorithm.

- Redundant node pruning
- Cul-de-sac pruning
- Intermediate node pruning

Redundant node pruning prevents **Anya** from generating nodes which have already been expanded and is thus similar in function to the A* closed list. An important benefit of redundant node pruning is that **Anya** will always terminate when this enhancement is applied – even if there does not exist any path from the start location to the target. Meanwhile intermediate node pruning and cul-de-sac pruning both aim to further speed up search by reducing the size of the priority queue. Intermediate pruning achieves this goal by avoiding the explicit generation of nodes that have only a single successor (these successors are expanded immediately, without being added to the `open` list). Cul-de-sac pruning is similar but identifies nodes that can be safely discarded because they cannot possibly lead to the goal. All of these enhancements are applied on-the-fly and all are guaranteed to preserve correctness and optimality.

5.1 Redundant Node Pruning

The version of **Anya** we have presented up to now does not specify any policy for pruning nodes during search. Whilst the algorithm can be used in this way the rules presented here serve two purposes: (i) reducing the number of expanded nodes and, consequently, improving the performance of **Anya**; (ii) guaranteeing completeness, i.e., the property that **Anya** will terminate even if there is no path to the target.

As a motivating example consider Figure 7 where the root r is reached via two paths of different length. In the example the green path is strictly longer than the red path and any points reached via the green path will have a g-value that is strictly larger than the same point when reached via the red path. Figure 8 shows a similar example where both green and red paths have the same cost yet still yield two identical successor nodes with the same interval I. We propose to handle such *root-level* redundancies as follows:

1. We store a hash table of all visited roots with their best g-values. We call this table the *root history*.
2. When generating a node n we check if its root is already in the root history with a g-value less than or equal to its current g-value.
3. If the current g-value of the root improves on the value stored in the root history we add the node to the `open` list. We also update the root history accordingly.

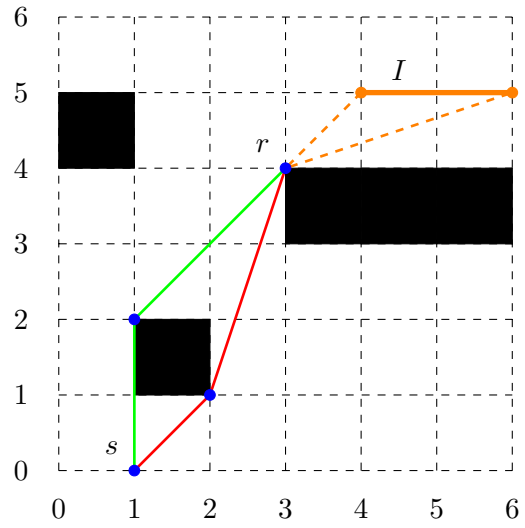

Figure 7: Reaching the same root through two paths of different length.

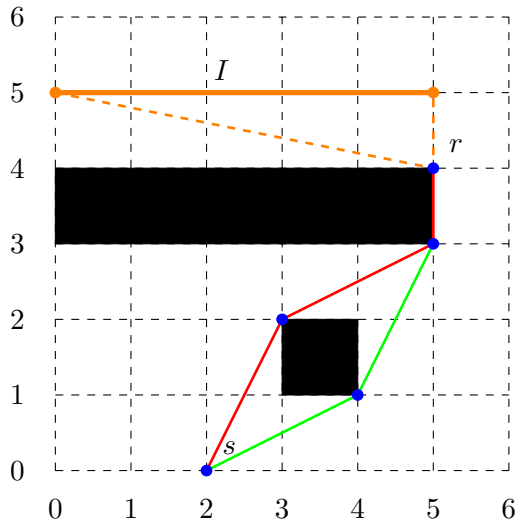

Figure 8: Reaching the same node (interval, root) through two paths of equal length.

4. Alternatively, if the current g-value of the root does not improve on the value stored in the root history we simply ignore the node.

The root history is implemented similarly to a standard A* closed list. We now show that redundant node pruning does not affect the correctness or optimality of **Anya** and that it does indeed guarantee completeness.

Lemma 6 *Redundant node pruning only prunes sub-optimal paths.*

Proof: Trivial. If a search node has a root with a sub-optimal g value, then it represents a sub-optimal path. □

Lemma 7 ***Anya*** *with redundant node pruning always terminates.*

Proof: For **Anya** to not terminate, it must explore paths of arbitrary length. Such paths must eventually involve the same root twice with the root being different in-between. Let n and n' be the two such search nodes. The g value associated with n' must be higher than the g value associated with n and, therefore, node n' must be pruned. Indeed all sufficiently long paths will be pruned and the `open` list will eventually be empty. □

Lemma 8 ***Anya*** *with redundant node pruning keeps at least one optimal path.*

Proof: If a search node $n = (I, r)$ is removed there exists another search node n' (but with different search parents) with a smaller (or equal) g-value that is kept. Assume that n is a search node of an optimal path $p_1, \ldots, p_k$, and let $p_i \in I$ be the point of this path that intersects I. Since the g-value of n is similar to that of n', there exists another path $p'_1, \ldots, p'_i, p_{i+1}, \ldots, p_k$ of similar length, and this path is not pruned. □

5.2 Cul-de-Sac and Intermediate Node Pruning

A* orders the node expansion purely based on the evaluation of the search nodes (i.e. their f-values). It is however possible to alter these ordering rules without compromising the guarantees provided by A*: correctness, optimality and completeness. We present two rules that speed up A* and illustrate their implementation with examples.

The first rule consists in the early identification of *cul-de-sacs* (cds). A cds is a search node that has no successor and does not contain the target. By definition a cds needs not be added to the `open` list since its expansion cannot lead to the target.

The early pruning of these nodes speeds up the algorithm by preventing some operations on `open` (adding and removing these nodes from the priority list) but also by reducing the size of the list, which makes every other operation faster. Adding and then removing a node from a priority list has $\log n$ complexity where n is the size of the list: reducing the size of list is therefore beneficial.

The pruning of the cul-de-sacs is illustrated on Figure 9 for cone nodes and on Figure 10 for flat nodes. In both cases the current node and root are shown in blue while the intervals in red can be pruned.

The second rule can be described as "pushing" the expansion in one direction as far as possible as long as it does not increase the branching factor. Practically if a search node is generated that is guaranteed to have only one successor, then we immediately generate this successor instead of the new node. If said successor also includes only one successor, we recursively generate its successor. The pruning of intermediate nodes is illustrated on Figure 11 for cone nodes and on Figure 12 for flat nodes.

The first obvious benefit of this strategy is a reduction in the number of operations on the `open` list. However a second benefit is that pushing the expansion of a node can lead to a cul-de-sac. When this happens no node at all is added to the `open` list, which helps keep the size of this list small and the operations on this list fast. A potential issue with this strategy is that, because some nodes are generated earlier than normal, they may be generated before we can apply redundant node pruning (presented in the previous subsection). We did not find this issue to significantly affect run-time in our experiments.

5.3 Discussion

We have discussed several different ways in which nodes from the frontier of search can be pruned. Two of the presented approaches, namely cul-de-sac pruning and intermediate node pruning, speed up search along a single fixed path. They do this by pruning away sterile branches and by skipping over intermediate locations where no actual branching occurs. The remaining approach, redundant node pruning, speeds up search by reasoning

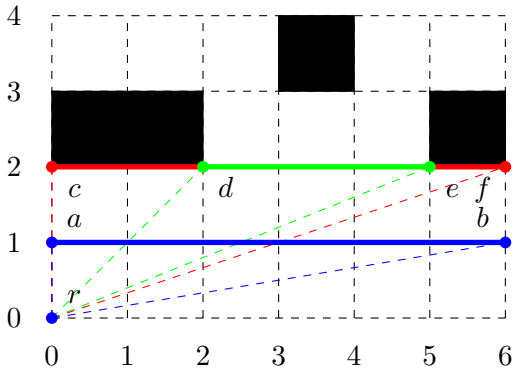

Figure 9: Cul-de-sacs in cone nodes: nodes $([c, d), r)$ and $((e, f], r)$ are not generated.

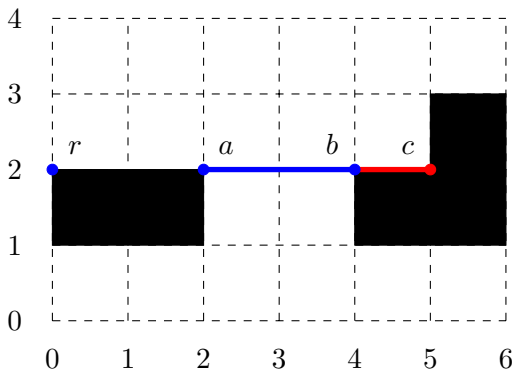

Figure 10: Cul-de-sac on flat nodes: node $((b, c], r)$ is not generated.

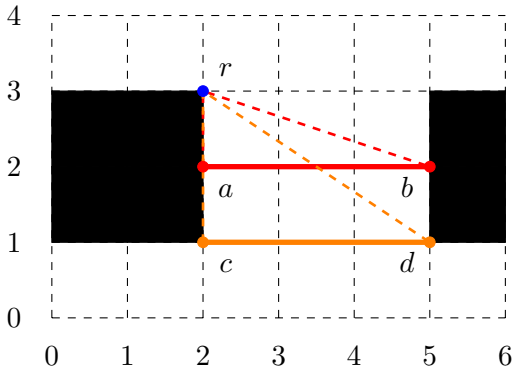

Figure 11: Intermediate node $([a, b], r)$ has only one successor, $([c, d], r)$, which is immediately generated.

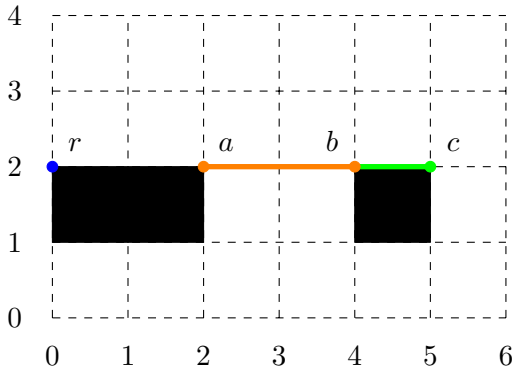

Figure 12: Intermediate node $([a, b], r)$ has only one successor, $((b, c], r)$ which is immediately generated.

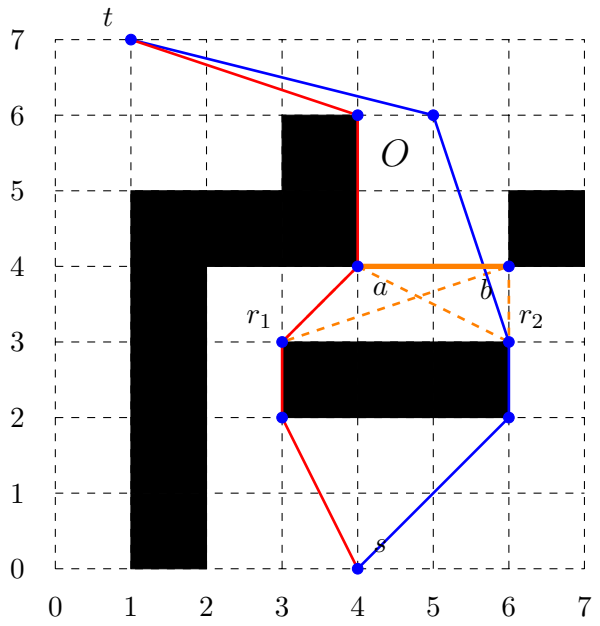

Figure 13: Illustrating that search nodes cannot be trivially pruned with search nodes $n_1 = ([a, b], r_1)$ and $n_2 = ([a, b], r_2)$: if O is not an obstacle the optimal path between s and t goes through n_1 (red); otherwise it goes through n_2 (blue).

more generally about all possible paths that could be used to reach a given point and then pruning away those paths which are sub-optimal with respect to length.

Pruning search nodes in **Anya** is more difficult than in classical A* because each **Anya** node represents a set of positions from the grid rather than just one. Consider the example of Figure 13; we are particularly interested in the interval $[a, b]$ which can be generated with root r_1 or r_2. The shortest path from s to t is through r_1 (~ 9.81 against ~ 10.6 for r_2). However if an obstacle is put on the cell labeled with O, then the optimal path switches to r_2 (~ 11.11 against ~ 11.19). The diagram suggests that, when given the target and two search nodes sharing the same interval, it may not be possible to prune either of them.

The situation described in Figure 13 is not uncommon in practice and such examples may motivate us to derive new and more sophisticated pruning rules to further enhance the performance of the **Anya** algorithm. We must be careful however to weight the improved pruning power of such new techniques against the overhead of applying them in the first instance. For example, an alternative (arguably, better) approach to avoiding redundant node expansions is to keep an *interval history* list in addition to (or instead of) a *root history*. Such a method would certainly avoid the problem outlined in Figure 13 but there are many more possible intervals than roots, which means the size of the hash table is potentially much larger and memory accesses are potentially slower. Additionally, comparing intervals for equality and membership requires extra time and may not be worth the investment[5].

5. We attempted a similar experiment but the results were not clearly positive.

Benchmark	#Maps	#Instances	Nodes Expanded by A*						
			Min	Q1	Median	Mean	Q3	Max	StDev
Baldur's Gate II	75	93160	2	166	2019	6302	9170	86720	9136
Dragon Age	156	159465	1	622	5880	14080	19150	126800	19744
StarCraft	75	198230	3	4808	26840	50000	70110	578900	63507
Random 10%	10	16770	2	239	548	1886	1485	59280	3921
Random 20%	10	17740	3	749	3869	8606	14680	53760	9905
Random 30%	10	19200	4	3520	14190	20290	33710	96090	19162
Random 40%	10	35360	3	12520	42850	51920	83770	169900	43558

Table 1: An overview of the seven benchmark problems used in our experiments. We give the number of maps and problem instances in each benchmark and the distribution of nodes expanded by a reference algorithm (A*) when solving all problems in each benchmark set.

6. Experiments

We conduct experiments on seven benchmark problem sets taken from Nathan Sturtevant's well known repository at `http://movingai.com/benchmarks`. Three of the benchmarks originate from popular computer games and often appear in the literature. They are: Baldur's Gate II , Dragon Age Origins and StarCraft . The maps in these benchmarks vary in size; from several thousand nodes up to several million. The remaining four benchmarks comprise grids of size 512×512 with randomly placed obstacles of varying densities, from 10% to 40%. Table 1 gives an overview of the benchmark problems. We give the number of maps and instances per problem set and a distribution for the number of node expansions required by a reference algorithm, (A*, using an octile distance heuristic[6]), to solve all problems in each benchmark set. The latter metric gives us an indicator for comparing the "difficulty" of problems appearing in each benchmark set.

We compare our purely online and optimal **Anya** algorithm with a number of state-of-the-art any-angle techniques. These are: Theta* (Nash et al., 2007), Lazy Theta* (Nash, Koenig, & Tovey, 2010), Field A* (Uras & Koenig, 2015a) and an any-angle variant of two-level Subgoal Graphs (SUB-TL) (Uras & Koenig, 2015b). All of these approaches are near-optimal and are not guaranteed to return the shortest path. The methods Theta*, Lazy Theta* and Field A* are all purely online. Only SUB-TL relies on an offline pre-processing step to further improve the performance of search. We use C++ implementations for each of these algorithms; the source codes are made publicly available in (Uras & Koenig, 2015a).

Anya is implemented in Java and executed on JVM 1.8. To allow for comparisons across different implementation languages we use the A* algorithm (Hart, Nilsson, & Raphael, 1968), implemented in both C++ and Java, as a reference point[7]. All experiments are performed on a 3GHz Intel Core i7 machine with 8GB of RAM and running OSX 10.8.4.

6. Octile distance is analogous to Manhattan distance but generalised to 8-connected grids.
7. The C++ implementation is taken from (Uras & Koenig, 2015a); the Java implementation is our own.

Benchmark	Avg. Node Expansion Speedup					Avg. Path Length Improvement (%)				
	Anya	Theta*	L.Theta*	F.A*	SUB-TL	Anya	Theta*	L.Theta*	F.A*	SUB-TL
Baldur's Gate II	91.13	1.95	1.96	1.01	907.10	4.65%	4.62%	4.61%	4.38%	4.58%
Dragon Age	19.60	1.05	1.05	0.90	57.45	4.34%	4.27%	4.22%	4.05%	4.28%
StarCraft 40%	40.73	1.27	1.27	0.95	166.00	5.02%	4.95%	4.92%	4.70%	4.88%
Random 10%	0.80	2.34	2.38	1.14	6.60	4.77%	4.63%	4.58%	3.83%	4.59%
Random 20%	0.77	1.23	1.17	0.80	2.56	4.57%	4.34%	4.15%	3.26%	4.30%
Random 30%	1.06	0.82	0.75	0.64	1.68	4.44%	4.12%	3.77%	3.12%	4.03%
Random 40%	2.20	0.90	0.86	0.82	2.40	4.14%	3.95%	3.48%	3.22%	3.74%

Table 2: We compare the performance of each algorithm in terms of average node expansion speedup and average path length improvement. Both metrics are taken with respect to our reference algorithm, A*. In both cases higher is better.

7. Results

We evaluate performance using three different metrics: search time, nodes expanded and path length. All results are presented relative to a benchmark algorithm, A*, which we combine with a standard octile distance heuristic. For example, when comparing search time or nodes expanded, we will give figures for the relative speedup of each algorithm vs A*. Under this paradigm a search time speedup of 2 means twice as fast while a node expansion speedup of 2 means half as many nodes were expanded. When comparing path length we give the percent improvement in path length vs A*. In all cases higher is better.

We begin with Table 2 which shows average performance figures for nodes expanded and path length on each of our seven benchmark problem sets. We make the following observations:

- **Anya** is best of the four purely-online algorithms in our comparison. Only the pre-processing-based SUB-TL algorithm expands fewer nodes, on average.
- **Anya**, as with all methods in our comparison, struggles to achieve a speedup on the four random benchmarks. In two of the four cases its performance is below that of the reference A* algorithm. Again, only the pre-processing-based SUB-TL algorithm is able to achieve a consistent, though much reduced, node expansion speedup.
- **Anya**, being optimal, shows the best improvement in path length; however all algorithms in our comparison are very close to optimal, on average.

Next, we evaluate performance in terms of search time. Rather than taking a simple average on a per benchmark basis (or across all benchmarks) we instead sort instances according to difficulty, as measured by the number of node expansions required for the reference A* algorithm to solve each problem. This approach gives a more holistic overview of performance and is independent from any bias associated with the selection of instances that comprise each benchmark set[8]. Results from this analysis are given in Figure 14. We make the following observations:

8. As per Table 1, problem instances that can be regarded as "easy" often outnumber instances that can be regarded as "hard". These difference have the effect of skewing performance indicators that are computed as simple averages over all instances in each benchmark set.

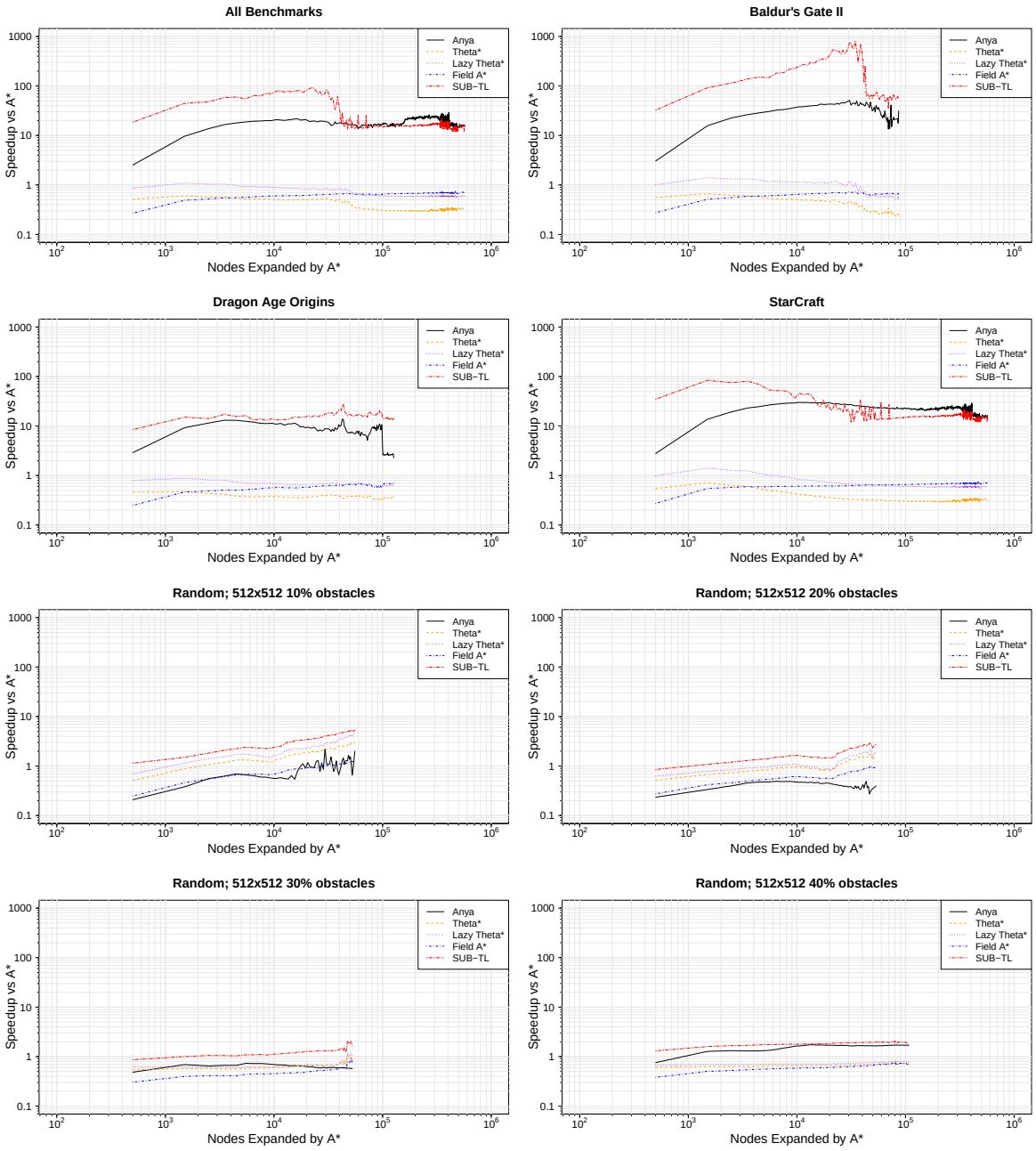

Figure 14: Search time speedup. We compare performance on each of our seven benchmarks in terms of search time. Figures are given as relative speedup vs. a reference A* algorithm. Problem instances are sorted by difficulty using A* node expansion rank. Note that each plot is log-log.

- **Anya** is often more than one order faster than the reference A* algorithm on the benchmarks drawn from real computer games. Performance is mixed on the four random benchmarks, with all evaluated methods struggling to achieve a speedup.

- **Anya** is the fastest of the four purely online methods under evaluation. Its performance is often comparable with the pre-processing based SUB-TL technique and, on particularly challenging instances from the StarCraft domain, **Anya** is non-dominated.

- **Anya**'s performance in terms of search time is lower than suggested by the previously evaluated node expansion metric. This reflects the fact that each node expansion made by **Anya** involves analysing the grid; looking for roots and searching for intervals.

7.1 Discussion

We have seen that **Anya** compares well with current state-of-the-art any-angle pathfinding algorithms. In an (almost) apples-to-apples comparison with three contemporary and purely online search technique (Theta*, Lazy Theta* and Field A*) we have seen that **Anya** usually expands fewer nodes per search and terminates up to one order of magnitude faster. These results are further underscored by the fact that **Anya** is the only online algorithm that guarantees to return a Euclidean-optimal path. We may surmise that, in many cases and applications, **Anya** appears preferable to each of these alternative algorithms.

Next, we make an apples-to-oranges comparison between **Anya** and the near-optimal and offline enhanced SUB-TL algorithm. We have seen that while **Anya** is usually not as fast as SUB-TL its performance is often comparable. Moreover, **Anya** retains an advantage when solving especially challenging instances drawn from real computer games. SUB-TL appears to be preferable to **Anya** in cases where additional space and time is available to create and store its associated subgoal graph or in cases where such overheads can be amortised over many online instances. When extra space and time is not available, or in cases where the map is subject to change (e.g. new obstacles are added or existing obstacles are removed), **Anya** appears to be preferable to SUB.

The main strength of **Anya** is that it searches over sets of nodes from the grid rather than considering individual locations one at a time. Expansion can thus be considered as a macro operator, meaning that **Anya** bears some similarity to speedup techniques using hierarchical abstraction (e.g. HPA* (Botea et al., 2004)). An important difference is that **Anya** constructs its abstract graph on-the-fly rather than as part of a pre-processing step.

One current drawback associated with **Anya** is that nodes can contain overlapping intervals. This occurs when an interval is reachable from two different root points, neither of which can be pruned (e.g. when both root locations are reached for the first time; as illustrated in Figure 13). Such nodes are, either in part or in whole, redundant and — provided their f-value is smaller than the optimal distance to the goal — will themselves beget yet more redundant successors. We can see this behaviour especially in the results for benchmarks Random 10% and Random 20% where SUB-TL achieves a speedup of several factors while **Anya** struggles to maintain parity with the reference A* algorithm. It seems reasonable to improve the current algorithm by attempting to identify such overlaps in order to prune them from consideration. An efficient and effective algorithm for achieving this goal is the subject of further work.

8. Related Work

Among the simplest and most popular approaches for solving the any-angle pathfinding problem is string-pulling. The main idea is to find a path on the input grid map (most often using some variant of A* search (Hart et al., 1968)) and then post-process that path in order to remove unnecessary turning points. Several such methods have appeared in the literature of Game Development; e.g. (Pinter, 2001; Botea et al., 2004).

A number of algorithms improve on string-pulling by interleaving node expansion and path post-processing during online search. Particular examples include Field D* (Ferguson & Stentz, 2005) and Field A* (Uras & Koenig, 2015a), both of which use linear interpolation to smooth grid paths one cell at a time, and Theta* (Nash et al., 2007), which introduces a shortcut each time a successful line-of-sight check is made; from the parent of the current node to any of its successors. Though still sub-optimal in many cases such approaches are nevertheless attractive for being able to search purely online and for being efficient in practice. In addition to the two examples given there are numerous other works, often appearing in the literature of Artificial Intelligence, that apply that apply and improve on the basic interleaving idea. We refer the interested reader to (Nash & Koenig, 2013) for a recent survey and overview.

Accelerated A* (Šišlák, Volf, & Pěchouček, 2009) is an online any-angle algorithm that is conjectured to be optimal but for which no strong theoretical argument is made. Similar to Theta*, it differs primarily in that line-of-sight checks are performed from a set of expanded nodes rather than a single ancestor. The size of the set is only loosely bounded and, for challenging problems, can include a large proportion of nodes on the closed list.

One recent and successful line of research involves the combination of string-pulling with an offline pre-processing step. Such works are compelling because they can significantly improve on the performance of purely online search; not just in terms of solution quality but also running time. Block A* (Yap, Burch, Holte, & Schaeffer, 2011) is one such example. This sub-optimal algorithm pre-computes a database of Euclidean-optimal distances in all possible tile configurations of a certain size (e.g. all possible 3x3 blocks). The database obviates the need for explicit visibility checks or indeed any type of online string-pulling. The pre-processing step needs to be performed exactly once; the database remains valid if the tiles on the map change or indeed if the map itself changes entirely. Another recent work improves on Theta* by combining that algorithm with a pre-processing based graph abstraction technique (Uras & Koenig, 2015b). This approach, referred to in Section 7 as SUB-TL, is shown to improve on both the running time and solution quality of Block A*. The main disadvantage (vs. Block A*) is that the abstract graph needs to be re-computed or repaired each time the map changes.

The Euclidean Shortest Path Problem is a well known and well researched topic in the areas of Computational Geometry and Computer Graphics. It can be seen as a generalisation of the Any-angle Pathfinding Problem. It asks for a shortest path in a plane but does not impose any restrictions on obstacle shape or obstacle placement (cf. grid aligned polygons made up of unit squares).

Visibility graphs (Lozano-Pérez & Wesley, 1979) are a family of well-known and popular techniques for optimally solving the Euclidean Shortest Path Problem. Searching in such graphs requires $O(n^2 \log_2 n)$ time but the approach can be much faster in practice. There

are two main disadvantages: (i) computing the graph requires an offline pre-processing step and $O(n^2)$ space to store; (ii) the graph is static and must be recomputed or repaired if the environment changes. More sophisticated variants such as Tangent Graphs (Liu & Arimoto, 1992) and Silhouette Points (Young, 2001) are particularly efficient variants of visibility graphs but the same disadvantages apply.

Another family of exact approaches for solving the Euclidean Shortest Path Problem is based on the Continuous Dijkstra paradigm (Mitchell et al., 1987). The most efficient of these algorithms (Hershberger & Suri, 1999) involves a pre-computation requiring $O(n \log_2 n)$ space and $O(n \log_2 n)$ time. The result is a Shortest Path Map; a planar subdivision of the environment that can be used to find a Euclidean shortest path in just $O(\log_2 n)$; but only for queries originating at a fixed source. Like visibility graphs, this approach also introduces additional memory overheads (storing the subdivision) and the pre-processing step must be re-executed each time the environment or the start location changes.

9. Conclusion

We study any-angle pathfinding: a problem commonly found in the areas of robotics and computer games. The problem involves finding a shortest path between two points in a grid but asks that the path is not artificially constrained to the fixed points of the grid. The best known algorithms for solving the any-angle problem, to date, all compute approximate solutions rather than optimal shortest paths. Additionally only one approach to date has been able to achieve a consistent speedup vs. the A* algorithm — a common reference point for measuring performance in the literature — and then only when the environment remains static and unchanging. In this work we present a new online, optimal and practically efficient any-angle technique: **Anya**. Where other works obtain good performance by reasoning at the grid level our method considers sets of points from the grid which are taken together as contiguous intervals. This approach requires revisiting the classical definition of search nodes and successors and requires the introduction of a new technique for computing the f-value of each node. We give a thorough algorithmic description of this new search paradigm and we give theoretical arguments for its completeness and optimality preserving characteristics.

In an (almost) apples-to-apples comparison we evaluate **Anya** against three contemporary near-optimal and online techniques: Theta*, Lazy Theta* and Field A*. We show that in a large majority of instances **Anya** is faster than each of these alternatives, all while guaranteeing to find an optimal shortest path. In an apples-to-oranges comparison we evaluate **Anya** against SUB-TL: a very fast pre-processing-based near-optimal any-angle technique. We show that **Anya** is non-dominated when compared to SUB-TL and even maintains an advantage on some particularly challenging instances drawn from real computer games. Another advantage is that, unlike SUB-TL, **Anya** does not assume the map is static; i.e. it can be readily applied to pathfinding problems involving dynamically changing terrain.

Any-angle pathfinding has received significant attention from the AI and Game Development communities but until now it has been an open question whether any optimal and online algorithm exists. **Anya** answers this question in the affirmative.

9.1 Future Work

There are several possible directions for future work. Perhaps the most obvious is the development of improvements and extensions to the current **Anya** algorithm. In the first instance we believe the empirical performance of **Anya** could be enhanced by generating successors nodes that do not contain any redundant (or partially redundant) intervals. One possibility is to keep a closed list of previously encountered intervals. A stronger variant of this idea involves bounding the g-value of grid intervals and only generating successor nodes when at least one point inside a candidate interval can be relaxed. A related and orthogonal improvement involves pre-processing the grid and identifying intervals apriori. This enhancement can speed up search by avoiding entirely all grid scanning and interval projection operations that are currently necessary in order to generate each node.

We have seen that reasoning over sets of points from the grid, rather than individual locations, is computationally beneficial. We believe the same type of search paradigm employed by **Anya** can be generalised to improve the performance of grid-optimal search in addition to any-angle pathfinding.

As a final suggestion for further work, we believe **Anya** can be generalised to two-dimensional maps with arbitrarily shaped polygonal obstacles, rather than just grids. A benefit of this generalisation would be to avoid the discretisation of the world in which a path is searched for. This would even improve the quality of the path returned as the optimal any-angle path is often non optimal in the non-discretised version of the map.

10. Acknowledgements

We thank Tansel Uras for assistance with the source codes used in the experimental section of this paper. We also thank Adi Botea and Patrik Haslum for helpful suggestions during the early development of this work.

The work of Daniel Harabor and Alban Grastien is supported by NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

The work of Dindar Öz and Vural Aksakalli is supported by The Scientific and Technological Research Council of Turkey (TUBITAK), Grant No. 113M489.

References

Botea, A., Müller, M., & Schaeffer, J. (2004). Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, *1*(1), 7–28.

Ferguson, D., & Stentz, A. (2005). Field D*: An Interpolation-Based Path Planner and Replanner. In *Robotics Research: Results of the 12th International Symposium, ISRR 2005, October 12-15, 2005, San Francisco, CA, USA*, pp. 239–253.

Graham, R. L., Knuth, D. E., & Patashnik, O. (1989). *Concrete Mathematics - A Foundation for Computer Science*. Addison-Wesley.

Harabor, D. D., & Grastien, A. (2013). An Optimal Any-Angle Pathfinding Algorithm. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Hershberger, J., & Suri, S. (1999). An Optimal Algorithm for Euclidean Shortest Paths in the Plane. *SIAM Journal on Computing*, *28*(6), 2215–2256.

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

Lozano-Pérez, T., & Wesley, M. A. (1979). An Algorithm for Planning Collision-Free Paths Among Polyhedral obstacles. *Communications of the ACM*, *22*(10), 560–570.

Mitchell, J. S. B., Mount, D. M., & Papadimitriou, C. H. (1987). The Discrete Geodesic Problem. *SIAM Journal on Computing*, *16*(4), 647–668.

Nash, A., Daniel, K., Koenig, S., & Felner, A. (2007). Theta*: Any-Angle Path Planning on Grids. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pp. 1177–1183.

Nash, A., & Koenig, S. (2013). Any-Angle Path Planning. *AI Magazine*, *34*(4), 9.

Nash, A., Koenig, S., & Tovey, C. A. (2010). Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*.

Pinter, M. (2001). Toward More Realistic Pathfinding. *Game Developer Magazine*, *8*(4).

Šišlák, D., Volf, P., & Pěchouček, M. (2009). Accelerated A* Trajectory Planning: Grid-based Path Planning Comparison. In *4th ICAPS Workshop on Planning and Plan Execution for Real-World Systems*.

Uras, T., & Koenig, S. (2015a). An Empirical Comparison of Any-Angle Path-Planning Algorithms. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel.*, pp. 206–211.

Uras, T., & Koenig, S. (2015b). Speeding-Up Any-Angle Path-Planning on Grids. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, pp. 234–238.

Yap, P., Burch, N., Holte, R. C., & Schaeffer, J. (2011). Block A*: Database-Driven Search with Applications in Any-Angle Path-Planning. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*.

Young, T. (2001). Optimizing Points-of-Visibility Pathfinding. In *Game Programming Gems 2*, pp. 324–329. Charles River Media.

Appendix

We provide additional details for the implementation of **Anya**'s successor set generation algorithm. Our method depends on basic operations which are technically simple: grid scanning, traversability tests and linear projection operations. We do not attempt to reproduce the mechanical details of such operations. Instead we focus our presentation toward intuitive understanding of the overall process.

Algorithm 3 Computing the successor set, supplemental.

function GENERATE-START-SUCCESSORS(a traversable and discrete start location s)
  $p \leftarrow$ first turning point (else farthest obstacle vertex) left of s
  $p' \leftarrow$ first turning point (else farthest obstacle vertex) right of s
  Construct a maximal half-closed interval I^1_{max} containing all observable points left of s
  Construct a maximal half-closed interval I^2_{max} containing all observable points right of s
  Construct a maximal closed interval I^3_{max} containing all visible points from the row above s
  Construct a maximal closed interval I^4_{max}, containing all visible points from the row below s
  $intervals \leftarrow$ Split each I^k_{max} at each corner point and take their union
  Construct for each $I \in intervals$ a new cone successor node with $r = s$
  return all start successors
end function

function GENERATE-FLAT-SUCCESSORS(an interval endpoint p, a root point r)
  if points r and p are on the same row **then** ▷ Observable successors
    $p' \leftarrow$ first turning point (else farthest obstacle vertex) in direction r-to-p
    $I_{max} \leftarrow$ new maximal interval with endpoints p (open) and p' (closed)
    $successors \leftarrow$ new flat node $n = (I_{max}, r)$
  else if p is adjacent to a traversable node that is not visible from r **then**
    $p' \leftarrow$ first turning point (else farthest obstacle vertex) on the row of p such that $\langle r, p, p' \rangle$ is taut
    $I_{max} \leftarrow$ new maximal interval with endpoints p (open) and p' (closed)
    $successors \leftarrow$ new flat node $n = (I_{max}, p)$ ▷ Non-observable flat successors
  end if
  return *successors*
end function

function GENERATE-CONE-SUCCESSORS(an interval endpoint a, an interval endpoint b, a root point r)
  if a and b and r are from the same row **then** ▷ Non-observable successors of a flat node
    $r' \leftarrow a$ or b, whichever is farthest from r ▷ Previously established this is a turning point
    $p \leftarrow$ a point from an adjacent row, reached via a right-angle turn at a ▷ Obstacle following
    $I_{max} \leftarrow$ a maximum closed interval, beginning at p and entirely observable from r'
  else if $a == b$ **then** ▷ Non-observable successors of a cone node
    $r' \leftarrow a$
    $p \leftarrow$ a point from an adjacent row, computed via linear projection from r through a
    $I_{max} \leftarrow$ a maximum closed interval, beginning at p and entirely observable from r'
  else ▷ Observable successors of a cone node
    $r' \leftarrow r$
    $p \leftarrow$ a point from an adjacent row, computed via linear projection from r through a
    $p' \leftarrow$ a point from an adjacent row, computed via linear projection from r through b
    $I_{max} \leftarrow$ a maximum closed interval, with endpoints a and b, which is entirely observable from r'
  end if
  for all $I \in$ { split I_{max} at each corner point } **do**
    $n' \leftarrow$ new search node with interval I and root point r'
    $successors \leftarrow successors \cup I$
  end for
  return successors
end function