Compromise-free Pathfinding on a Navigation Mesh

Michael L. Cui
Monash University
Melbourne, Australia
mlcui1@student.monash.edu

Daniel D. Harabor
Monash University
Melbourne, Australia
daniel.harabor@monash.edu

Alban Grastien
Data61, Canberra
Australian National University
alban.grastien@data61.csiro.au

Abstract

We want to compute geometric shortest paths in a collection of convex traversable polygons, also known as a navigation mesh. Simple to compute and easy to update, navigation meshes are widely used for pathfinding in computer games. When the mesh is static, shortest path problems can be solved exactly and very fast but only after a costly preprocessing step. When the mesh is dynamic, practitioners turn to online methods which typically compute only approximately shortest paths. In this work we present a new pathfinding algorithm which is compromise-free; i.e., it is simultaneously fast, online and optimal. Our method, **Polyanya**, extends and generalises **Anya**; a recent and related interval-based search technique developed for computing geometric shortest paths in grids. We show how that algorithm can be modified to support search over arbitrary sets of convex polygons and then evaluate its performance on a range of realistic and synthetic benchmark problems.

1 Introduction

A navigation mesh divides the traversable space in a planar environment into a collection of convex polygons. Popular with researchers and game developers alike, navigation meshes offer compelling advantages vs. other representational techniques: (i) they are complete, meaning every traversable point appears in the mesh; (ii) they usually have low density, meaning they have small memory requirements and are fast to search; (iii) they are flexible, meaning they are easily constructed and easily modified. Given two points on a navigation mesh one can compute a path between them in a variety of ways but only after accepting some type of compromise. We briefly describe three popular and broadly representative techniques; all assume the input mesh is a Constrained Delaunay Triangulation (CDT):

- Channel Search [Kallmann, 2005] is a two-step technique that first finds an abstract path between the polygons containing the start and target. Then, as part of a post-processing step, the path of triangles is refined to a concrete sequence of points. The compromise in this case is optimality: Channel Search usually returns only an approximately shortest path.
- TA* [Demyen and Buro, 2006] improves on Channel Search by computing approximately shortest paths in a single shot. A significant advantage is that TA* can compute optimal paths, but only by repeating the search, possibly many times, and caching each result. The compromise in this case is optimality *or* speed.
- TRA* [Demyen and Buro, 2006] is a performance-oriented extension of TA* which involves an offline preprocessing step (a smaller abstract graph is constructed which is faster to search). The principal compromise in this case is flexibility: TRA* works best when the environment is static. However, there is clear and significant interest in pathfinding with navigation meshes in dynamic environments; e.g. [van Toll *et al.*, 2012; Kallmann *et al.*, 2004] and commercial pathfinding libraries such as `Unreal Engine` and `NavPower`.

In this paper we present a new algorithm for navigation-mesh pathfinding that is compromise-free: i.e., simultaneously fast, online and optimal. Our method, **Polyanya**, extends and generalises **Anya** [Harabor *et al.*, 2016]; from the related Any-angle Pathfinding Problem (where obstacles have to follow the grid) to the more general Euclidean Shortest Path Problem (where obstacles can be arbitrary polygons). When searching for a path **Polyanya** employs a novel encoding where sets of points, drawn from the edges of the mesh, are taken together as contiguous intervals. Each interval is evaluated using an admissible heuristic and its successors are derived by projecting the interval, from one polygon to the next. **Polyanya** always computes (in a single online search) the length of the optimal path between any two points on the mesh if such a path exists. We give a full theoretical description of the algorithm and demonstrate its efficacy on a range of problem instances drawn from real games.

2 Problem Statement

We now establish the necessary terminology to precisely describe the Euclidean Shortest Path Problem (ESPP) and its objective function. Figure 1 will help to illustrate some ideas.
Polygons: A *polygon* is a bounded figure in a plane whose shape is defined by a set of points called *vertices* and which are in turn connected by a closed set of *edges*. Each edge

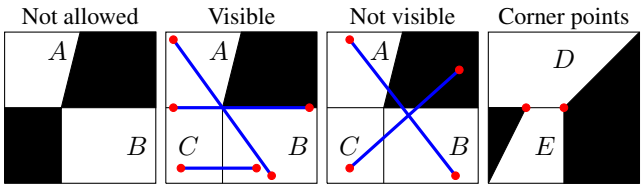

Figure 1: Examples of visible points, non-visible points and corner points. We show several simple meshes with traversable polygons labeled A–E. Points of interest are shown as circles.

$e = (a, b)$ is a contiguous interval between two different vertices; it is closed at both ends and never intersects any vertex except a and b. This means edges can overlap but only at their endpoints. Polygons can overlap but only if they share a common edge or vertex. Two polygons that share an edge are said to be *adjacent*.

Maps and Meshes: A *map* is a plane in which every point is either traversable or not traversable. Any such map can be represented, either exactly or to within some arbitrary precision, by a collection of convex polygons, each of which is either traversable or not traversable. The set of all traversable polygons, $\mathbf{M}$, is known collectively as a *Navigation Mesh*. The remaining polygons are called *obstacles*.

Visibility: We say that two traversable points are *co-visible* if they can be connected by a straight line that intersects only points found in $\mathbf{M}$. An agent can travel in straight line between any two co-visible points.

There are different ways to interpret a situation like the one depicted on the first example of Fig. 1, whether an agent should be able to "squeeze" in between the two obstacles and, if not, what it means to start from such an ambiguous point. The ambiguity can be removed by inserting a very small, non-zero, polygon centred on the ambiguous point. This polygon is traversable if the agent is allowed to squeeze in; otherwise it is not. The fact that the polygon is very small implies that the set of paths remains essentially the same. Because this disambiguation technique is possible we assume in this paper that such ambiguous situations never occur.

Corner Points: A *corner point* is a vertex c that appears in some optimal path $\langle p_1, c, p_2 \rangle$ s.t. p_1 and p_2 are not co-visible.

Paths: A *path* is a sequence of traversable points $\pi = \langle p_1, \ldots, p_k \rangle$ where each successive pair of points, p_i and p_{i+1}, are co-visible. The *length* of π is the cumulative total of all straight-line (Euclidean) distances d between every successive pair of points; i.e., $len(\pi) = \Sigma_{i=1}^{k-1}\, d(p_i, p_{i+1})$. A path is *optimal* if there exists no alternative path π' whose length is shorter.

Objective Function: The *Euclidean Shortest Path Problem* takes as input a pair of traversable points drawn from the map, a start point s and a target point t, and asks for a path between them, π^*, such that $len(\pi^*)$ is minimum.

3 Algorithm Description

Polyanya is a novel online algorithm for computing Euclidean shortest paths among polygonal obstacles in the plane. **Polyanya** takes as input a navigation mesh and a pair of traversable points called the *start* and *target*; it returns an optimal path between the start and target points if such a path exists. As is typical in game settings we assume the navigation mesh is created apriori: e.g., by a level designer or as the result of applying one of the many readily available mesh generation programs to an environment (in our experiments we employ the freely available triangulation library `Fade2D`[1]). At its core **Polyanya** performs a forward-driven point-to-point shortest path search. It follows a well established schema whose main components are:

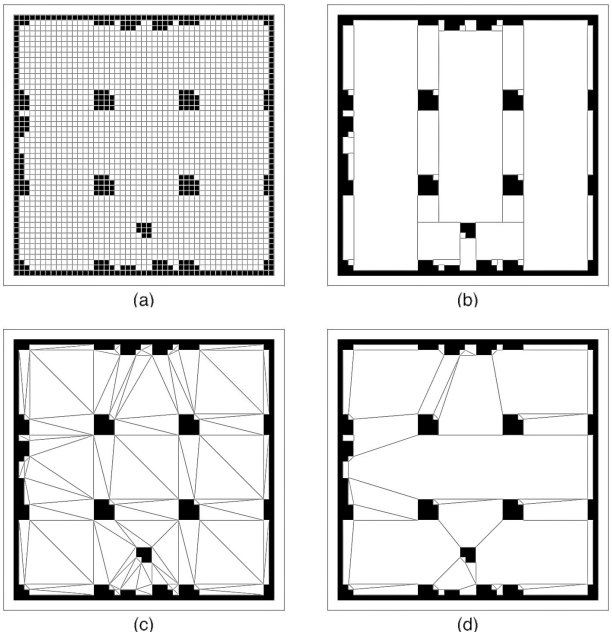

Figure 2: (a) A grid mesh. This game map appeared among the benchmarks at the 2014 Grid-based Path Planning Competition. (b) A merged grid mesh. (c) A CDT mesh. (d) A merged CDT mesh.

1. A *model*, to represent partial solutions (i.e., the search nodes).
2. A *successor function*, which takes a search node, the parent, and generates a set of related search nodes, the children, such that every child can be reached from the parent directly and without intersecting any obstacles.
3. An *evaluation function*, to measure how promising each generated search node appears.
4. A *priority queue*, used to track the order of expansion.

In this section we describe items 1-3 and then paint in broad strokes the overall search and pruning strategies employed by **Polyanya**. Our priority queue (item 4) is a standard binary heap whose description we omit.

3.1 Search Nodes

A search node is a partial solution to a pathfinding problem. Under a classical best-first model each partial solution is identified by a single node n that represent a single path to the target. **Polyanya** generalises this model by representing together and in a single node a set of related partial solutions (cf. just one) which all have a common prefix path from s.

[1] `http://www.geom.at/fade2d/html/`

Definition 1 (From [Harabor *et al.*, 2016]) *A* search node (I, r) *is a tuple where* r *is a point called the* root *and* $I = [a_I, b_I]$ *is an interval such that each point* $p \in I$ *is visible from* r*. 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.*

The generalised definition can be interpreted as follows: there exists a concrete (possibly optimal) path from s to r. At the point r the path splits, creating a partial solution for each point $p \in I$. **Polyanya** considers all of these solutions together and at the same time.

3.2 Successors

The *successor function* is an important part of any pathfinding algorithm: its role is to advance the frontier of the search and in the process bring us closer to the target. Such a function, $\mathbf{succ}(n)$, takes as input a search node, n, and returns a set of adjacent search nodes $n' \in \mathbf{succ}(n)$ which can all be reached in a single *step* or more generally through the application of a single *action*. We now give a formal definition for **Polyanya**'s nodes and then describe how to compute them.

Definition 2 (From [Harabor *et al.*, 2016]) *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 path* $\langle r, p, p' \rangle$ *is taut (i.e., locally optimal);*
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.*

Most pathfinding algorithms advance the search frontier from one point to the next. **Polyanya** is different: it proceeds instead from interval to interval. In broad strokes, we generate the successors of a search node (I, r) by applying an action that *pushes* the interval I away from r and through the interior of an adjacent traversable polygon $Y \in \mathbf{M}$ (called the *opposite* polygon). The successors of (I, r) are found on the perimeter of Y; since Y is convex those points can be reached from I with straight lines. Once the pushing operation is complete we simply walk the perimeter of Y and identify which points, if any, can be assigned to a successor node interval, I'. We distinguish between two kinds of successors:

1. *Observable successors* are those where every $p' \in I'$ is visible, from r through I. They have as their root $r' = r$.
2. *Non-observable successors* are those where every $p' \in I'$ is not visible from r. Since each p' must be reached by a taut local path $\langle r, r', p \rangle$ we set r' as one of the endpoints of the interval $I = [a, b]$. Such paths can only be taut if r' is also a corner point.

Figure 3 gives an example of the pushing process and of the different types of successors. To compute the set of observable successors we simply *project* the node (I, r) through the shaded polygon and onto the points of its perimeter. Each edge (or part thereof, if only a section is visible) in the projected region becomes an interval for an observable successor whose root is $r' = r$. The set of non-observable successors is found to the left and right of the projected region – but only if

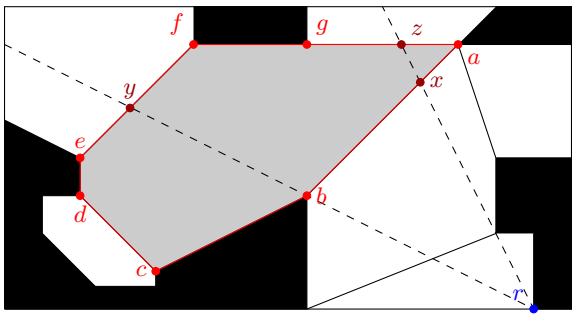

Figure 3: Search node $([x, b], r)$ produces three observable successors: $([y, f], r)$, $([f, g], r)$, and $([g, z], r)$; four non-observable successors: $([b, c], b)$, $([c, d], b)$, $([d, e], b)$, and $([e, y], b)$.

the corresponding left and right endpoint of the interval I is also a corner point. When this is not the case we do not generate any non-observable successors through that endpoint (because any local path to such a successor is sub-optimal).

The initial node requires some extra care. When the start point sits inside a polygon all successors are defined by (I, r) where the interval I is an edge of that polygon and the root r is the start point. In case the start point lies on an edge or a vertex the polygon is no longer unique, which makes the procedure slightly more complex. In these cases the intervals of the successors are all the edges (of any of these polygons) that do not include the start node.

The final node also requires some extra care. We require the goal, when it is found, to be contained within an interval of a search node. If the target is an interior point in the mesh, it will never be contained within an interval. To alleviate this, we explicitly generate the target as an additional successor when pushing into the polygon containing the target. The interval comprises only the target and the root is determined as with any other successor — either the same root if the target is observable, or an endpoint of the interval if the target is non-observable.

3.3 Evaluation Function

The purpose of the evaluation function is to determine which node should be expanded next. The evaluation function estimates the length of the optimal path that can be produced from a search node derived from the current node. This evaluation needs to be as high as possible (to prune nodes that are not promising) but, for soundness, it must always remain below the actual value of the optimal path.

A* search nodes are traditionally evaluated by a combination of the g- and h-values. The g-value is the cost of a concrete (possibly optimal) path from the point s to n while the h-value is the heuristic value, a lower-bound estimate on the remaining cost from n to t. The total cost of the partial solution is called the f-value and computed as $f = g + h$. **Polyanya**, similarly to **Anya**, uses the length of the path to root point r as g. For h we compute a minimum Euclidean distance, from r through I and then to the target. This distance underestimates the real length since it ignores any obstacles between r and t. Computing this heuristic is a simple matter of geometry, as illustrated in Figure 4:

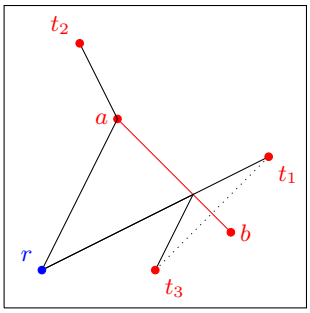

Figure 4: Evaluation of $([a, b], r)$.

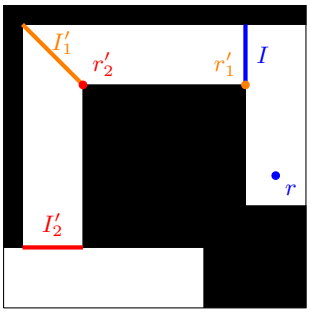

Figure 5: Intermediate nodes.

- if the target lies on the other side of I from r and in the cone formed by I from r (as for t_1), then the heuristic is the straight-line distance $d(r, t)$;
- if the target lies on the other side of I from r but on either side of the cone (as for t_2), then the heuristic is the minimal value between $d(r, a_I)+d(a_I, t)$ and $d(r, b_I)+d(b_I, t)$;
- if t lies on this side of I from r (as for t_3), then the heuristic value can be computed by using its mirror (here, t_1) through I.

3.4 Search and Pruning

Polyanya is a classical best-first search. However, unlike classical A*, **Polyanya** does not keep a closed list of all visited visited search nodes.

Termination: **Polyanya** terminates when a node containing the goal in its interval is expanded. If a path exists between the start and target, Lemma 4 (next section) shows that the target is eventually expanded by the search, so the search terminates in this case. If a path does not exist, **Polyanya** may endlessly generate search nodes with bigger and bigger g values if $\mathbf{M}$ contains a cycle. To avoid this, we keep a root history like **Anya** which stores the best g value for each visited root. Whenever a search node would be generated, we compare the g value with the stored history and discard the node if it is worse. With this root history, we can ensure that **Polyanya** is complete and always terminates with the same proofs used in [Harabor *et al.*, 2016]. It is also possible to check that a path exists between s and t by identifying the containing polygons Y_s and Y_t and checking if they belong to the same connected component in the navigation mesh graph.

The two pruning strategies detailed in [Harabor *et al.*, 2016] also translate to **Polyanya**, and can be improved:

Cul-de-sac Pruning: A cul-de-sac is any node which does not have any successors and does not contain the target. As expanding these will never reach the target, we can prune them away. If a node pushes into an obstacle, it is trivially a cul-de-sac as no successor can lie within an obstacle.

We identify a similar pruning technique which we call "dead-end pruning". A dead-end is a polygon which is only adjacent to one traversable polygon, and can be easily identified while reading in the mesh. If a node pushes into a dead-end that does not contain the target, we can prune it away as all the successors will be cul-de-sacs. Similar pruning techniques have previously been explored in the context of grid maps (see [Björnsson and Halldórsson, 2006]).

In Figure 3, the two successors $([f, g], r)$ and $([d, e], b)$ are pruned away as they are cul-de-sacs. The successor $([c, d], b)$ is also pruned away because it pushes into a dead-end.

Intermediate Pruning: If a node is expanded with a single observable successor, we can immediately expand that successor without pushing it onto the open list, provided it does not contain the target. The process can be repeated recursively until there is an increase in the branching factor (or until an obstacle is reached).

We generalise this technique by immediately expanding *any* single successor, including non-observable successors. In Figure 5, node (I, r) is expanded, begetting a single successor (I'_1, r'_1) which is immediately expanded. That begets a single successor (I'_2, r'_2), which is further expanded. A similar idea is described in [Harabor and Grastien, 2014].

4 Theoretical Properties

The proof for correctness of **Polyanya** mimics that of **Anya** [Harabor *et al.*, 2016], but we reproduce it here for self-containment. We prove that the optimal path always appears in the open list (Corollary 2); that the first search node expanded that contains the target represents the optimal path (Lemma 3); and that a search node is eventually expanded that includes the target (Lemma 4). The analysis assumes an optimal path always exists and we can always check this is the case using the procedure from Section 3.4.

Recall that a search node (I, r) represents the set of paths from the start s, to r (this part being described by the parents of the search node), then through a point of I, and finally to the target t. We say that (I, r) is a search node of those paths.

Lemma 1 *If $n = (I, r)$ is a search node of the optimal path then either I contains the target t or n has at least one successor n' that is also a search node of π^*.*

Proof: We assume the premise of the lemma and further assume that r does not appear in I. Let Y be the opposite polygon of (I, r). Then either the target appears in Y and the triangle inequality tells us that the suffix of the optimal path consists in moving straight to the target (this node is generated as mentioned in Section 3.2).

Otherwise the optimal path consists in exiting Y through a point p on one of its edges. If the optimal path requires turning on a point r' from I, then p is not visible from r (since a straight line would be shorter) and r' is a corner point at either extremity of I. Otherwise p is visible from r. Either way, a search node is generated that includes $p \in I'$ and the last turning point is updated accordingly. □

Corollary 2 *The open list always contains a search node of an optimal path (or this node is currently being processed).*

Proof: By induction: the initial search node is a search node of an optimal path; every time a search node of an optimal path is removed from the open list, another one is inserted (thanks to Lemma 1). □

Lemma 3 *The first expanded node that contains the target corresponds to an optimal path.*

Proof: Let f^* be the length of the optimal path. Assume that a node n is expanded that contains the target and that does

not correspond to an optimal path. Notice that the f-value of n is the length of the sub-optimal path ($f(n) > f^*$). From Corollary 2 we know that there is a node n' of the optimal path in the open list. Since the heuristic is admissible we know that $f(n') \leq f^*$ holds. Then $f(n') \leq f^* < f(n)$ contradicts the fact that n was chosen to expand next. □

Lemma 4 *A search node is eventually expanded that includes the target.*

Proof: Every search node has a bounded number of successors. Furthermore, only a finite number of successive search nodes can have the same g-value (when the corresponding intervals share the same root); otherwise the g-value increases by a non-trivial value (at least the smallest distance between two corner points). Therefore the g-value of the first node in the open list increases steadily and, eventually, reaches the value of the optimal path. At this point the algorithm will expand the search node corresponding to the optimal path. □

5 Different Types of Meshes

We assume throughout this work that the geometry of the environment is represented directly as a navigation mesh and given as input to the pathfinding system. In computer game applications navigation meshes can be created manually (by programmers and artists) or produced automatically from detailed scene geometry (e.g. using tools such as `Recast Navigation`). **Polyanya** is compatible with any type of planar mesh provided each polygon is convex.

We will see that different types of navigation meshes can have different strengths and weaknesses and two meshes that appear similar in principle can produce dramatically different search performance in practice. The problem of which mesh type to choose, and when to choose one type over another, is a non-trivial topic and beyond the scope of our present work (for a partial survey see [Kallmann and Kapadia, 2014]; for a comparative analysis of some recent works see [van Toll *et al.*, 2016]). Nonetheless in Section 6.1 we give experimental results for three simple types of meshes which we produced to test **Polyanya**. These are shown in Figure 2(b)-(d):

- **CDT**: a constrained Delaunay triangulation of the traversable space, created using the library `Fade2D`.
- **M-CDT**: we greedily merge adjacent polygons in a CDT mesh while trying to maximise area and retain convexity.
- **Rect**: we greedily construct a rectangle mesh from a grid map, always adding the largest possible rectangle.

Each type of mesh is very fast to construct, typically requiring a few seconds or less of computation. As the overhead is small we could, in principle, construct the mesh from scratch and online. In practice we compute the meshes apriori.

6 Empirical Analysis

We test **Polyanya** on a variety of realistic and synthetic grid benchmarks which are described in [Sturtevant, 2012]. We choose these problems because they are diverse, challenging and because their ubiquity in the literature makes for straightforward comparisons. All benchmarks are available from the HOG2 online repository [2]. In Table 1 we give a summary overview of the input meshes which we describe in Section 5 and which we construct from the grid benchmarks during an offline preprocessing step.

Maps	#	CDT			M-CDT			Rect		
		#P	#DE	Deg	#P	#DE	Deg	#P	#DE	Deg
BGII	75	1.3K	0.6K	2.0	0.8K	0.6K	2.0	0.6K	<0.1K	3.4
DA2	67	1.2K	0.4K	2.0	0.6K	0.4K	2.0	0.5K	0.1K	3.2
DAO	156	1.8K	0.6K	2.0	0.9K	0.6K	2.0	0.8K	0.1K	3.2
Maze	60	27.5K	3.4K	2.0	12.3K	3.4K	2.0	10.8K	2.1K	2.0
Rand	70	159.8K	23.8K	2.2	60.5K	23.8K	2.5	47.4K	7.5K	3.1
Room	40	13.6K	3.4K	2.1	6.9K	3.4K	2.2	3.5K	0.1K	2.4
SC1	75	11.5K	4.2K	2.0	6.4K	4.2K	2.0	5.2K	0.9K	3.3
WC3	36	2.2K	0.7K	2.0	1.1K	0.7K	2.0	0.8K	0.2K	3.0

Table 1: We construct and give metrics for a number of simple mesh types (see Section 5). #P = avg. num polygons; #DE = avg. num dead-end polygons (see Section 3.4); Deg = average degree.

Maps	Avg. Expansions				Avg. Speedup			
	Anya	Polyanya			Anya	Polyanya		
		CDT	M-CDT	Rect		CDT	M-CDT	Rect
BGII	79	67	**37**	131	19.9	40.2	**55.7**	23.7
DA2	228	261	**120**	388	2.5	10.3	**17.6**	7.0
DAO	956	908	**499**	1725	8.2	8.3	**13.5**	5.3
Maze	6633	7079	1708	**1614**	22.8	103	**158.2**	157.2
Rand	17476	30615	**14243**	17934	0.9	0.6	**1.0**	0.7
Room	1431	1862	895	**844**	10	63.8	106.0	**107.8**
SC1	1379	1444	**755**	2537	21	18.1	**29.4**	9.4
WC3	89	109	**55**	132	3.2	30.3	**43.7**	21.0

Table 2: **Anya** vs **Polyanya**. We measure average performance in terms of node expansions and search time speedup vs A*. Highlighted are best results for each metric and benchmark pair.

We implemented **Polyanya** in C++ and compiled our code with g++ 6.3.1 using `-O3`. In one of our experiments we contrast this algorithm against its direct progenitor, **Anya**, which is coded in Java. To account for differences across implementation languages we measure performance as *search time speedup*: i.e. relative improvement vs grid A*. We have written two versions of this algorithm, one in Java and the other in C++. Both versions make similar implementation choices, both are hand-optimised [3] and we believe them to be broadly comparable. All of our source code is publicly available[4]. All experiments are performed on a 1.7 GHz Intel Core i5 machine with 4GB of RAM and running Linux 4.8.13.

6.1 Anya

We begin with Table 2 where we compare **Polyanya** running on a variety of input meshes against its forebear, **Anya**. Both algorithms are online and optimal but **Anya** requires the traversable space be approximated using a fixed-resolution grid which is explored row-by-row. By comparison, **Polyanya** takes as input an exact convex partitioning of the traversable space and explores it polygon-by-polygon. We

[2] `https://github.com/nathansttt/hog2`

[3] We use bit-packed arrays to store the grid and closed list, preallocate all memory and avoid *sqrt* operations in the octile heuristic.

[4] `https://bitbucket.org/dharabor/pathfinding`

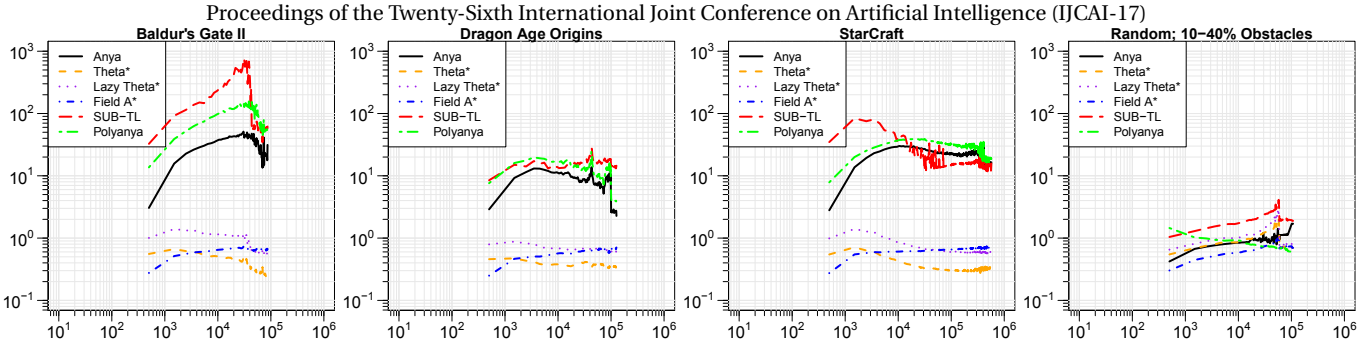

Figure 6: **Polyanya** vs. contemporary near-optimal algorithms developed for the Any-angle Pathfinding Problem. We measure performance vs A*. The x-axis shows how many nodes A* expands (on average) and the y-axis measures (average) speedup. Notice both axes are log10.

observe that while **Anya** can sometimes expand fewer nodes, **Polyanya** usually has a search time advantage. The best variant, **Polyanya**+M-CDT, outperforms **Anya** on both metrics; it can improve node expansions by up to several factors and search times by up to one order of magnitude, on average.

6.2 Approximate Any-angle Techniques

In Figure 6 we extend with additional data an apples-to-oranges experiment from [Harabor *et al.*, 2016] which compares **Anya** with a variety of contemporary algorithms for the Any-angle Pathfinding Problem (APP). APP is a special case of ESPP where the edges that comprise each obstacle are axis aligned and the traversable space can be represented exactly using a fixed resolution grid. The principal points of comparison are: Theta* and Lazy Theta* [Nash and Koenig, 2013], Field A* [Uras and Koenig, 2015a] and SUB-TL [Uras and Koenig, 2015b]. The first three algorithms are approximate and entirely online. SUB-TL assumes the map is static and performs additional pre-processing. All four methods produce approximately shortest paths.

The principal metric in this case is search-time speedup vs grid A*. We run the same experiment as [Harabor *et al.*, 2016] and give results for **Polyanya**+M-CDT. We observe that **Polyanya** outperforms all approximate and online techniques by a large margin. Compared to SUB-TL, **Polyanya** is often comparable and sometimes better.

6.3 TA* and TRA*

We reproduce an experiment from [Demyen and Buro, 2006] on the grid benchmark *Baldur's Gate II* and compare against published results for the algorithms TA* and TRA*. A contrast between these methods and our work is given in Section 1. Figure 7 (left) is taken directly from [Demyen and Buro, 2006] and shows relative improvement *medians* for TA* and TRA* vs. the authors' own reference implementation of grid A*. Figure 7 (right) shows our results with **Polyanya** on the same maps and with the same instances.

We observe that **Polyanya** (all variants) is competitive with or significantly better than the online algorithm TA*. In the best case, we improve performance by 2.5 times. Meanwhile TRA* appears faster still and by a similar margin. It is important to point out that the times reported for TA* and TRA* are for a single run of each algorithm (denoted by F=1). In [Demyen and Buro, 2006] the authors run each method *up to 10 times* to obtain optimal or close-to-optimal paths. **Polyanya**,

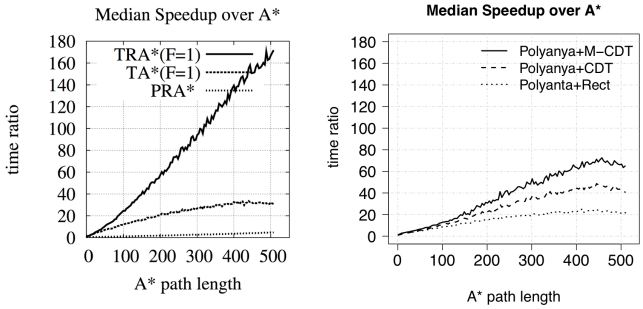

Figure 7: **Polyanya** vs. TA* and TRA* on benchmark BGII

by comparison, computes the optimal path in a single shot. It seems reasonable to assume that in a like-for-like setup the advantage **Polyanya** has vs. TA* will be magnified while its disadvantage vs TRA* will be at the very least reduced.

7 Conclusion

We introduce **Polyanya**, a new exact and online algorithm for solving the point-to-point Euclidean Shortest Path Problem (ESPP) on a navigation mesh. Contemporary algorithms for ESPP exist but all involve some form of compromise. For example, well known methods such as Theta* and TA* can solve ESPP instances entirely online only by trading optimality for speed (or vice versa). Alternatively, preprocessing-based algorithms such as TRA* and SUB-TL can solve ESPP instances up to two orders of magnitude faster but only under certain sometimes prohibitive assumptions that compromise flexibility (e.g. the map never changes). **Polyanya**, by comparison, is compromise free: it solves ESPP instances exactly, entirely online and very fast. In an empirical comparison we find that **Polyanya** is up to two orders faster than a range of approximate and online ESPP algorithms. We also show that **Polyanya** is often able to keep pace with, and sometimes outperform even preprocessing-based techniques.

There are many possibilities for further research. For example we believe the h-value heuristic can be modified in order to solve single-source multiple-target paths. Another possibility involves stronger pruning of redundant search nodes; e.g. when an interval is reached from two different root points. A third promising direction is to investigate how different mesh types impact the performance of search; e.g. getting the right balance between polygon size vs. degree.

References

[Björnsson and Halldórsson, 2006] Yngvi Björnsson and Kári Halldórsson. Improved Heuristics for Optimal Path-finding on Game Maps. In *Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference, AIIDE, June 20-23, Marina del Rey, California*, volume 6, pages 9–14, 2006.

[Demyen and Buro, 2006] Douglas Demyen and Michael Buro. Efficient Triangulation-Based Pathfinding. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence, AAAI, July 16-20, Boston, Massachusetts, USA*, pages 942–947, 2006.

[Harabor and Grastien, 2014] Daniel Damir Harabor and Alban Grastien. Improving Jump Point Search. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS, Portsmouth, New Hampshire, USA, June 21-26*, 2014.

[Harabor *et al.*, 2016] Daniel Harabor, Alban Grastien, Dindar Öz, and Vural Aksakalli. Optimal Any-angle Pathfinding in Practice. *Journal of Artificial Intelligence Research*, 56(1):89–118, May 2016.

[Kallmann and Kapadia, 2014] Marcelo Kallmann and Mubbasir Kapadia. Navigation Meshes and Real-time Dynamic Planning for Virtual Worlds. In *ACM SIGGRAPH 2014 Courses*, page 3. ACM, 2014.

[Kallmann *et al.*, 2004] Marcelo Kallmann, Hanspeter Bieri, and Daniel Thalmann. Fully Dynamic Constrained Delaunay Triangulations. In *Geometric Modeling for Scientific Visualization*, pages 241–257. Springer, 2004.

[Kallmann, 2005] Marcello Kallmann. Path Planning in Triangulations. In *IJCAI Workshop on Reasoning Representation and Learning in Computer Games*. 2005.

[Nash and Koenig, 2013] Alex Nash and Sven Koenig. Any-Angle Path Planning. *AI Magazine*, 34(4):9, 2013.

[Sturtevant, 2012] N. Sturtevant. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[Uras and Koenig, 2015a] Tansel Uras and Sven Koenig. 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*, pages 206–211, 2015.

[Uras and Koenig, 2015b] Tansel Uras and Sven Koenig. Speeding-Up Any-Angle Path-Planning on Grids. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS, Jerusalem, Israel, June 7-11*, pages 234–238, 2015.

[van Toll *et al.*, 2012] Wouter G. van Toll, Atlas F. Cook, and Roland Geraerts. A Navigation Mesh For Dynamic Environments. *Computer Animation and Virtual Worlds*, 23(6):535–546, 2012.

[van Toll *et al.*, 2016] Wouter G. van Toll, Roy Triesscheijn, Marcelo Kallmann, Ramon Oliva, Nuria Pelechano, Julien Pettré, and Roland Geraerts. A Comparative Study of Navigation Meshes. In *Proceedings of the 9th International Conference on Motion in Games*, MIG '16, pages 91–100, New York, NY, USA, 2016. ACM.