Euclidean Pathfinding with Compressed Path Databases

Bojie Shen, **Muhammad Aamir Cheema**, **Daniel D. Harabor** and **Peter J. Stuckey**
Faculty of Information Technology, Monash University, Melbourne, Australia
{bojie.shen, aamir.cheema, daniel.harabor, peter.stuckey}@monash.edu

Abstract

We consider optimal and anytime algorithms for the Euclidean Shortest Path Problem (ESPP) in two dimensions. Our approach leverages ideas from two recent works: Polyanya, a mesh-based ESPP planner which we use to represent and reason about the environment, and Compressed Path Databases, a speedup technique for pathfinding on grids and spatial networks, which we exploit to compute fast candidate paths. In a range of experiments and empirical comparisons we show that: (i) the auxiliary data structures required by the new method are cheap to build and store; (ii) for optimal search, the new algorithm is faster than a range of recent ESPP planners, with speedups ranging from several factors to over one order of magnitude; (iii) for anytime search, where feasible solutions are needed fast, we report even better runtimes.

1 Introduction

We consider the Euclidean Shortest Path Problem (ESPP) which asks us to find obstacle-avoiding paths between pairs of points in the plane. This is a well known problem motivated by a variety of real-world applications including computational geometry, robotics and computer games. In each of these settings, it is desirable to compute paths that are as short as possible and as quickly as possible. Simultaneously achieving both of these properties is challenging and the problem has given rise to a variety of different techniques. Among the most popular and effective are: any-angle grid-based algorithms [Nash and Koenig, 2013; Harabor *et al.*, 2016], mesh-based path planners [Demyen and Buro, 2006; Cui *et al.*, 2017] and modern variations on Visibility Graphs [Oh and Leong, 2017].

Leading works in this area all rely on state-space search to find a solution and that search is often (though not always; see [Demyen and Buro, 2006]) an all-or-nothing affair; i.e. until a best solution is found, nothing is returned. This behaviour may be undesirable as it introduces the potential for so-called *first move lag*, where a mobile agent must wait for the search to finish completely before it can take even a first step toward its target. In this work we propose new algorithmic techniques that can mitigate first move lag using anytime behaviour [Hansen and Zhou, 2007]; i.e. we aim to compute "good" solutions quickly and we guarantee to return optimal solutions eventually, given sufficient time.

Our approach combines the strengths of two recent pathfinding techniques: Polyanya [Cui *et al.*, 2017], an online mesh-based ESPP algorithm, and Compressed Path Databases (CPDs) [Botea, 2011; Strasser *et al.*, 2014], a family of preprocessing-intensive speedup techniques developed for grids and spatial networks. Like many ESPP algorithms, ours is a two step approach involving offline preprocessing followed by online search. In broad strokes:

- During the offline phase, we preprocess the input mesh to extract a graph of co-visible points. We then preprocess the graph to create a CPD: an auxiliary data structure that stores compressed all-pairs data and which can be used to efficiently extract optimal paths between any pair of graph vertices u and v.
- During the online phase, Polyanya connects the start and target points to the co-visible graph. The CPD then proceeds to identify candidate paths: from each of the m outgoing successors of the start point to each of the n incoming successors of the target node.

Because each candidate path is a feasible solution, our approach can provide strong anytime performance and it guarantees to return the optimal path after considering at most $m \times n$ possible paths.

We give a complete description of the new algorithm and a number of additional enhancements that can speed up optimal search. We then demonstrate effectiveness in a range of experiments: on maps from real games and in comparison to a range of leading ESPP techniques appearing in the recent literature. For computing optimal paths, we show that the new method can be substantially faster: from a few factors to over one order of magnitude. For computing fast anytime solutions, and for solutions with bounded suboptimal costs, we show that the gains are larger still.

2 Preliminaries

In the Euclidean Shortest Path Problem (ESPP), we are asked to find point-to-point paths in a continuous 2D workspace which contains polygonal obstacles in fixed positions. Any non-obstacle point from the workspace is a potential start or

target position and the objective is to find an obstacle avoiding, distance minimum path, between pairs of points that are a priori unknown. We next define some necessary terminology.

A **polygon** is a closed set of edges and a set of points each called a vertex. Each edge is a contiguous interval between two different vertices (i.e. $e = [v_1, v_2]$), where v_1 and v_2 are the closed ends of e. Polygons can overlap but only if they share a common edge or vertex.

Two points are **visible** if there exists a straight line between this pair that does not intersect with any point from the interior of a polygon. We suppose that a mobile point-sized agent can directly travel between any pair of co-visible points.

A **path** is a sequence of points $\mathcal{P} = \langle p_1, p_2, \cdots, p_k \rangle$ such that $\forall\ p_i,\ p_{i+1} \in \mathcal{P}$, p_i and p_{i+1} are co-visible. The **cost** of a path $\mathcal{P}$ is the cumulative Euclidean distance between every successive pair of points; i.e. $\text{cost}(\mathcal{P}) = \sum_{i=1}^{k-1} d(p_i, p_{i+1})$ where $d(p, p')$ is the Euclidean (straight line) distance between p and p'. A path is **optimal** if its cost is minimum among all paths between its start and end points.

A vertex is called a **convex vertex** if it is located at the convex corner of an obstacle. For a path $\mathcal{P}$ to be optimal, $\forall\ p_i \in \mathcal{P}$ except start and target, p_i is a convex vertex. A vertex is a **dead-end vertex** if it never occurs on an optimal path, unless it is the start or end of the path.

2.1 Navigation Meshes

A navigation mesh divides the non-obstacle regions into a set of convex polygons. In Fig. 1, black polygons are obstacles whereas green/white polygons correspond to a navigation mesh. Popular with game developers [Rabin, 2008], navigation meshes have several attractive properties: they are easy to compute [Kallmann and Kapadia, 2014], are cheap to store and update, and guarantee representational completeness (i.e. every traversable point appears in the mesh). Navigation meshes have been used for pathfinding in various settings: optimal search [Cui *et al.*, 2017], suboptimal search [Kallmann, 2005] and anytime search [Demyen and Buro, 2006].

2.2 Polyanya

We briefly review Polyanya [Cui *et al.*, 2017], a state-of-the-art optimal mesh-based planner which appears as an important ingredient for the rest of the paper. Polyanya search instantiates A* search [Hart *et al.*, 1968] but on a navigation mesh. The algorithm can therefore be described in the same general way: there exist search nodes which generate successors and these are expanded in best first order according to some admissible heuristic function. Polyanya differs from A* only in the domain-specific model used for each of these components. We sketch the details below (see Fig. 1).

Search nodes: A search node is a tuple of the form (I, r) where r is a distinguished vertex called the *root* and I is a contiguous interval of points from an edge of the mesh with every point $i \in I$ being visible from r. The model can be understood as follows: the root r corresponds to the last turning point on the path and I represents all the possible taut continuations of the path, on the way to the target. The start point s is a special case and defined as $(I = [s], r = s)$.

Successors: The successors of node (I, r) are generated by "pushing" the interval I away from r and across the face

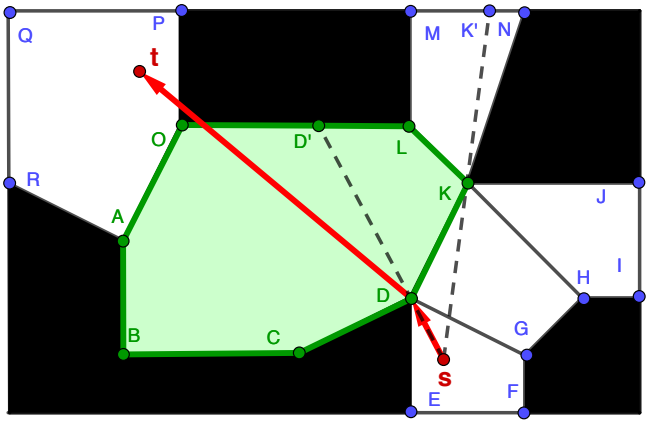

Figure 1: Node expansion in Polyanya. When the current node ([D,K],s) is expanded, it generates the observable successors ([D',L],s), and ([L,K],s); and non-observable successors ([D',O],D), ([O,A],D), ([A,B],D) , ([B,C],D), and ([C,D],D).

of an adjacent traversable polygon. There are two types of successors: *observable* and *non-observable*. A successor $(I', r' = r)$ is observable if each $p' \in I'$ is visible from r. By contrast, a successor $(I', r' \neq r)$ is said to be *non-observable* if each point $p' \in I'$ is not visible from r. Note that observable successors share the same root as the parent. For non-observable successors, the root r' is one of the two endpoints of the parent interval I. Fig. 1 shows the successors for node ([D,K], s). The target is a special case and can be generated as soon as the search reaches its containing polygon.

Evaluation Function: To prioritise a node $n = (I, r)$ for expansion, Polyanya instantiates the f-value function: $f(n) = g(n) + h(n)$. Here $g(n)$ is the cost of the optimal path from the source node s to the root r. The function h is an admissible lower-bound and indicates the cost from r, via some point $p \in I$, to the target t. The estimate requires only simple geometry. Consider for example the node $n = ([\mathtt{D}, \mathtt{K}], \mathtt{s})$ from Fig. 1. The h-value is minimized by choosing the point D; i.e. $h = d(\mathtt{s}, \mathtt{D}) + d(\mathtt{t}, \mathtt{D})$, where d is the Euclidean straight line distance. See [Cui *et al.*, 2017] for more details.

Polyanya terminates when the target is expanded or when the open list becomes empty.

3 Offline Preprocessing

We now describe the auxiliary data structures required by our new algorithm and the offline preprocessing step that constructs them. There are two main steps: constructing a graph of co-visible convex vertices and building a corresponding CPD. This phase takes as input a navigation mesh which can be constructed as described in [Cui *et al.*, 2017].

3.1 Identifying Co-Visible Vertices

A variety of methods exist for generating a graph of co-visible vertices. All have worst-case upper-bounds of $O(n^2)$ where n is the number of vertices in the planar environment. Faster performance can be achieved in practice by only considering and connecting convex vertices. Variations of this idea appear in the literature and under different names; e.g. Tangent

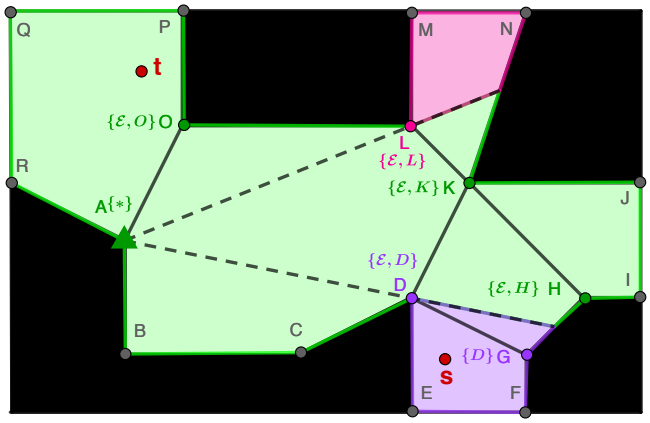

Figure 2: Green area corresponds to the area visible from the source node A. The first move on the optimal path from A to any node in the purple (resp. red) area is D (resp. L).

Graphs [Liu and Arimoto, 1992], Silhouette Points [Young, 2001] and Sparse Visibility Graphs [Oh and Leong, 2017].

We now propose a new practical algorithm for computing such a Visibility Graph, in two dimensions, using the Polyanya path planner. The vertex set V of the visibility graph consists of all convex vertices of the obstacles. In Fig. 2, $\{\mathtt{A}, \mathtt{D}, \mathtt{G}, \mathtt{H}, \mathtt{K}, \mathtt{L}, \mathtt{O}\}$ are convex vertices. Other obstacle vertices (e.g., C) cannot appear on any optimal path, and are dead-end vertices. Next, for each $v \in V$, we run a Polyanya-like depth-first search where each expansion step only generates observable successors. If a successor's interval contains a convex vertex v', we add an edge $(v, v') \in E$, where initially $E = \emptyset$. The cost of this edge is $d(v, v')$. This algorithm has quadratic worst-case but in practice runs much faster.

A similar idea appears in [Oh and Leong, 2017] but their searches are conducted using Anya [Harabor *et al.*, 2016]: an optimal any-angle path planner where obstacles are rasterised using a grid. In experiments, we compare against this method and improve it using our more general mesh-based approach.

3.2 Building the CPD

Given the graph of co-visible nodes, we construct a corresponding CPD [Botea, 2011]: an all-pairs data structure that encodes the first move (equiv. first arc) on the optimal path from each node $s \in V$ to every other node $t \in V$. The procedure is offline and requires one complete Dijkstra search for each source node $s \in V$. The worst-case complexity is therefore $O(|V||E|+|V|^2 \log |V|)$. However, each Dijkstra search can be executed in parallel with a potential speedup depending on the number of processors available on the machine.

First-Move Tables: Using a modified Dijkstra's algorithm, we compute for each source node $s \in V$, a *first move table* where $fm[s, t]$ returns a symbol that tells which of the outgoing arcs of s appear on an optimal path, from s to any $t \in V$. When s and t are co-visible, we also store an additional redundant symbol $\mathcal{E}$ indicating that the two nodes are directly reachable along a straight-line Euclidean-optimal path. Table 1 shows all first moves for source vertices A, D and G in Fig. 2. Another special symbol is "*" (wildcard)

Ordering	A	D	G	H	K	L	O
A	*	{$\mathcal{E}$,D}	D	{$\mathcal{E}$,H}	{$\mathcal{E}$,K}	{$\mathcal{E}$,L}	{$\mathcal{E}$,O}
D	{$\mathcal{E}$,A}	*	{$\mathcal{E}$,G}	{$\mathcal{E}$,H}	{$\mathcal{E}$,K}	{$\mathcal{E}$,L}	{$\mathcal{E}$,O}
G	D	{$\mathcal{E}$,D}	*	{$\mathcal{E}$,H}	{$\mathcal{E}$,K}	{$\mathcal{E}$,L}	{$\mathcal{E}$,O}

Table 1: First moves for A, D and G for the example of Fig. 2.

which we add for table entries where $s = t$. We include the redundant symbols and wildcards because these substantially improve compression as shown in [Chiari *et al.*, 2019].

Compression: We compress first-move tables using run-length encoding (RLE) [Strasser *et al.*, 2014]. RLE compresses a string of symbols into representative sub-strings, called *runs*. Each run has two values: a start index and a first-move symbol. For example, the string $\mathcal{E}; \mathcal{E}; \mathcal{E}; \mathcal{E}; \mathtt{D}; \mathtt{D}$, can be compactly represented as two runs: $1\mathcal{E}; 5\mathtt{D}$.

To improve RLE compression we apply several known enhancements. First, we allow the wildcard symbol "*" to be compressed with any other preceding or subsequent symbol. Secondly, for table entries with multiple symbols, we choose the one that produces a longer run. For example, row A in Table 1 can compress into just two runs: $1\mathtt{D}; 4\mathcal{E}$ (cf. 3 runs if we choose $\mathcal{E}$ as the symbol for column D).

The effectiveness of RLE compression is dependent on the way the candidate nodes are ordered. Following the suggestion in [Strasser *et al.*, 2015], we apply the Depth-First-Search (DFS) Ordering. In Table 1, the order of the columns is a DFS order of convex vertices in Fig. 2.

4 Online Search

CPDs can efficiently retrieve optimal paths when both source s and target t are the vertices of the co-visible graph. We use the function $fm[s, t]$ which extracts from the database a first-move symbol, from s to t. Each extraction operation requires a binary search through an RLE-encoded string of symbols [Strasser *et al.*, 2014]. Once a first-move is extracted, it can be executed (i.e. followed) to reach a new location. The entire pathfinding process can thus be implemented using simple recursion: we extract and follow optimal moves until the target is reached.

One of the main challenges in ESPP is that s and t can be arbitrary (i.e. a priori unknown) locations on the map. To handle such cases we propose to first identify all graph vertices visible from s, denoted V_s, and all graph vertices visible from t, denoted V_t. We then extract a set of paths, from each $v_s \in V_s$ to each $v_t \in V_t$. Let $cpd(v_i, v_j)$ denote the cost of the optimal path between v_i and v_j. The minimal path (i.e., the one with shortest distance) sd from s to t is then

$$sd = \min\{d(s, v_s)+cpd(v_s, v_t)+d(v_t, t) \mid v_s \in V_s, v_t \in V_t\}$$

In Fig. 2, $V_s = \{\mathtt{D}, \mathtt{G}, \mathtt{H}, \mathtt{K}, \mathtt{L}\}$ and $V_t = \{\mathtt{A}, \mathtt{O}\}$ and the optimal path from s to t can be obtained by computing the pair-wise optimal paths for each $v_s \in V_s$, $v_t \in V_t$. This basic algorithm extracts at most $|V_s| \times |V_t|$ candidate paths using the CPD and guarantees to return an optimal solution.

4.1 Incremental Exploration

We now consider a more sophisticated algorithm, End Point Search (EPS), that improves performance by reducing the

Algorithm 1: End Point Search (EPS)

Input: s:start, t:target, CPD: compressed-path-database
Output: an optimal path from s to t
Initialization: $V_s = \emptyset$, $V_t = \emptyset$, $p = \emptyset$, $sd = \infty$

1 $end = s$; $opp = t$;
2 **while** $search_s$ *and* $search_t$ *are not exhausted* **do**
3 $v = search_{end}$←getNextVisibleVertex();
4 **if** $v = s$ *or* $v = t$ **then**
5 **return** $\langle s, t \rangle$;
6 **if** $v \neq \perp$ **then**
7 **for** *each* $v' \in V_{opp}$ **do**
8 p, sd = CPD←getSmallerPath(v,v');
9 $search_s, search_t$ ←setSearchBound(sd);
10 V_{end}←append(v);
11 $end, opp = opp, end$;
12 **return** p;

number of pair-wise optimal paths that must be examined before guaranteeing optimality. The key idea of Algorithm 1 is to incrementally explore the visible area from each of s and t, discovering visible vertices for s and t one by one.

We propose to execute two best-first Polyanya searches, denoted $search_s$ and $search_t$, each of which is resumable and each of which generates only observable successors at every expansion step and returns visible vertices as they are found. The algorithm iteratively expands nodes from $search_s$ and $search_t$ in turn until both searches are exhausted (line 2). The end and opp variables (line 1) define from which end of the path we currently generate visible vertices and which is the other/opposite end. During each iteration, the algorithm progresses the relevant search by calling getNextVisibleVertex (line 3). If the returned vertex is s or t, the search terminates because s and t are visible from each other and the optimal path is $\langle s, t \rangle$ (line 5). If the search is not exhausted (i.e., it does not return $\perp$), the algorithm updates the shortest path p and the shortest distance sd by considering all paths from vertices at the opposite end V_{opp} to this new vertex v. Specifically, for each $v' \in V_{opp}$, the algorithm calls getSmallerPath which uses the CPD to get the optimal path from v' to v and updates p and sd if needed (lines 7 and 8). The search bound for both searches $search_s$ and $search_t$ is updated to be the shortest distance sd found so far (line 9). The new vertex v is added to the corresponding visible set V_{end}. The two ends end and opp are then swapped so that the search is alternated between $search_s$ and $search_t$ (line 11). When the while loop concludes, the algorithm returns the best found path. Note that EPS is a bi-directional path extraction algorithm. Different from the bi-directional search algorithms [Holte *et al.*, 2016], the main challenge is to avoid $|V_s| \times |V_t|$ total path extractions rather than balancing the searching effort between the two sides.

4.2 Pruning Candidate Paths

The function getNextVisibleVertex returns a vertex v visible from its root r (one of s or t). We can immediately discount dead-end vertices, and *non-turn* vertices v where the angle from r to v does not allow turning around v (it intersects a polygon obstacle). In Fig. 3, vertex G is visible from s but there is no turning point possible since the incident edge sG continues into the obstacle polygon. We can also prune a vertex v which cannot lead to a shorter path than the current bound, e.g. where $d(s, v) + d(v, t) \geq sd$. For example in Fig. 3, we can safely ignore the vertex K as $d(\mathtt{s}, \mathtt{K}) + d(\mathtt{K}, \mathtt{t}) > sd$, where sd is the length of the optimal path found so far (highlighted as red). Finally getNextVisibleVertex can terminate when the top of the open list has an f value greater than sd, since no path using this entry can be shorter than sd.

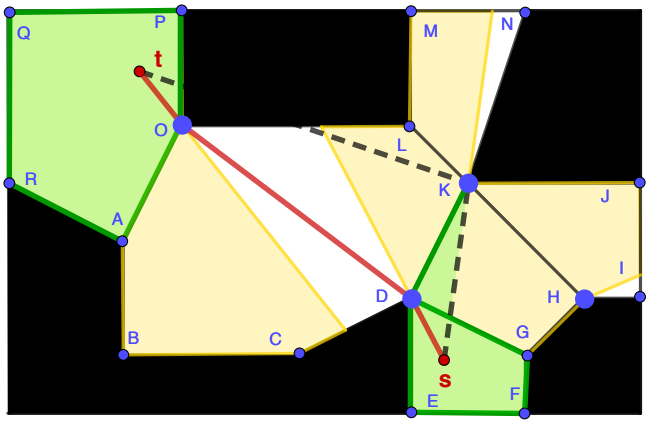

Figure 3: An example of End Point Search. The red lines show the optimal path. The area shown green or yellow corresponds to the space visible from s and t. The green area shows the space incrementally explored by Polyanya when $search_s$ and $search_t$ are both exhausted.

We can avoid extracting paths for pairs ($v_s \in V_s$,$v_t \in V_t$) if they cannot lead to a shorter path than the current bound, i.e. $d(s, v_s) + d(v_s, v_t) + d(v_t, t) > sd$ since $d(v_s, v_t)$ is a lower bound on the shortest path distance $cpd(v_s, v_t)$. Similarly we can prune vertex pairs (v_s, v_t) where the first move is not taut, e.g. if $w = fm[v_s, v_t]$ and $\langle s, v_s, w \rangle$ is not taut then it cannot be part of a shortest path. For example, the first move $fm[\mathtt{H}, \mathtt{0}]$ is 0 but $\langle \mathtt{s}, \mathtt{H}, \mathtt{0} \rangle$ is non taut so we do not need to consider the pair (H,0) further.

4.3 CPD Cost Caching

In each iteration of the while loop, the algorithm uses CPD to extract the paths between a vertex v and every $v' \in V_{opp}$. We use the CPD to extract the optimal path *from* v' *to* v and, for each vertex v_x on the extracted path, we cache $spd(v_x, v)$, the shortest path distance from v_x to v. For a subsequent CPD path extraction, if the optimal path *from* v'' *to* v reaches the vertex v_x for which $spd(v_x, v)$ is cached, we can use the cached distance to get the path length from v'' to v. This simple caching strategy avoids unnecessarily using the CPD to extract the path that is already cached. Although the algorithm can cache $spd(v_x, v)$ for every $v \in V_s \cup V_t$, in our implementation, we only cache $spd(v_x, v)$ for the vertex v in the current iteration of the while loop and reuse the space in each iteration for the new v. This ensures that the caching uses $O(1)$ space for each vertex, i.e., the total space used by the caching is $O(|V|)$ where $|V|$ is the number of nodes in the co-visible graph.

4.4 Putting it All Together

End Point Search gives us an incremental exploration of the pairs of endpoints on the CPD, which is reduced by pruning and improved by caching CPD distances, eventually leading to the optimal path. Overall the approach is correct.

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

Proof. (Sketch) Clearly Algorithm 1 explores all paths examined by the equation defining sd at the beginning of Section 4 except those vertices that are non-turn or have f-values bigger than current distance sd (thus can never be part of the optimal path), and vertex pairs (v_s, v_t) where the shortest possible path is longer than the current distance sd. Hence the returned path is optimal. □

Example 1. Fig. 3 gives an example of the algorithm in action. The search space of our End Point Search (EPS) reduces the observable successors generated as the f-value of the rest of the successors are greater than the sd (i.e. the path shown as the red line). The non-turn vertices: {G} and {A}, and dead-end vertices: {E,F}, and {P,Q,R} are filtered out at the beginning, and the EPS only extracts one path (i.e the optimal path highlighted as red) from CPD. The vertex K can be safely ignored by our distance pruning approach introduced above. Vertices L and H are never found because the $search_s$ exhausts before exploring them. Specifically, both search nodes ([K,L],s) and ([H,K],s) are not expanded by Polyanya as their f-values are bigger than the search bound sd.

5 Experiments

We run experiments on a variety of grid map benchmarks which are described in [Sturtevant, 2012], including 373 game maps from four sets of maps: DAO (156), DA (67), BG (75), SC (75). All benchmarks are available from the HOG2 online repository.[1] We compare our algorithm with a range of competitors detailed below:

Polyanya [Cui *et al.*, 2017] is a fast, optimal, online pathfinding algorithm on navigation mesh. The source code of Polyanya and input navigation mesh are from the publicly available repository.[2]

ENLSVG (Edge-N-Level Spare Visibility Graph) [Oh and Leong, 2017] is an optimal, off-line pathfinding algorithm. The implementation is taken from an online repository.[3]

Poly-ENLSVG is an improvement of the original ENLSVG algorithm which we improve by applying our Polyanya-based visible vertex retrieval approach (see Section 3.1) for the insertion phase of ENLSVG. Here, we prune the dead-end and non-turn vertices to further improve the performance.

SUB-NL (N-level Subgoal graph) [Uras and Koenig, 2015] is a suboptimal, off-line pathfinding algorithm. We run Theta-A* [Nash *et al.*, 2007] on top of N-level subgoal graph, using the publicly available implementation.[4]

We implemented our algorithm in C++. All the experiments are performed on a 2.6 GHz Intel Core i7 machine with 16GB of RAM and running OSX 10.14.6.

[1] https://github.com/nathansttt/hog2

[2] https://bitbucket.org/mlcui1

[3] https://github.com/Ohohcakester

[4] http://idm-lab.org

	#M	#Q	#V	#CV	Build Time		Raw Memory		CPD Memory	
					Avg	Max	Avg	Max	Avg	Max
DAO	156	159k	1727.6	926.5	0.033	0.831	8.012	134.977	0.207	3.640
DA	67	68k	1182.9	610.8	0.006	0.048	2.244	20.611	0.063	0.254
BG	75	93k	1294.4	667.7	0.011	0.233	3.887	66.064	0.119	1.366
SC	75	198k	11487.5	5792.7	0.711	8.463	190.38	2202.23	2.325	14.075

Table 2: Total number of Maps (#M) and Queries (#Q), average number of vertices (#V) and convex vertices (#CV) in the maps, average and maximum building time in minutes, and average and maximum memory before compression (Raw memory) and after compression (CPD memory) in MB for the four benchmark suites.

	Total		Poly-ENLSVG		EPS			
	$\|V_s\|$	$\|V_t\|$	$\|V_s\|$	$\|V_t\|$	$\|V_s\|$	$\|V_t\|$	#Paths	#FirstMoves
DAO	69.324	71.495	19.778	19.987	15.093	14.923	54.182	773.041
DA	46.171	45.707	13.202	12.922	10.656	10.650	25.114	324.294
BG	51.926	49.175	15.629	14.335	9.185	9.015	15.264	140.324
SC	180.013	178.874	45.889	45.356	29.517	29.449	110.214	1767.046

Table 3: $|V_s|$ (resp. $|V_t|$) denotes the average number of vertices visible from s (resp. t) considered by an algorithm to obtain the results. Total includes all visible vertices for s or t without any pruning. For EPS, we also show the average number of path extractions and first move extractions from the CPD.

Experiment 1: CPD Statistics Table 2 shows the average and maximal size of CPD, and building time for the four benchmarks suites. Clearly, our CPDs are memory efficient and the compression reduces the size of first-move tables by up to two orders of magnitude. The tables have very small numbers of runs per entry and hence very fast lookup times. Our CPDs are cheap to build, and for most of the maps can be computed within a few minutes. Note that the CPDs are built on a 12 core Macbook Pro laptop and the performance would be better/worse if more/less processors are available.

Experiment 2: Query Processing Time In Fig. 4 , we compare the query processing time for our approach against the competitors. We sort the queries by the number of node expansions required by the standard A* search to solve them (which is a proxy for how challenging a query is) and the x-axis corresponds to the percentile ranks of queries in this order. Fig. 4 shows that EPS significantly outperforms the competitors on all four benchmarks especially when the queries are more challenging. Note that the y-axis is shown logarithmic. EPS is around 2-3 times faster than SUB-NL (which does not guarantee optimal solutions) and 2-4 times faster than Poly-ENSLVG. Polyanya is faster than EPS for the less challenging queries because, for such queries, s and t are close (and often visible from each other) and the dominant cost for EPS is the two incremental Polyanya searches from s and t. For more challenging queries, EPS is more than an order of magnitude faster than Polyanya.

Table 3 reports the average number of the vertices visible from s and t expanded by Poly-ENLSVG and EPS after pruning non-turn and dead-end vertices. Both algorithms significantly reduce the number of visible vertices expanded. Since EPS makes use of the search bound sd to restrict the Polyanya search, it expands a smaller number of visible ver-

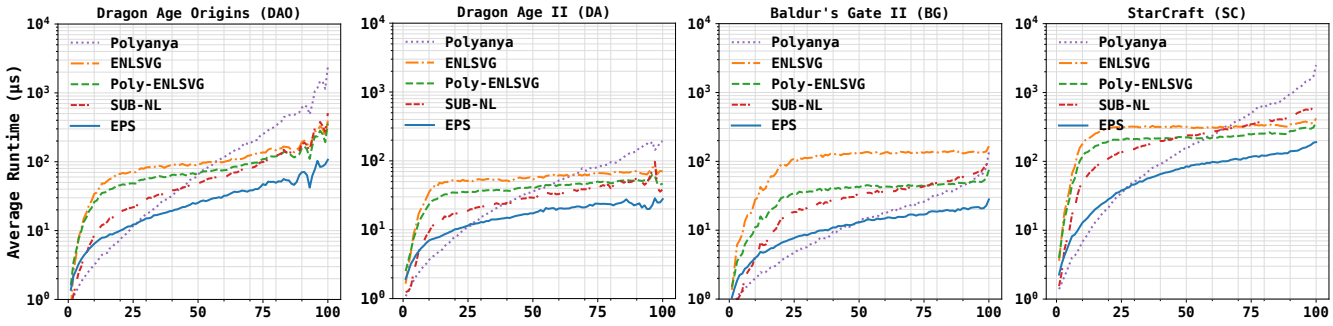

Figure 4: Runtime comparison on the four benchmarks (lower the better). The x-axis shows the percentile ranks of queries in number of node expansions needed by A* search to solve them.

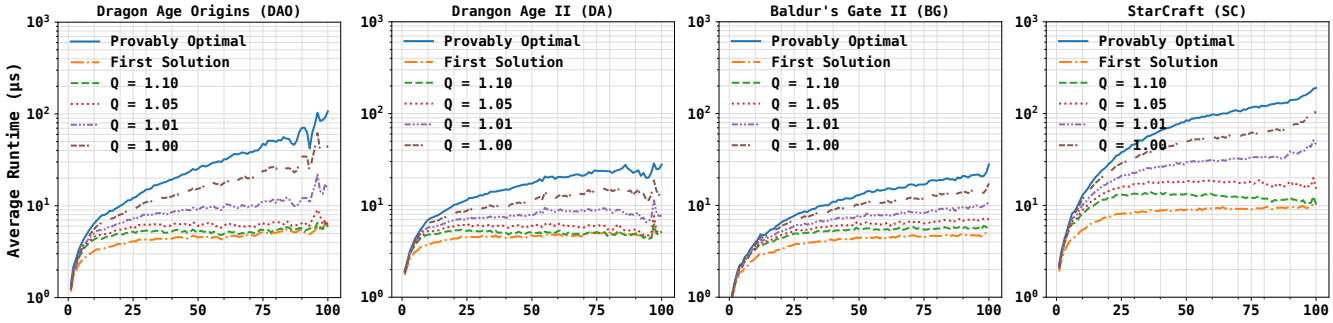

Figure 5: EPS anytime behaviour. The x-axis is the same as in Fig. 4. The y-axis shows the average runtime when EPS finds the first path with length within a certain factor Q of optimal path length (i.e., 1.00, 1.01, 1.05 and 1.10). Q = 1.0 is the time when EPS happens to discover the optimal path but cannot guarantee its optimality. The provably optimal path is the guaranteed optimal path at termination.

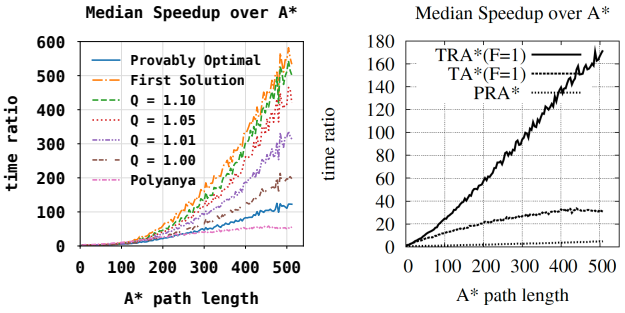

Figure 6: (left) Speedup of EPS (over A* search) for finding solutions of different quality on benchmark suite BG, and (right) a reproduced graph for the same experiment for TRA*.

tices than Poly-ENLSVG especially for BG and SC benchmarks. Also, note that the number of path extractions by EPS is much smaller than $|V_s| \times |V_t|$ since path pruning can avoid considering many of them.

Experiment 3: Anytime Search In time-constrained applications (e.g., computer games), anytime pathfinding is often desirable which returns a valid but potentially suboptimal path as soon as possible before progressively optimizing it until an optimal path is found. This motivates us to consider EPS as an anytime search algorithm. Fig. 5 shows the runtimes of EPS to find the first valid path, a path with length within a certain factor of the optimal path length, or the guaranteed optimal path (i.e., when EPS terminates). EPS demonstrates excellent anytime behaviors, e.g., it finds the first valid path within 10μs and paths within 10% of optimal within 15μs.

In Fig. 6 (left) we show the speedup of EPS anytime search compared to A* search. Fig. 6 (right) shows a graph reproduced from [Demyen and Buro, 2006] showing similar comparison for TRA* anytime search, a popular mesh-based planner, which aims at finding the first solutions fast. EPS finds first solutions 3 times faster and finds the optimal solution (i.e., $Q = 1.00$) in similar time that TRA* requires to find the first solution.

6 Conclusion

We introduce a new approach to Euclidean path finding which substantially improves the state of the art for optimal ESPP and has equally impressive anytime behaviour. It makes use of powerful CPD approaches to handle path finding on the visibility graph, and an efficient incremental attachment of the end points to this graph, to quickly find high quality solutions, and prove optimality fast.

Acknowledgements

Muhammad Aamir Cheema is supported by FT180100140 and DP180103411. Daniel D. Harabor is supported by DP190100013.

References

[Botea, 2011] Adi Botea. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2011, October 10-14, 2011, Stanford, California, USA*. The AAAI Press, 2011.

[Chiari *et al.*, 2019] Mattia Chiari, Shizhe Zhao, Adi Botea, Alfonso Emilio Gerevini, Daniel Harabor, Alessandro Saetti, Matteo Salvetti, and Peter J. Stuckey. Cutting the size of compressed path databases with wildcards and redundant symbols. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, pages 106–113. AAAI Press, 2019.

[Cui *et al.*, 2017] Michael Cui, Daniel Damir Harabor, and Alban Grastien. Compromise-free pathfinding on a navigation mesh. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 496–502. ijcai.org, 2017.

[Demyen and Buro, 2006] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 942–947. AAAI Press, 2006.

[Hansen and Zhou, 2007] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[Harabor *et al.*, 2016] Daniel Damir Harabor, Alban Grastien, Dindar Öz, and Vural Aksakalli. Optimal any-angle pathfinding in practice. *Journal of Artificial Intelligence Research*, 56:89–118, 2016.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Holte *et al.*, 2016] Robert C. Holte, Ariel Felner, Guni Sharon, and Nathan R. Sturtevant. Bidirectional search that is guaranteed to meet in the middle. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 3411–3417. AAAI Press, 2016.

[Kallmann and Kapadia, 2014] Marcelo Kallmann and Mubbasir Kapadia. Navigation meshes and realtime dynamic planning for virtual worlds. In *ACM SIGGRAPH 2014 Courses*, page 3. ACM Press, 2014.

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

[Liu and Arimoto, 1992] Yun-Hui Liu and Suguru Arimoto. Path Planning Using a Tangent Graph for Mobile Robots Among Polygonal and Curved Obstacles. *International Journal of Robotics Research*, 11:376–382, August 1992.

[Nash and Koenig, 2013] Alex Nash and Sven Koenig. Any-angle path planning. *AI Magazine*, 34(4):9, 2013.

[Nash *et al.*, 2007] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. 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*, pages 1177–1183. AAAI Press, 2007.

[Oh and Leong, 2017] Shunhao Oh and Hon Wai Leong. Edge n-level sparse visibility graphs: Fast optimal any-angle pathfinding using hierarchical taut paths. In *Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS 2017, 16-17 June 2017, Pittsburgh, Pennsylvania, USA*, pages 64–72. AAAI Press, 2017.

[Rabin, 2008] Steve Rabin, editor. *AI Game Programming Wisdom 4*. Charles River Media, 2008.

[Strasser *et al.*, 2014] Ben Strasser, Daniel Harabor, and Adi Botea. Fast first-move queries through run-length encoding. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. AAAI Press, 2014.

[Strasser *et al.*, 2015] Ben Strasser, Adi Botea, and Daniel Harabor. Compressing optimal paths with run length encoding. *Journal of Artificial Intelligence Research*, 54:593–629, 2015.

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

[Uras and Koenig, 2015] 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 2015, Jerusalem, Israel, June 7-11, 2015*, pages 234–238. AAAI Press, 2015.

[Young, 2001] Thomas Young. Optimizing Points-of-Visibility Pathfinding. In *Game Programming Gems 2*, pages 324–329. Charles River Media, 2001.