Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

Online Computation of Euclidean Shortest Paths in Two Dimensions

Ryan Hechenberger, Peter J Stuckey, Daniel Harabor,
Pierre Le Bodic, Muhammad Aamir Cheema
Faculty of Information Technology, Monash University, Australia
{Ryan.Hechenberger, Peter.Stuckey, Daniel.Harabor, Pierre.LeBodic, Aamir.Cheema}@monash.edu

Abstract

We consider the online version of Euclidean Shortest Path (ESP): a problem that asks for distance minimal trajectories between traversable pairs of points in the plane. The problem is made challenging because each trajectory must avoid a set of non-traversable obstacles represented as polygons. To solve ESP practitioners will often preprocess the environment and construct specialised data structures, such as visibility graphs and navigation meshes. Pathfinding queries on these data structures are fast but the preprocessed data becomes invalidated when obstacles move or change. In this work we propose RAYSCAN, a new algorithmic approach for ESP which is entirely online. The central idea is simple: each time we expand a node we also cast a ray toward the target. If an obstacle intersects the ray we scan its perimeter for a turning point; i.e. a vertex from which a new ray can continue unimpeded towards the target. RAYSCAN is fast, optimal and entirely online. Experiments show that it can significantly improve upon current state-of-the-art methods for ESP in cases where the set of obstacles is dynamic.

Introduction

Finding an obstacle-avoiding shortest path, in two dimensions, is an often studied problem with many practical applications. In robotics and computer games, for example, it is desirable to compute paths that are as short as possible and as quickly as possible. Quickly, because computational resources (CPU, memory) are often restricted. Short, because such paths minimise travel distance and because they satisfy other more subjective criteria. For example, a human observing the execution of a path planning agent may consider that agent to be "more intelligent" if its planned trajectory is free from detours. Another important aspect for path planning is the ability to account for dynamic changes that occur in the environment (e.g. a fallen tree blocks a path or a player adds or destroys a building in a computer game). We categorise works from this area into three distinct (and overlapping) categories: any-angle techniques, visibility graphs and mesh-based navigation.

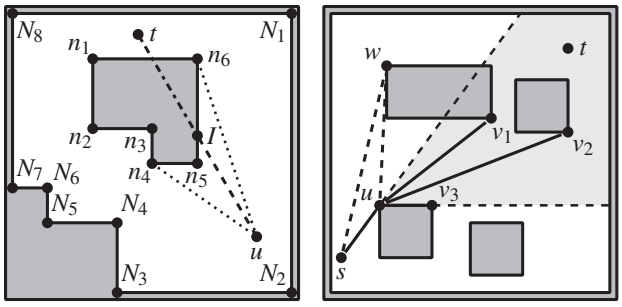
Any-angle techniques discretise the set of obstacles using a fixed resolution grid but then proceed to find so-called any-angle paths, which are not restricted to the points of the grid (Daniel et al. 2010). Any-angle paths are desirable because they can be substantially shorter than their grid-optimal counterparts. State of the art works in this area, such as Anya (Harabor and Grastien 2013), compute optimal any-angle paths quickly and online, where we consider online to be with little setup overhead. In the event of dynamic changes, the grid map can be updated in constant time. The main drawback to any-angle pathfinding is that obstacles must be rasterised approximately using the fixed resolution grid. This leads to suboptimality in practice, where the grid approximation and the reality mismatch. Meanwhile, increasing the resolution of the grid slows search.

Visibility Graphs (VGs) are another family of well known techniques that work by preprocessing the environment to obtain a graph of co-visible points (Lozano-Pérez and Wesley 1979). Using such a graph allows obstacles to be represented accurately, eliminating mismatch and allowing Euclidean Shortest Paths to be computed precisely. The main drawback is twofold: (i) VGs require an often prohibitive precomputation step and; (ii) updates to the environment invalidate the graph. Repair or recomputation times for VGs are, again, often prohibitive (Hong, Murray, and Wolf 2016).

Mesh-based planners are a family of techniques which have the combined strengths of any-angle path planning and VGs. Here the environment is lightly preprocessed to obtain a "navigation mesh"; a data structure that comprises a set of convex non-overlapping traversable polygons[1]. State of the art works such as Polyanya (Cui, Harabor, and Grastien 2017) compute paths extremely fast and they allow the negative space around obstacles to be represented exactly. This means optimal paths on the mesh are guaranteed to be shortest paths in practice. Meanwhile, dynamic changes are handled online, via localised repair of the navigation mesh. A main drawback is that repair operations typically require more than constant time (van Toll, Cook IV, and Geraerts 2012). Thus, depending on the set of dynamic changes, and their frequency, the cost of repairing a navigation mesh may come to dominate the total running time of the algorithm.

In this work we propose **RAYSCAN**, a new approach for computing Euclidean Shortest Paths. Unlike existing tech-

[1] In some applications (e.g. games), the mesh may be constructed by hand. Such planners are therefore "preprocessing-free".



(a) Points order (b) Projection field

Figure 1: (a) Points $n_1, n_2, \cdots, n_6$ orientate CCW. Points $N_1, N_2, \cdots, N_8$ orientate CW. (b) Projection field for u is $\mathrm{AS}(\vec{su}, \vec{uv_3})$.

niques, RAYSCAN does not depend on any pre-computation of the environment: given a set of obstacles it computes directly and in an online fashion a Euclidean Shortest Path for a distinguished pair of traversable start and target positions. The first ingredient is **ray shooting**: the algorithm follows a direct trajectory from a specified location to the target node, stopping only if an obstacle impedes the ray. The second ingredient is **scanning**. Once an obstacle is detected the algorithm scans its perimeter looking for a traversable point from which a new ray can continue, unimpeded toward the target. We give a thorough description of the algorithm and we provide proofs for correctness and optimality. In an empirical evaluation we compare RAYSCAN to Polyanya (Cui, Harabor, and Grastien 2017), the most performant ESP technique of which we are aware. Results show that RAYSCAN improves on this leading algorithm in cases where the environment regularly changes and where the Polyanya navigation mesh must be continuously repaired.

Overview of RAYSCAN

RAYSCAN is presented in this paper as a 2D ESP, made to work with a 2D environment, represented as set of non-intersecting polygonal obstacles (**inner-obstacles**), and a single enclosing polygon (**enclosure** or **outer-obstacle**), where all inner-obstacles are within and non-intersecting. The **outer-boundary** is defined as the convex hull of the enclosure.

RAYSCAN does not provide the method of searching, rather it can be view as a fast method of producing a subset of edges for a VGs during the search. This paper uses the A* algorithm (Hart, Nilsson, and Raphael 1968) to drive the search, using **start** (s), **target** (t) and points on the polygons (like VGs) as the nodes in the search. When pushing a successor edge $\vec{uv}$, the edge weight is the Euclidean distance from u to v, $|\vec{uv}|$.

During the search, we expand nodes by producing successors. The expanding node will always be denoted by u throughout the paper.

The search is directed with the heuristic function $\mathrm{h}(v) = |\vec{vt}|$. The minimum f value would be the next node to expand, which is given as $\mathrm{f}(v) = \mathrm{g}(u) + \mathrm{h}(v)$, where $\mathrm{g}(u)$ is the g value (the shortest path to u).

The search is directed by ray shooting, where a ray is shot from u along a direction vector, then returns the first polygon hit and its intersection. This ray gives us an obstructing obstacle that we need to navigate around, which in a 2D environment, is restricted to two ways, clockwise (CW) or counter-clockwise (CCW) around the obstruction. This presents the second concept, the **scan** phase, where we trace a scan line along the polygon in a CW or CCW direction w.r.t. u. e.g. a CW-scan will have a scan line from u, starting at the intersecting point of a ray, we will follow the edges of the polygon in a CW direction around u. The scan will continue until the scan line reaches a point where continuing the scan will force the line to rotate CCW, which we designate as a **turning point**. If there are no obstacles between u and a turning point, it is also a **visible point** and may be a successor node to the search, otherwise it is a **blocked point**.

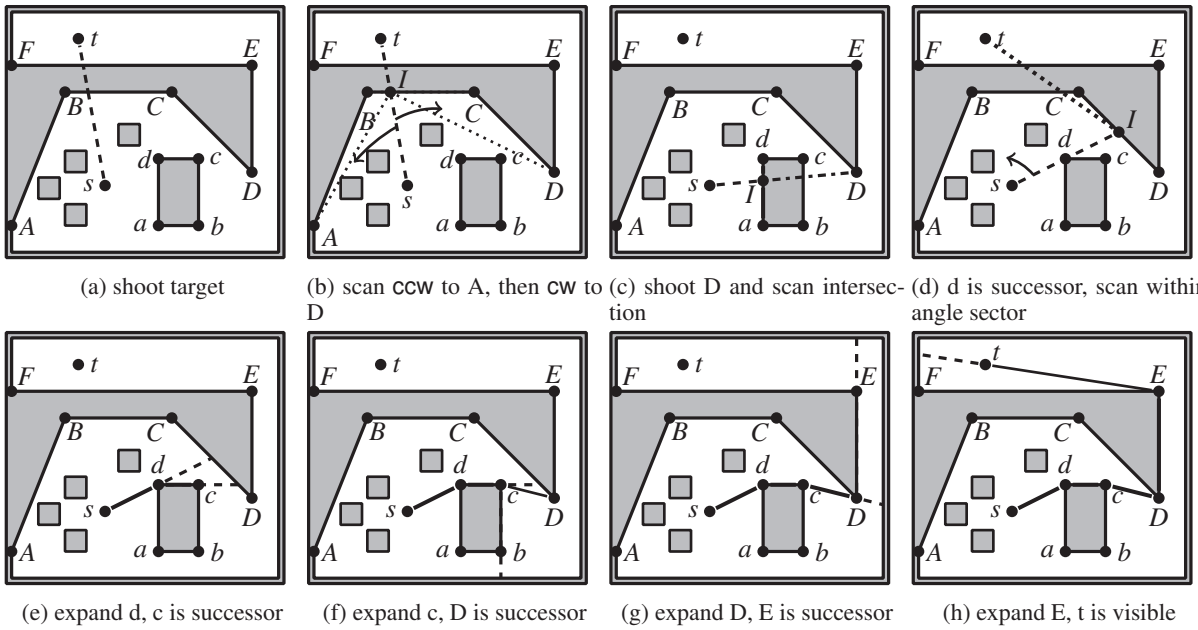
Since a scan is made up of many other recursive scan, the whole process of this scan starting from the initial scan is called a **full scan**. This is relevant since the generation of successors for any node will perform multiple full scans.

The order of the points in the polygon obstacles are relevant to be able to scan efficiently. In this paper, the inner-obstacles points have a CCW orientation, while the enclosure has a CW orientation. Refer to Figure 1a, node u can scan CCW from intersection I to n_5 and n_4 (points decreasing), although to scan further will result in a CW direction w.r.t. u, thus n_4 is the turning point. The CW scan will find points increasing and end at the turning point n_6. The enclosure points are ordered the opposite orientation in order to maintain the decreasing index for CCW scan and increasing for CW, since we are following the polygon from the inside.

A point on a polygon is considered to be a **concave point** if its inside angle is more than $180°$ (e.g. n_3), or **convex point** otherwise (e.g. n_1, N_4). Expanding nodes, with the possible exception of s, would always be convex.

The **angled sector** or **sector** is defined from the expanding node u. We define a region by two angles (can be represented by vectors), from the CCW angle a_{ccw} turning CW to the CW angle a_{cw}. Let angle sector be defined as $\mathrm{AS}(a_{\mathsf{ccw}}, a_{\mathsf{cw}})$, take the example from Figure 1a, the angled sector $\mathrm{AS}(\vec{un_4}, \vec{un_6})$ from an expanding u shows that points like t, n_3 and n_6 lay within this sector (as ray $\vec{ut}$ is on or between $\vec{un_4}$ to $\vec{un_6}$) but points n_2 and N_6 are outside the sector. If a_{ccw} and a_{cw} are the same angle, than it is treated as any point p whose angle $\vec{up}$ are equal are inside the sector, otherwise all points are outside. The angle sectors restrict the scan space. Every scan has an associated angle sector, where if its scan line ever leaves the angle sector, the scan ends with no turning point been found. A scan can also end when we find a point on the outer-boundary.

Every expanding node has a special angle sector know as the **projection field**. The projection field for s is a special type of angle sector known as a whole sector $\mathrm{WAS}(a)$, which considers that all angles falls within. All other expansion projection fields are formed based of its predecessor point $P(u)$. For example, in Figure 1b the shaded region is u's projection field.



(a) shoot target (b) scan CCW to A, then CW to D (c) shoot D and scan intersection (d) d is successor, scan within angle sector

(e) expand d, c is successor (f) expand c, D is successor (g) expand D, E is successor (h) expand E, t is visible

Figure 2: RAYSCAN Example - The dashed line is a ray shot, it turns dash dot line after intersecting with a polygon. Each corner scanned by RAYSCAN is shown by a dotted line from s to the corner. The solid lines are edges on the shortest path.

RAYSCAN by Example

Figure 2a: consider the shortest path query from s to t shown. The first thing we should do in searching for a shortest path from s to t is determine if t is visible from s. Hence we shoot a ray from s to t. If this is unblocked we are finished. Otherwise we will hit an obstacle (actually the enclosure) as illustrated, the edge (B, C).

Figure 2b: we must go around the obstacle. We first identify the projection field of s to be $\mathrm{WAS}(\vec{st})$. We then scan CW or CCW from intersection I. The scan looks for a turning point on the obstacle for a path from the expanding node s.

Scanning CCW, we find no turning point before we reach A, which is a point on the outer-boundary. We know that when a scan reaches an outer-boundary point, there is no way around the obstacle, so end the scan with no turning point.

We then scan CW and find the turning point D, since tracing from D to E would result in the scan going CCW.

Figure 2c: we shoot a ray from s to our turning point D and find that it is blocked by (d, a), thus is not a visible point.

For a non-visible point, we will try to find a path that can lead to it, so we do as we did with t and scan CW and CCW around the obstacle edge (d, a). This however requires a slight alteration of the sector to prevent a possible looping of scans. The modification will be a split along the ray (this case $s\vec{D}$) of the current sector, such that a CW scan will hold the CW half, while CCW the CCW half. Since we currently have a whole sector, we will split it up to the original ray $\vec{st}$, thus the CCW scan will use $\mathrm{AS}(\vec{st}, s\vec{D})$ and the CW scan $\mathrm{AS}(s\vec{D}, \vec{st})$.

We begin a scan on this obstacle in the CCW direction, which is reversed from the scan that lead to D. We find the turning point d and shoot a ray to d, finding it is a visible point. Point d must be a neighbour of s in the VGs and is pushed onto the A* queue as a successor of s. Next instead of doing the CW scan, we will continue scanning this sector, as this method scans recursively.

Figure 2d: since we found a visible point, we have no obstacle to navigate around. For this case, we continue the ray past d and hit the edge (C, D). We will only do a scan in the same direction as the scan that found d (CCW), as doing a CW scan here will have us search within the sector $AS(\vec{sd}, s\vec{D})$, which is blocked by the obstacle we just scanned.

Therefore we only scan CCW, using the CCW split along $\vec{sd}$ to give the sector $\mathrm{AS}(\vec{st}, \vec{sd})$. This scan will result in the sweep line leaving the sector when crossing from C to B, thus the scan ends with no turning point found.

The scan recurses back to do the CW scan from $s\vec{D}$, finding the visible point a and adds it to the queue. The ray itself hits an edge on the outer-boundary, where a continuation scan CW will reach the point on the outer-boundary and end, thus the recursion recurses back to the start $\vec{st}$ which has already scanned both directions, thus the expansion of s is complete.

Figure 2e: now we pop the lowest f value point of the queue, d. Since we are only interested in paths that *turn* we will restrict the region the scan occurs to its projection field $\mathrm{AS}(\vec{sd}, \vec{dc})$. We first check if t falls within the projection field of d, which $\vec{dt}$ does not, thus we cannot shoot towards the target. There are two other avenues to reach t from d, to turn around its own polygon (along $\vec{dc}$) or to turn out of its projection field (along $\vec{sd}$). We will need to check both.

We shoot a ray from d in the direction $\vec{sd}$ (the projection field's a_{CCW}), getting blocked by (C, D) and scan CW with

the projection filed, finding no turning point before leaving the sector. We do not need to scan CCW since that will immediately leave the sector. We shoot a ray $\vec{dc}$ and hit not an edge, but a point c. At this case we need to check if c can be a successor from d and find that it can, therefore we add c as a successor. The ray continues past c and get blocked by edge (C, D), thus we scan CCW and find no turning point.

Next we pop a off the queue, its scan adds b to the queue but nothing else.

Figure 2f: when c is popped off the queue, it has the projection field $AS(\vec{dc}, \vec{cb})$. Following the process from d, we find that $\vec{ct}$ is not within the projection field.

We start a scan to turn out of the projection field by shooting the ray from d along its a_{CCW} $\vec{dc}$, blocked by the edge (C, D). Scanning CW we reach the visible point D and add it as a successor of c. The recursion hits the outer-boundary and thus ends.

The scan to turn around c's obstacle occurs in much the same way as d did, thus the ray $\vec{cb}$ finds b as a successor then ends due to encountering the outer-boundary.

Figure 2g: we pop D off the queue, and expand it. We do not shoot to target due to being outside its projection field $\text{AS}(\vec{DE}, \vec{cD})$. Shooting a ray $\vec{DE}$ will find E as a successor and then end, then shooting the ray $\vec{cD}$ will hit the outer-boundary and also end.

The next lowest f value will be b. When following the same process, it will find point c and D, though these will not be added as b's successors due to its f value being higher than their current ones.

Figure 2h: the final node to expand is E. The target ray $\vec{Et}$ falls within the projection field $\text{AS}(\vec{EF}, \vec{DE})$, thus we shoot the ray $\vec{Et}$. Since t is visible, there is no need to find any other successors.

We can say we have now found the shortest path, since with our heuristic, $\text{g}(t) = \text{f}(E)$.

RAYSCAN

RAYSCAN expands nodes by generating the successors on-the-fly, as detailed in Algorithm 1.

The function START_SUCCESSORS (line 1) is the initial call. The first action is to shoot a ray from s to t (line 2), which gives up the blocking polygon p and its intersection point to the ray I. If t is visible, then the optimal path has been found (line 4). Otherwise, we have an obstacle impeding t, thus we need to do a full scan CW and then CCW of p from the intersection I (lines 7 to 8). These scans use $\text{WAS}(\vec{ut})$ representing the 360° angle sector, which is a special case of an $\text{AS}(\vec{ut}, \vec{ut})$ except instead of only vectors angled to $\vec{ut}$ falling within, it allows all angles.

The SHOOTRAY(u, a) function handles the shooting of rays. It shoots a ray from point u in direction a, adding a potential successor and returning the polygon p and intersection point I with the polygon that eventually blocks the ray. Note that all rays will hit something because of the enclosure. An example of a ray shoot is shown in Figure 3, points v_i signify collinear nodes that are hit by the ray but are not blocking it, which is handled by lines 25 to 28. Out of

Algorithm 1 Generate u's successors within projection field

```
 1: function START_SUCCESSORS
 2:     (p, I) ← SHOOTRAY(s, st)
 3:     if t is visible from s then
 4:         return optimal path to t
 5:     end if
 6:     F ← WAS(st)
 7:     SCAN(s, p, I, F, CW)
 8:     SCAN(s, p, I, F, CCW)
 9: end function
10: function SUCCESSORS(u, F = AS(a_ccw, a_cw))
11:     if ut is within AS(a_ccw, a_cw) then
12:         (p, I) ←SHOOTRAY(u, ut)
13:         if t is visible from u then
14:             return optimal path to t
15:         end if
16:         SCAN(u, p, I, AS(ut, a_cw), CW)
17:         SCAN(u, p, I, AS(a_ccw, ut), CCW)
18:     end if
19:     (p, I) ← SHOOTRAY(u, a_ccw)
20:     SCAN(u, p, I, F, CW)
21:     (p, I) ← SHOOTRAY(u, a_cw)
22:     SCAN(u, p, I, F, CCW)
23: end function
24: function SHOOTRAY(u, a)
25:     if ray hits one or more turning points then
26:         n ← first turning point n hit
27:         PUSH_SUCCESSOR(u, n)
28:     end if
29:     Let (p, I) be the first polygon p blocking the ray
30:             at intersection I
31:     return (p, I)
32: end function
33: function SCAN(u, p, I, F = AS(a_ccw, a_cw), d)
34:     scan p from I in direction d to
35:             find a turning point n
36:     if scan goes out of F or
37:             touches part of the outer-boundary then
38:         return
39:     end if
40:     (p', I') ←SHOOTRAY(u, un)
41:     if n is visible from u then
42:         if d = CW then
43:             SCAN(u, p', I', AS(un, a_cw), CW)
44:         else
45:             SCAN(u, p', I', AS(a_ccw, un), CCW)
46:         end if
47:     else
48:         SCAN(u, p', I', AS(un, a_cw), CW)
49:         SCAN(u, p', I', AS(a_ccw, un), CCW)
50:     end if
51: end function
```

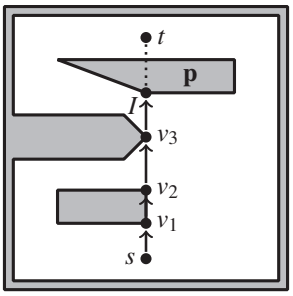


Figure 3: Ray shot on collinear points

these points only the closest point (v_1 in this case) is added (line 26). The function returns the blocking polygon p and the point of intersection I with this polygon (line 31).

The SUCCESSORS(u, $F = \mathrm{AS}(a_{\mathsf{ccw}}, a_{\mathsf{cw}})$) function is used to generate successors for any expanding node u (excluding s) which fall within the projection field $F = \mathrm{AS}(a_{\mathsf{ccw}}, a_{\mathsf{cw}})$. Like START_SUCCESSORS, we first shoot a ray at the target and scan if blocked (lines 11-18), except this is only done if the target ray $\vec{ut}$ falls within F. The scan also splits F along the target ray $\vec{ut}$, where we use the CW split to scan CW and the CCW split for the CCW scan.

After handling the target, if t was not visible, then we need to shoot a ray towards both extremities of the projection field and then scan inwards. This will find the successors within the field that bends around obstacles that could lead out of the projection field in both orientations. First, we shoot a ray towards the CCW extremity (line 19) and scan CW direction from its intersection (line 20). Then, we shoot a ray towards CW extremity and scan CCW (lines 21 and 22). Each of these scans potentially adds entries to the queue.

The call SCAN(u,p,I,$F = \mathrm{AS}(a_{\mathsf{ccw}}, a_{\mathsf{cw}})$,$d$) scans along polygon p in direction d starting from intersection point I on the polygon, remaining within the projection field F. We skip along the polygon p until we find a turning point n (lines 34 and 35). If the scan from I to n at any point is not within F (line 36), then the scan halts without finding any turning point. The scan also halts when it touches any part of the outer-boundary (line 37), as we cannot "bend around" parts of the outer-boundary.

When we find a turning point n, we shoot a ray to this point (line 40), which will determine if n is a visible point from u, which is a point of interest. Remember, SHOOTRAY will only add n as a successor if it is the first collinear point hit, e.g. $n = v_1$ is added from s, $n = v_3$ however will not, even though both points are visible. Regardless of whether n is visible or not, SHOOTRAY finds the polygon p' that blocks the ray and returns it. We then need to search for more successors; if n is visible, then we want to continue the scan in the same direction (lines 42–46). This will continue the scan from the blocking polygon p' that lies past n, although sector F is split along $\vec{un}$ and only the d part of the split is taken for the scan, as we have already scanned the other side. If n is blocked, then F is still split but we will scan both sides, recursing the scan CW and CCW (lines 48–49).

Correctness

We begin with an informal argument about correctness of the algorithm. First of all, all shortest paths in the VGs only ever bend around obstacles (see e.g., (Rohnert 1986)). Hence, the restriction to looking for visible turning points in the projection field of a node is valid.

The algorithm tries to go directly to the target if possible. If it hits an obstacle then it will try to go around the obstacle by scanning in each direction for possible turning points.

The on-the-fly generation of part of the VGs does not generate all possible VGs entries for node u that fall within its projection field F. There are three cases to consider.

If t falls within F then we shoot a ray towards t. If t is not visible from u blocked by a polygon p, we scan to go around the obstacle p either clockwise or counter-clockwise. Note that we will never directly consider obstacles visible from u in F which are not within the angle sectors to the points $\mathrm{AS}(\vec{ut}, a_{\mathsf{cw}})$ and $\mathrm{AS}(a_{\mathsf{ccw}}, \vec{ut})$. This is safe since no such obstacle can be involved in a shortest path to t from u unless one of the direct edges to n_{cw} or n_{ccw} is blocked.

If the edges are blocked then we perform a reverse scan to find a way to try to reach one of the turning points. For example in Figure 4b, the ray $\vec{ut}$ is blocked by an edge of the polygon p_c and the two scans lead to its endpoints n_1 and n_2 which are also not visible from u (blocked by p_b and p_d, respectively). The recursive reverse scans on p_b and p_d find the two points x_4 and x_5 which are turning points that will allow us to reach the endpoints n_1 and n_2.

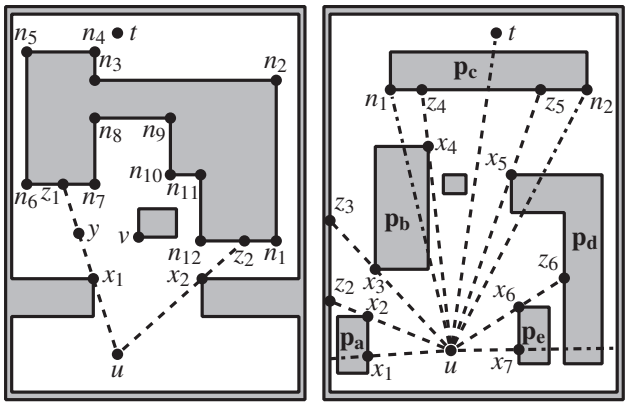
Similarly the shortest path may not be direct to t, then we shoot rays along the extremes of the projection field a_{cw} and a_{ccw}. The scans starting from these ray shoots may terminate without crossing if they both hit the outer-boundary. In this case objects in the projection field F or u which lead to entries in the VGs are ignored. This idea of shooting ray along the extremes is that we are trying to find valid successors that will lead to a path leaving the projection field, which is especially important for when t does not fall within the projection field, though is still required regardless.

The visible points within the projection field that are not found as successors we will prove cannot possibility be part of any optimal path. The idea behind this is that visible points are first identified as turning points from a scan, which requires a ray to hit the edge in question that will lead to the turning point. In other words, the discovery of an optimal successor is dependent on a ray shooting to direct the search.

As consequence of this, these successors not found may be discovered further on in the search, which would result in a non-optimal path to such successors. The guarantee is that no nodes on a shortest path to the target will have this property.

We now formally argue the correctness of the algorithm.

Given a node u, we say that a polygon p is visible from u along a point x_i if the ray $u\vec{x}_i$ hits the polygon p. Figure 4a shows a polygon p with vertices n_1 to n_{12}. It is visible along the points x_1 and x_2 (see the dashed rays). The polygon p is not visible along n_1 or v. We denote the point where the ray $u\vec{x}_i$ hits p as z_i (see z_1 and z_2). We define a *futile* polygon $FP_{<u,x_1,x_2,p>}$ of the form $u, z_1, n_i, n_{i+1}, \cdots, n_m, z_2$



(a) Futile Polygon (b) Lemma 2

Figure 4

where each n_i is a vertex of p. In Figure 4a, the futile polygon is $u, z_1, n_7, n_8, \cdots, n_{12}, z_2$.

Lemma 1. *Let p be a polygon which is visible from u along two points x_1 and x_2. If the target t lies outside of the futile polygon $FP_{<u,x_1,x_2,p>}$ then the shortest path from u to t cannot pass through any point v strictly inside this futile polygon.*

Proof. We prove this by contradiction. Assume that the shortest path from u to t passes through a point v strictly inside the futile polygon. Such a path must cross either uz_1 or uz_2. Without loss of generality, assume that the path crosses uz_1 at a point y. Then, uy is shorter than uvy which contradicts that the shortest path passes through v. □

Lemma 1 implies that, when t is outside the futile polygon of u, we do not need to consider successors of u that are strictly inside the futile polygon. Before we present the next lemma, we explain another notation. Note that there is at least one polygon that is visible from u along each x_i (is hit by the ray $u\vec{x}_i$). We use p_i^{cw} to refer to the *first* polygon that is touched or hit by $u\vec{x}_i$ *and* extends in the CW direction of x_i. Similarly, we use p_i^{ccw} to refer to the *first* polygon that is touched or hit by $u\vec{x}_i$ *and* extends in the CCW direction of x_i. Consider the example of Figure 4b that shows execution of SUCCESSORS(u,F) where the projection field is between $u\vec{x}_1$ and $u\vec{x}_7$. For point x_4, the ray $u\vec{x}_4$ hits two polygons p_b and p_c. The first polygon that is hit by this ray and extends in CW direction of x_4 is p_c. Thus, p_4^{cw} is p_c whereas p_4^{ccw} is p_b. Similarly for x_3, p_3^{cw} is p_b and p_3^{ccw} is the enclosure (which is hit at point z_3 by $u\vec{x}_3$). For x_1, p_1^{cw} and p_1^{ccw} both are p_a.

Lemma 2. *Let S be the set of successors pushed by the call* SUCCESSORS*(u,F). Furthermore, let e_{ccw} and e_{cw} be two extreme points (not necessarily vertices of any polygon) hit by the rays towards the CCW and CW extremes of F, respectively. Let $X = S \cup e_{ccw} \cup e_{cw}$ be sorted in clock-wise order and numbered 1 to m such that $x_1 = e_{ccw}$ and $x_m = e_{cw}$ (e.g., see x_1 to x_7 in Figure 4b). For every $i < m$, there exists a polygon p that is visible along both x_i and x_{i+1}.*

Proof. x_i and x_{i+1} will never be collinear w.r.t. u, as SHOOTRAY(u, a) returns only the first turning point hit by the ray a. Thus, we only need to prove for the case assuming X does not contain such collinear points. Next, we show that p_i^{cw} is visible along both x_i and x_{i+1}.

Case 1: (x_{i+1} is on p_i^{cw}). See $x_i = x_3$, $x_{i+1} = x_4$ and p_3^{cw}=p_b in Fig. 4b representing this case. Since x_{i+1} is visible and is on p_i^{cw}, p_i^{cw} is visible along both x_i and x_{i+1}.

Case 2: (x_{i+1} is not on p_i^{cw}). See $x_i = x_4$, $x_{i+1} = x_5$ and p_4^{cw}=p_c in Fig. 4b representing this case. We use $u\vec{n}$ to denote a ray by which the algorithm discovered p_i^{cw}, i.e., $\vec{ut}$ discovers p_c in Fig. 4b. Since x_{i+1} is not a point on p_i^{cw}, x_{i+1} must be on a polygon p that blocks p_i^{cw}. Note that p cannot extend beyond x_{i+1} in CCW direction. Otherwise, either p will generate a successor in CCW direction which is not possible (because x_i and x_{i+1} are consecutive successors in sorted order) or p extends beyond $u\vec{n}$ in CCW order which is also not possible because the ray $u\vec{n}$ hits p_i^{cw} and is not blocked by any polygon p. Since p does not extend beyond x_{i+1} in CCW direction, x_{i+1} must have been discovered by the algorithm while scanning p in CCW (from some point on p that lies in CW of x_{i+1}). Recall that, for each visible point x_{i+1}, SHOOTRAY(u, $u\vec{x_{i+1}}$) returns an intersection on a polygon p' that obstructs the ray $u\vec{x_{i+1}}$ (line 40 in Algorithm 1) and continues scanning p' in the same direction (line 42 to 46 in Algorithm 1), i.e., CCW direction in this case. Thus, the algorithm continues scanning p' in CCW direction. We prove that $p' = p_i^{\mathsf{cw}}$ which proves p_i^{cw} is visible along both x_i and x_{i+1} (because p' is visible along x_{i+1}). Note that no visible turning point can be found by scanning p' in CCW direction as x_i and x_{i+1} are consecutive successors. Furthermore, the scan and potential recursive scans resulting from it terminate only if a scan extends beyond the projection field (i.e., beyond $u\vec{n}$ in CCW direction) or if it touches the outer-boundary. The scan cannot extend beyond $u\vec{n}$ in CCW direction because it would imply that p' or some other obstructing polygon p'' from the recursive scans blocks p_i^{cw} for the ray $u\vec{n}$ which is not true. If the scan touches the outer-boundary, it implies that p' is the outer-obstacle which cannot obstruct p_i^{cw} unless p_i^{cw} itself is the outer-obstacle. Thus, p' must be p_i^{cw}. □

Lemma 3. *Let S be the set of successors pushed by the call* SUCCESSORS*(u,F). Let $v \notin S$ be a node in the complete VGs that lies within the projection field F. The shortest path from u to t does not need to consider v.*

Proof. Let X be as defined in Lemma 2. Since v lies within the projection field, it must lie between x_i and x_{i+1} for some i. As per Lemma 2, there exists a polygon p which is visible along both x_i and x_{i+1}. Since v is visible from u and lies between x_i and x_{i+1}, it must lie within the futile polygon $FP_{<u,x_i,x_{i+1},p>}$. Furthermore, if t lies inside the projection field F and is visible from u, it is already added as a successor of u (thus uvt is no shorter than ut). Otherwise, t must lie outside of this futile polygon. Therefore, v cannot be on the shortest path from u to t (Lemma 1). □

Theorem 1. RAYSCAN *returns an optimal path from s to t.*

Proof. The initial scan from s finds all successors that could be on a path to t. The proof is analogous to Lemma 3 where the projection field is the whole space. Furthermore, we know (Rohnert 1986) that we only have to consider successors in the projection field of a node u on any optimal path. Lemma 3 implies that SUCCESSORS finds all successors of a node u within the projection field that could be part of the shortest path to t. Since the algorithm always pushes a sufficient set of the successors of the entire VGs and adds correct successors for s, it explores any optimal path possible in the VGs, and hence finds the optimal path. □

Experimental Results

We compare the RAYSCAN approach against the state-of-the-art Euclidean pathfinding method Polyanya (Cui, Harabor, and Grastien 2017). Polyanya is the fastest method we are aware of for optimally solving ESP with no precomputing required (except for creating the mesh).

The tests were conducted using the Moving AI 2D pathfinding benchmarks for grids (Sturtevant 2012). These benchmarks include many groups of tests, separated by games, along with random, mazes and street maps. The maps contain an octile grid with blocked and traversable cells, along with a scenario file filled with hundreds or thousands of scenarios, each with a start and target point specified by Cartesian coordinates. For our instances, these are treated as points in the bottom-left corner of each cell.

To run RAYSCAN on these benchmarks, the blocked cells were traced to construct a set of inner-obstacles and enclosure. If there are multiple enclosures found, then the largest by area is kept while discarding the rest. Afterwards any polygon outside this enclosing are removed, along with any polygon inside another, which can occur when an enclosing polygon was within another.

If two obstacles touch each other at a point (see D6 in Figure 5a), it is ambiguous if passing through this touching point is allowed or not. To avoid such ambiguous cases, we cut a cell for every concave corner of each polygon. E.g., see the dashed lines in Figure 5a. This changes the input polygons and generates smoother looking polygons, e.g., the stairs from D0 to H4 are changed to a diagonal line and the new polygon is the area shown shaded (light and dark). Consequently, some s and t instances in the benchmarks may lie in an invalid location, i.e., inside a modified obstacle or outside the enclosed polygon. These instances are removed.

The ray casting method used by RAYSCAN makes use of spatial filtering with a modified Bresenham's algorithm (Bresenham 1965). We fill in a square grid with the polygon lines using Bresenham's. When we cast a ray, we draw a line from its start in the ray's direction, performing line intersection tests of all polygon lines found from the filled in cells it passes through.

Polyanya is given as input the same as RAYSCAN. Since Polyanya requires a navigation mesh, the polygon obstacles were converted using modified code from (Cui, Harabor, and Grastien 2017), which also makes use of `Fade2D`[2] for a Constrained Delaunay Triangulation, generating the triangles needed for the navigation mesh.

The experiments were all run on on a single core of an Intel Core i7-8750H fixed at 2.2 GHz, with 16 GB of RAM. Each instance ran six times, dropping the minimum and maximum search times, then averaged.

Comparison

We compare RAYSCAN against Polyanya and base our results on how the search benefits from lower setup cost and how speeding up the ray shooting will result in a fast search.

Figures 5b and 5c compare the search time of 200 instances of the map arena2 and Aurora, which are picked from equidistant in order as was given in the Moving AI benchmarks for the maps.

These figures directly compare Polyanya time (x-axis) to RAYSCAN time (y-axis) for each instance. A dividing line is added to show how they compare, where above the line Polyanya runs faster, while below RAYSCAN does. These results shows that while our method is running slower while using a Bresenham's line to shoot our rays, these runtimes are still within the low milliseconds, which is much less time then constructing a navigation mesh.

We have a second comparisons (the triangles), which uses an oracle to shoot the ray. This acts as a baseline to how fast RAYSCAN could reach if ray shooting was cheaper to do. These fall below the lines with these examples while others shows it along the line (thus as good as Polyanya).

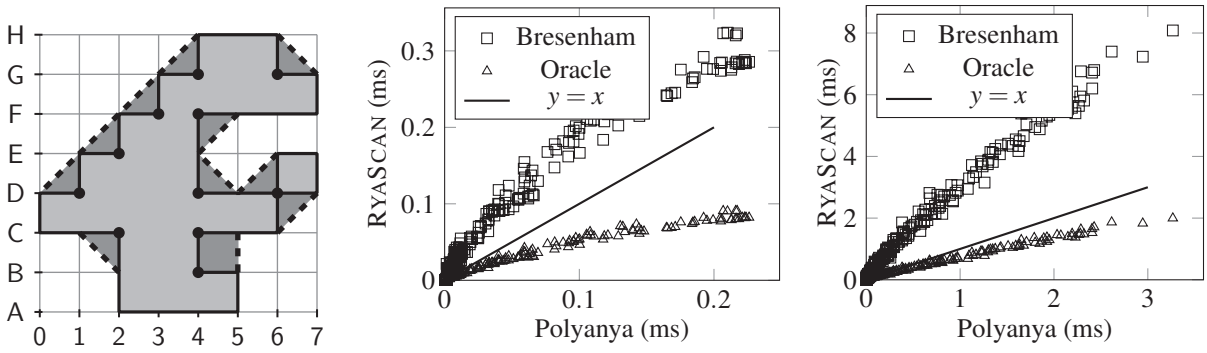
Table 1 gives more precise picture on the idea of taking the setup costs into account. Column P_s is calculated as the time it take to perform Fade2D Constrained Delaunay Triangulation, while column R_s includes the creation of RAYSCAN nodes for search and the filled grid for Bresenham ray shooting to be performed.

We can see that Polyanya takes more time to initialise compare to RAYSCAN, especially for large maps like the random maps. If we assume runtimes always take the 50th percentile for each, both methods will need to perform c_{50} number of searches before the total search and setup time of RAYSCAN exceeds Polyanya. The same for c_{90} using 90th percentile. Looking at how c_{50} differs from c_{90}, we see that we can get many more short searches done compared to long searches before RAYSCAN is overall slower, showing the cost for RAYSCAN is number of expansions.

The c_i variables shows that even though many searches can be done in the time it takes to set Polyanya up, Polyanya would still be better in most cases for static maps, with some minor exceptions on a particular map (random/512-40-0) with a few search instances, where RAYSCAN ran faster. This is due to Polyanya repeating work.

What is really being examined is dynamacally environments, where changes to the obstacles may occur between each search or on set intervals. This would require Polyanya to perform some mesh repairs, whereas RAYSCAN will mainly need to update its ray shooter to the changes (e.g. our Bresenham's shooter will undraw removed polygons and draw added polygons). Tests on dynamically changing maps remain as future work, although with $R_s \ll P_s$, the update time is favorable to our method.

[2] https://www.geom.at/products/fade2d/



(a) Trimming Example (b) DAO-arena2 (c) SC1-Aurora

Figure 5

			RayScan				Polyanya					
map	#	d	R_s	R_t	R_{50}	R_{90}	P_s	P_t	P_{50}	P_{90}	c_{50}	c_{90}
dao/arena2	914	2.59	$1.4k$	$94k$	73	264	$4.5k$	$53k$	32	181	78	37.9
wc3/battleground	1015	5.45	$4.1k$	$88k$	71	179	$14k$	$18k$	11	40	163	71.3
sc1/Aftershock	1794	2.77	$6.6k$	$544k$	192	763	$28k$	$173k$	48	280	146	43.5
sc1/Aurora	2957	2.35	$20k$	$5.7M$	$1.2k$	$5k$	$171k$	$1.9M$	313	$1.8k$	175	46.8
random/512-10-0	1631	16.4	$25k$	$15M$	$5.9k$	$23k$	$6.9M$	$2.4M$	832	$3.8k$	1357	362
random/512-40-0	2389	2.17	$23k$	$29M$	$9.7k$	$29k$	$1.7M$	$23M$	$5.7k$	$23k$	427	272
rooms/30_000	1896	2.66	$6k$	$510k$	185	643	$23k$	$231k$	76	306	160	51.9
mazes/001	278	1.41	$9.3k$	$559k$	$1.5k$	$4.8k$	$196k$	$172k$	456	$1.5k$	176	55.8
street/Paris_1_256	1053	2.65	$2.2k$	$385k$	158	$1.1k$	$23k$	$187k$	53	567	198	35.9

Table 1: Selected maps benchmark: # field are the number of search queries; d the average successor RayScan generates per node; R_s and P_s is the setup times for the methods; R_t and P_t is the total search time (excludes setup); R_{50} and P_{50} is the 50th percentile of search and R_{90} and P_{90} the 90th; c_{50} and c_{90} is how many instances RayScan's takes to exceed Polyanya at the ith percentile, accounting for setup, i.e. $R_s + c_i R_i \approx P_s + c_i P_i$. All times are in microseconds.

Related Work

A standard approach to the ESP problem is to create a VGs, which maps each corner point on a polygon to the other corner points that it can see. By then inserting the start and target points into this graph (and learning which points are visible to them) we can perform an A* search from start to target. The disadvantage of this approach is that computation of the VGs is expensive, and if the obstacles on the plane are highly dynamic, then the re-computation of the VGs can dominate the pathfinding run time (see e.g. (Wybrow, Marriott, and Stuckey 2006)).

An alternate approach, Polyanya (Cui, Harabor, and Grastien 2017), is to use a navigation mesh, which breaks the free space into convex polygons, and then use specialised methods to search through space of convex polygons. This requires a navigation mesh which is much cheaper than to construct the VGs but still requires significant computation.

The closest work to ours, also using ray casts, is in the thesis of Oprea (2017). This method tackles any-angle pathfinding on an octile grid, and is limited to obstacles with horizontal and vertical edges. Rather than find all the successors of a node, this approach traces around obstacles, optimistically assuming paths are free and then checks the path is clear after it reaches the target. Computational results (Oprea 2017) show it is far from competitive with Polyanya.

The obstacle avoidance algorithm Convexpath-sf (Hong, Murray, and Wolf 2016) solves the ESP problem. Similar to our approach it constructs a subgraph of the VGs that contains the shortest path from s to t. This construction makes use of the idea of only moving around an obstacles convex hull to reach the target, if it determines the points inside the hull are not needed. Then when it constructs these paths, it determines if the edges are unobstructed (by ray cast), or it finds another obstacle to navigate around. It also applies a filter that can determine if adding in that obstacle may be needed for the optimal path, reducing the size of the produced graph. Their results show an average running time in seconds for larger maps, which is orders of magnitude slower than Polyanya.

Conclusion

In this paper we introduce an online approach to navigating a 2D plane with non-convex polygonal obstacles which requires minimal setup. While the time per search query is slower than the state of the art Polyanya, its setup time is very small. Thus it provides a compelling solution for navigation in rapidly changing maps. There is scope to improve its run time using hardware support for ray casting.

Acknowledgements

Muhammad Aamir Cheema is supported by Australian Research Council FT180100140 and DP180103411.

References

Bresenham, J. E. 1965. Algorithm for computer control of a digital plotter. *IBM Systems journal* 4(1):25–30.

Cui, M. L.; Harabor, D.; and Grastien, A. 2017. Compromise-free pathfinding on a navigation mesh. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 496–502. AAAI Press.

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

Harabor, D., and Grastien, A. 2013. An optimal any-angle pathfinding algorithm. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling*, 308–311.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2):100–107.

Hong, I.; Murray, A. T.; and Wolf, L. J. 2016. Spatial filtering for identifying a shortest path around obstacles. *Geographical Analysis* 48(2):176–190.

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

Oprea, P. 2017. *A Novel Online Any-Angle Path Planning Algorithm*. Ph.D. Dissertation, University of Kent,.

Rohnert, H. 1986. Shortest paths in the plane with convex polygonal obstacles. *Information Processing Letters* 23:71–76.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.

van Toll, W. G.; Cook IV, A. F.; and Geraerts, R. 2012. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds* 23(6):535–546.

Wybrow, M.; Marriott, K.; and Stuckey, P. 2006. Incremental connector routing. In *Proceedings of 13th International Symposium on Graph Drawing*, number 3843 in LNCS, 446–457. Springer-Verlag.