Fast Optimal and Bounded Suboptimal Euclidean Pathfinding

Bojie Shen, Muhammad Aamir Cheema, Daniel D. Harabor, and Peter J. Stuckey

Faculty of Information Technology, Monash University, Australia

Abstract

We consider optimal and suboptimal algorithms for the Euclidean Shortest Path Problem (ESPP) in two dimensions. For optimal path planning, 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 (CPD), a speedup technique for pathfinding on grids and spatial networks, which we exploit to efficiently compute candidate paths, in order to construct a completely novel ESPP algorithm, End Point Search (EPS). In a range of experiments and empirical comparisons we show that: (i) the auxiliary data structures required by EPS 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 performance. For suboptimal path planning, we extend the CPD such that it computes and compresses first move data of a larger number of selected candidate nodes covering every point in the Euclidean space. Our approach is search-free, simultaneously fast, and returns a path within a fixed bound of the optimal solution. In a range of empirical results, we show that: (i) our approach outperforms both offline/online optimal and suboptimal ESPP algorithms proposed in the literature; (ii) our approach demonstrates excellent path quality, better than all existing suboptimal ESPP algorithms; and (iii) the approach offers flexibility by allowing a trade-off between the CPD construction cost (space and time) and the suboptimality bound.

Keywords: Euclidean shortest path planning, Compressed path databases, Heuristic search
2010 MSC: 00-01, 99-00

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 robotics [1], indoor location-based services [2] and computer games [3]. 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 [4, 5], mesh-based path planners [6, 7] and modern variations on Visibility Graphs [8].

Leading works in this area all rely on state-space search to find a solution and that search is often (though not always; see [6]) 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 [9]; 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 [7], an online mesh-based ESPP algorithm, and Compressed Path Databases (CPDs) [10, 11], a family of preprocessing-intensive speedup techniques developed for grids and spatial networks. We provide two versions, one designed for finding optimal paths, and a second for bounded suboptimal paths. Like many ESPP algorithms, both use a two step approach involving offline preprocessing followed by online search. In broad strokes:

Symbol	Description
s	The start/source of a pathfinding query.
t	The target of a pathfinding query.
p_i	A point in arbitrary Euclidean space.
v_i	A vertex on a polygonal obstacle.
I	A contiguous interval (i.e., segment) of an edge of the mesh
$\mathcal{P}$	A path that consists of a sequence of points $\langle p_1, p_2, \cdots, p_k \rangle$.
$\|\mathcal{P}\|$	The cost (i.e., length) of a path $\mathcal{P}$.
$\oplus$	A concatenation operator that concatenates two paths $\mathcal{P}_i$ and $\mathcal{P}_j$.
$d(x, y)$	The Euclidean distance between two points x and y.
$sp(x, y)$	The shortest path from x to y.
$fm[x, y]$	A function that extracts the *first-move* on $sp(x, y)$ using CPD where x and $y \in$ CPD.
$cpd(x, y)$	The shortest path $sp(x, y)$ extracted using CPD where x and $y \in$ CPD.
c_i	The centroid of a circle.
δ	The radius of a circle.
ϵ	The error bound for bounded suboptimal search.
$lb(x, y)$	A lower bound on the shortest distance between x and y, i.e., $lb(x, y) \leq \|sp(x, y)\|$.
$bsp(s, t)$	A bounded-suboptimal path between s and t, i.e., $\|bsp(s, t)\| \leq \|sp(s, t)\| + \epsilon$.

Table 1: Summary of the notations used in the paper

- During the offline phase, we preprocess the input mesh to extract a graph of "interesting" 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 "interesting points" u and v.
- During the online phase, we connect the start and target points to the "interesting points" graph. We use the CPD to identify candidate paths. In the optimal algorithm, we consider paths from each of the $|V_s|$ outgoing successors of the start point s to each of the $|V_t|$ incoming successors of the target t. In the suboptimal algorithm, we only consider paths from the nearest "interesting point" to the start to the nearest "interesting point" to the target.

For the optimal approach, 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 $|V_s| \times |V_t|$ possible paths where $|V_s|$ and $|V_t|$ correspond to the number of convex vertices visible from start and target, respectively. For the suboptimal approach, the whole pathfinding process is completed by considering only one path instead of $|V_s| \times |V_t|$ paths, so it is very fast.

We give a complete description of the new algorithms 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, both optimal and bounded suboptimal, appearing in the recent literature. For optimal path construction, 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 speed gains are even larger.

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 priori unknown. We next define some necessary terminology. Table 1 summarizes the symbols used in this paper.

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

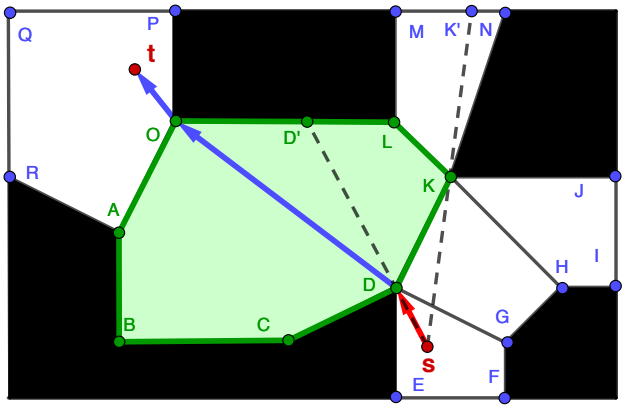


Figure 1: Node expansion in Polyanya. When the current node ([D,K],s) is expanded, it generates the observable successors ([D',L],s), and ([L,K],s); and non-observable successors ([D',O],D), ([O,A],D), ([A,B],D) , ([B,C],D), and ([C,D],D).

can overlap but only if they share a common edge or vertex. A **convex polygon** is a polygon where every line drawn between points in the polygon remains within the polygon.

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., $|\mathcal{P}| = \sum_{i=1}^{k-1} d(p_i, p_{i+1})$ 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. We use the notation $sp(s, t)$ to define the shortest (i.e., optimal) path from s to t. We use the notation $\oplus$ to join paths together, assuming the two endpoints are identical so $\langle p_1, \ldots, p_n \rangle \oplus \langle p'_1, p'_2, \ldots, p'_m \rangle$ generates the path $\langle p_1, \ldots, p_n, p'_2 \ldots, p'_m \rangle$ assuming $p_n = p'_1$.

A vertex of a polygon is called a **convex vertex** if any line between two points "near" the corner stays within the polygon. More formally, if we form the triangle of the vertex with its two neighbours on the polygon, then the triangle is within the polygon. For a path $\mathcal{P}$ to be optimal in a Euclidean path finding problem with polygonal obstacles, $\forall\ p_i \in \mathcal{P}$ except start and target, p_i must be a convex vertex [8, 12]. 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 [13], navigation meshes have several attractive properties: they are easy to compute [14], 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 [7], suboptimal search [15] and anytime search [6].

2.2. Polyanya

We briefly review Polyanya [7], 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 [16] 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

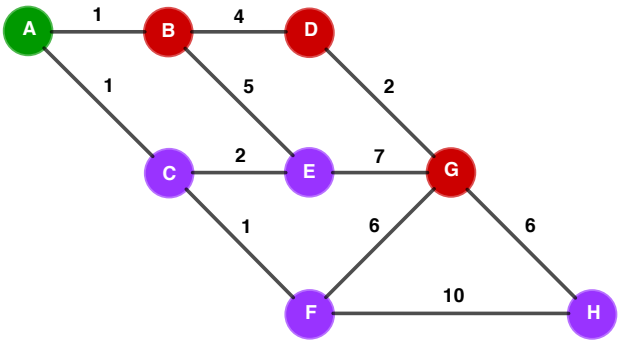


Figure 2: An example weighted-graph, the source node A is highlighted as green. The first move on the optimal path from A to any node that shown as red (resp. purple) is B (resp. C)

Ordering	A	B	D	G	H	E	C	F
A	*	B	B	B	C	C	C	C
B	A	*	D	D	D	A	A	A
D	B	B	*	G	G	B	B	B

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

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 $(I = [s], r = s)$. In the example of Fig. 1, $([\mathtt{D}, \mathtt{K}], s)$ is a search node where the root is s and the contiguous interval visible from s is $[\mathtt{D}, \mathtt{K}]$.

Successors: The successors of node (I, r) are generated by "pushing" the interval I away from r and across the face of an adjacent traversable polygon. There are two types of successors: *observable* and *non-observable*. A successor (I', r) is observable if each point $p' \in I'$ 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 $([\mathtt{D}, \mathtt{K}], s)$ in green. Since the edge $[\mathtt{L}, \mathtt{K}]$ is visible from the root s, it is an observable successor and has the same root, i.e., the successor node is $([\mathtt{L}, \mathtt{K}], s)$. In contrast, the edge $[\mathtt{O}, \mathtt{A}]$ is not entirely visible from the root s and it is a non-observable successor with root D, i.e., the successor node is $([\mathtt{O}, \mathtt{A}], \mathtt{D})$. 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 $n = ([\mathtt{O}, \mathtt{A}], \mathtt{D})$ from Fig. 1. The g-value (shows in red) is the shortest distance from s to the root D. The h-value (shows in blue) is the minimum Euclidean distance from D to t that passes through the edge $[\mathtt{O}, \mathtt{A}]$, i.e., $h = d(\mathtt{D}, \mathtt{O}) + d(\mathtt{O}, t)$, where $d()$ is the Euclidean straight line distance. See [7] for more details.

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

2.3. Compressed Path Database

A **compressed path database** (CPD) [10] is an auxiliary data structure that encodes and compresses the first move (equivalent first arc) on the optimal path from each node $s \in V$ to every other node $t \in V$. Given a weighted graph V (i.e., an example is shown in Fig 2), CPDs are constructed offline using 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: With only a small modification to the basic Dijkstra algorithm (specifically, for each node, we maintain the first outgoing arc on the optimal path from s to this node), 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 $t \in V$. Table 2 shows all first moves for source vertices `A`, `B` and `D` in Fig. 2. Note that each $fm[s,s]$ is assigned a wildcard symbol [17] "*" (i.e., "don't care" symbols) as we never need to look up a move from s to itself.

Compression: The CPD compresses first-move tables using run-length encoding (RLE) [11]. 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. E.g., the string `C`; `C`; `C`; `D`; `D`; `D`, can be compactly represented as two runs: 1`C`; 4`D`. In addition to that, the wildcard symbol "*" is allowed to be compressed with any other preceding or subsequent symbol. For example, row `A` in Table 2 can be compressed into just two runs: 1`B`; 5`C`.

The effectiveness of RLE compression is dependent on the way the candidate nodes are ordered. Following the suggestion in [18], we use a Depth-First-Search (DFS) ordering of nodes. Specifically, we run a DFS on a randomly selected node and the DFS ordering corresponds to the order in which these nodes are accessed by the DFS. This ordering tends to order the nodes that have the same symbol closer to each other which helps with compression. In Table 2, the order of the columns is a DFS visit order in Fig. 2 starting from node `A`.

Path Extraction: CPDs can efficiently retrieve optimal paths for any given start-target pair within the graph. We denote the function $fm[s,t]$ which extracts from the database a first-move symbol, from s to t. Each extraction operation $fm[s,t]$ requires a binary search on the RLE-encoded first-move table of s to find the first-move symbol to reach t, which runs in $O(\log n)$ where n is the number of symbols in it [11]. 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 a simple recursion: we extract and follow optimal moves until the target is reached.

3. Optimal Search

Our first contribution is an algorithm for quickly finding optimal Euclidean paths. We examine the two components of offline preprocessing followed by path extraction.

3.1. 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 [7].

3.1.1. Identifying Co-Visible Vertices

A variety of methods exists 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 many times in the literature and under different names; e.g. Tangent Graphs [19], Silhouette Points [20] and Sparse Visibility Graphs [8].

We now propose a new efficient 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. 3, $\{$`A`, `D`, `G`, `H`, `K`, `L`, `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 modified Polyanya search to find convex visible vertices from v. Specifically, we modify the Polyanya such that it only generates visible successors and the search runs in a depth-first search manner without using any heuristic. If a successor's

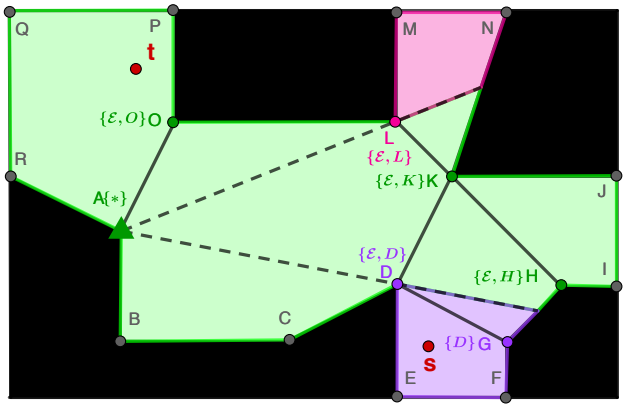


Figure 3: Green area corresponds to the area visible from the source node A. The first move on the optimal path from A to any node in the purple (resp. pink) area is D (resp. L).

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 3: First moves for A, D and G for the example of Fig. 3.

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 the usual quadratic worst-case but in practice runs much faster.

Consider Fig. 3 as an example and assume that the source node is A. The search starts by generating all the visible successors for the two adjacent polygons that contain A. It cannot expand further for the successors that are on the obstacles or map boundary (e.g., ([A, B], A), ([B, C], A), ([P, O], A) and ([P, Q], A) etc.,). Thus, the remaining visible successors are ([D, K], A), ([K, L], A) and ([A, O], A). When we expand the successor ([D, K], A), it finds two visible convex vertices D and K, and generates the visible successor ([H, K], A). Since it is a depth-first search, the successor ([H, K], A) is expanded which finds the visible convex vertex H, and generates the successors ([H, I], A), ([I, J], A) and ([K, J], A) (which are all ignored because they are either on obstacles or on the map boundary. Similarly, the successors ([K, L], A) and ([A, O], A) are processed and two visible convex vertices L and O are found by them, respectively. The search terminates after exploring the area visible from A (green area shown in Fig. 3). Thus, this Polyanya search adds edges from A to each of the convex visible vertices {D, K, H, L, O} into E along with their corresponding Euclidean distances.

We remark that a previous work [8] used a similar approach to find co-visible vertices but their searches are conducted using Anya [5]: an optimal any-angle path planner where polygonal obstacles are rasterised using a grid. Anya searches on a grid map and generates the successors by considering the adjacent grid row. The search space is explored row-by-row. On the other hand, Polyanya extends Anya in Euclidean space which searches on the convex polygons of mesh and explores polygon-by-polygon, hence improves the node expansions by a few factors and achieves up to one order of magnitude speed up [7]. In experiments, we compare against this method and we improve it using our more general mesh-based approach.

3.1.2. Building the CPD

Given the graph of co-visible nodes, we construct a corresponding CPD [10]: an all-pairs data structure that encodes the first move (equivalent first arc) on the optimal path from each node $s \in V$ to every other node $t \in V$.

First-Move Tables: As stated earlier, the first-move table of s stores the first-move symbol $fm[s, t]$ for every $t \in V$. When s and t are co-visible (i.e., $fm[s, t] = t$), in addition to storing the first-move t, we

also store a redundant symbol $\mathcal{E}$ which represents that s and t are co-visible (i.e., a direct path from s to t exists). For example, for the first-move from A to D, we store D as well as $\mathcal{E}$. Another special symbol is (wildcard) "*" symbol which we add for table entries where $s = t$, since these entries will never be retrieved. We include the redundant and wildcards symbols because they substantially improve compression as shown in [17] and explained shortly. Table 3 shows all first moves for source vertices A, D and G in Fig. 3.

Compression: We compress first-move tables using run-length encoding (RLE) [11]. To improve RLE compression we apply several known enhancements [17]. First, we allow the wildcard symbol "*" to be compressed with any other preceding or subsequent symbol. Secondly, for table entries with redundant symbols, we choose the one that produces a longer run. For example, row A in Table 3 can compress into just two runs: 1D; 4$\mathcal{E}$ (cf. 3 runs if we choose $\mathcal{E}$ as the symbol for column D).

We use a Depth-First-Search (DFS) ordering of columns as suggested in [18]. In Table 3, the order of the columns is a DFS traversal order of the convex vertices appearing in Fig. 3 starting from A.

3.2. Online Search

CPDs can efficiently retrieve optimal paths when both source s and target t are the vertices of the co-visible graph as discussed in Section 3.1.2. 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 an optimal path from v_i and v_j (extracted via the CPD). If s and t are not visible to each other, the shortest path (i.e., the one with shortest distance) sp from s to t is then

$$sp(s,t) = \text{argmin}\{|\langle s, v_s\rangle \oplus cpd(v_s, v_t) \oplus \langle v_t, t\rangle| \mid v_s \in V_s, v_t \in V_t\} \tag{1}$$

In Fig. 3, $V_s = \{\texttt{D}, \texttt{G}, \texttt{H}, \texttt{K}, \texttt{L}\}$ and $V_t = \{\texttt{A}, \texttt{O}\}$ and the optimal path from s to t can be obtained by computing the pair-wise optimal paths for each $v_s \in V_s$, $v_t \in V_t$. As evident from Eq. (1), this basic algorithm extracts at most $|V_s| \times |V_t|$ candidate paths using the CPD and guarantees to return an optimal solution.

3.2.1. Incremental Exploration

We now consider a more sophisticated algorithm, End Point Search (EPS), that improves performance by reducing the number of pair-wise optimal paths that must be examined before guaranteeing optimality. Algorithm 1 provides an overview of the algorithm. Additional pruning rules and optimisations are discussed in Sections 3.2.2 and 3.2.3.

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, generates only visible successors at every expansion step and returns visible vertices as they are found. V_s and V_t record the visible vertices returned so far by $search_s$ and $search_t$, respectively. The shortest path sp and its length $|sp|$ are initialized to be empty and infinity, respectively (we use $\leftarrow$ as an assignment operator and $=$ as an equality condition in the pseudocode).

The algorithm iteratively expands nodes from $search_s$ and $search_t$ in turn until both searches are exhausted (line 2). We use *cur* (resp. *opp*) to denote the current (resp. opposite) direction in which the search is expanding (line 1); i.e., if *cur* is start s then *opp* is target t and vice versa. During each iteration, the algorithm incrementally progresses the relevant Polyanya search which returns the next visible vertex v (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., v is not empty), the algorithm updates the shortest path sp by considering all paths from visible vertices at the opposite end V_{opp} to this new vertex v. Specifically, for each $v' \in V_{opp}$, the algorithm uses the CPD to get the optimal path from v' to v and updates sp if the new path p is shorter than sp (lines 7 to 10). The search bound for both searches $search_s$ and $search_t$ is updated to be the shortest distance $|sp|$ found so far (line 11). The new vertex v is added to the corresponding visible set V_{cur}. The two ends *cur* and *opp* are then swapped so that the search

Algorithm 1: End Point Search (EPS)

Input: s:start, t:target, CPD: compressed-path-database
Output: an optimal path from s to t
Initialization: $V_s \leftarrow \emptyset$, $V_t \leftarrow \emptyset$, $sp \leftarrow \langle\rangle$, $|sp| \leftarrow \infty$

```
1  cur ← s; opp ← t;
2  while search_s and search_t are not exhausted do
3  |  Let v be the next visible vertex returned by search_cur;
4  |  if v = s or v = t then // this implies s and t are co-visible
5  |  |  return ⟨s, t⟩;
6  |  if v ≠ φ then // search_cur is not exhausted
7  |  |  for each v' ∈ V_opp do
8  |  |  |  p ← ⟨opp, v'⟩ ⊕ cpd(v', v) ⊕ ⟨v, cur⟩;
9  |  |  |  if |p| ≤ |sp| then
10 |  |  |  |  sp ← p;
11 |  |  |  |  set |sp| as the search bound for both search_s and search_t;
12 |  |  V_cur ← V_cur ∪ v;
13 |  cur, opp ← opp, cur;
14 return sp;
```

is alternated between $search_s$ and $search_t$ (line 13). When the while loop concludes, the algorithm returns the best found path sp.

Note that EPS is a bi-directional path extraction algorithm. In traditional bi-directional *search* algorithms [21], the search is guaranteed to meet in middle and the challenge is to balance the searching effort between the two sides. In contrast, EPS is a bi-directional *path extraction* algorithm that only requires a bi-directional insertion to connect with the CPD nodes. Here, the main challenge is to avoid $|V_s| \times |V_t|$ total path extractions.

3.2.2. Pruning Candidate Paths

Recall that, in each iteration, the algorithm obtains a vertex v visible from cur (line 3). We can immediately discount *dead-end* vertices, and *non-turn* [22] vertices. A vertex is called a dead-end vertex if it cannot lead to anywhere else in the map, e.g., in Fig. 4, `E` is a dead-end vertex for the source s. A vertex v on a polygon P is a non-turn vertex for cur (e.g., s or t) if v is visible from cur and the ray shot from cur to v enters P from v – the non-turn vertex v does not allow turning *around* such obstacle. In Fig. 4, vertex `G` is visible from `s` but there is no turning point possible since the ray `sG` continues into the obstacle polygon. In contrast, `H` is not a non-turn vertex because the ray from s to it does not enter the polygon (and we can turn around this obstacle from `H`).

We can also prune a vertex v which cannot lead to a shorter path than the current bound, e.g. where $d(s, v) + d(v, t) \geq |sp|$. For example in Fig. 4, we can safely ignore the vertex `K` as $d(\texttt{s}, \texttt{K}) + d(\texttt{K}, \texttt{t}) > |sp|$, where $|sp|$ is the length of the optimal path found so far (highlighted as red). Finally, $search_{cur}$ can terminate when the top of the open list has an f value greater than $|sp|$, since no path using this entry can be shorter than $|sp|$.

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., $d(s, v_s) + d(v_s, v_t) + d(v_t, t) > |sp|$ 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 from either end is non-taut, i.e., string pulling results in a shorter path. For example, let w be the first move on the shortest path from v_s to v_t. If $\langle s, v_s, w\rangle$ is non-taut then it cannot be part of a shortest path. Similarly, we can prune the vertex pair if $\langle t, v_t, w'\rangle$ is not taut where w' is the first move on the shortest path from v_t to v_s. Consider the example in Fig. 4, the first move from `H` to `O` is `O` but $\langle \texttt{s}, \texttt{H}, \texttt{O}\rangle$ is non-taut so we do not need to consider the pair (`H`,`O`) further.

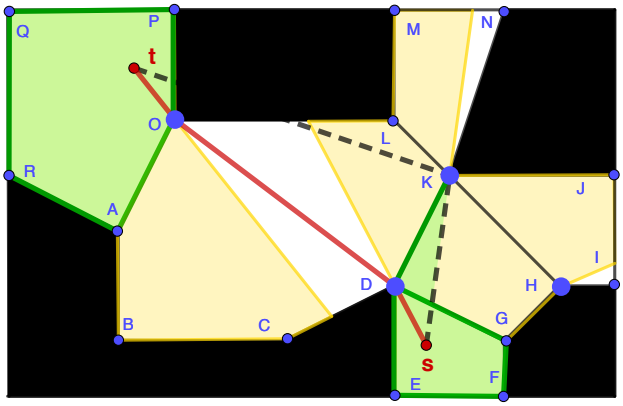


Figure 4: An example of End Point Search. The red lines show the optimal path. The area shown green or yellow corresponds to the space visible from s and t. The green area shows the space incrementally explored by Polyanya when $search_s$ and $search_t$ are both exhausted.

3.2.3. CPD Cost Caching

In each iteration of the while loop, the algorithm uses the CPD to extract the paths between a vertex v and every $v' \in V_{opp}$ (line 8). 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 $|cpd(v_x, v)|$, 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 $|cpd(v_x, v)|$ 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 $|cpd(v_x, v)|$ for every $v \in V_s \cup V_t$, in our implementation, we only cache $|cpd(v_x, v)|$ 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. Moreover, we observed that this caching is already quite effective and caching the distance for each $v \in V_s \cup V_t$ does not result in a significant further improvement in query performance.

3.2.4. 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 an optimal path. Next, we prove the correctness of our approach.

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

Proof. If s and t are visible to each other, one of the two Polyanya searches $search_s$ and $search_t$ will discover this returning $\langle s, t \rangle$ at line 5. Otherwise, the shortest path must contain at least one vertex v_s visible from s (i.e., $v_s \in V_s$) and at least one vertex v_t visible from t (i.e., $v_t \in V_t$). Algorithm 1 explores all paths through $v_s \in V_s$ and $v_t \in V_t$ (see Equation (1) at the beginning of Section 3.2) except: 1) those vertices that are dead-end, non-turn or have f-values bigger than current distance $|sp|$ (thus can never be part of the optimal path); 2) and the vertex pairs (v_s, v_t) where the shortest possible path through them, $\langle s, v_s \rangle \oplus cpd(v_s, v_t) \oplus \langle v_t, t \rangle$, is either non-taut or longer than the current distance $|sp|$. Hence the returned path is optimal. □

Example 1. Fig. 4 gives an example of the algorithm in action. End Point Search (EPS) starts the $search_s$ and returns a visible vertex `D` (line 3). The non-turn vertex ($\{$`G`$\}$) and dead-end vertices ($\{$`E`, `F`$\}$) of the polygon containing s are visited by Polyanya but are pruned as explained in Section 3.2.2. Then, the search is swapped and the $search_t$ returns a visible vertex `O` after filtering $\{$`A`$\}$ (a non-turn vertex) and $\{$`P`, `Q`, `R`$\}$ (dead-end vertices). The CPD is used to extract the path from `D` to `O`, and the shortest path sp (shown

Map	#M	#Q	#V	#CV	Build Time		Raw Memory		CPD\$\mathcal{E}$ Memory		CPD Memory	
					Avg	Max	Avg	Max	Avg	Max	Avg	Max
DAO	156	159k	1727.6	926.5	0.033	0.831	8.012	134.977	0.310	4.605	0.207	3.640
DA	67	68k	1182.9	610.8	0.006	0.048	2.244	20.611	0.115	0.484	0.063	0.254
BG	75	93k	1294.4	667.7	0.011	0.233	3.887	66.064	0.188	1.828	0.119	1.366
SC	75	198k	11487.5	5792.7	0.711	8.463	190.38	2202.23	3.615	19.493	2.325	14.075

Table 4: 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), after compression without using the $\mathcal{E}$ symbol (CPD\$\mathcal{E}$ memory) and after compression (CPD memory) in MB for the four benchmark suites.

red in Fig. 4) and search bound $|sp|$ are updated accordingly (lines 7 - 11). In the next iteration, $search_s$ expands the successor ([D,K],s) and returns the next visible vertex K. However, the vertex K can be safely ignored by our distance pruning approach introduced earlier because the path through K (shown in broken lines) is longer than the current shortest path sp, i.e., $d(s, \mathtt{K}) + d(\mathtt{K}, t) > |sp|$. After that, the $search_s$ and $search_t$ are both exhausted as the f-values of the rest of the successors in their respective queues are bigger than the search bound $|sp|$, e.g., the successor ([K,H],s) is never explored by $search_s$ because its f-value $d(s, \mathtt{K}) + d(\mathtt{K}, t) > |sp|$. Thus, the algorithm terminates and returns sp (the path shown in red).

3.3. Experiments

We run experiments on a variety of grid map benchmarks which are described in [23], 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: [7] is a fast, optimal, online pathfinding algorithm on navigation meshes. The source code of Polyanya and input navigation mesh are from the publicly available repository.[2]

ENLSVG: (Edge-N-Level Spare Visibility Graph) [8] is an optimal, off-line pathfinding algorithm. The implementation of ENLSVG 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 3.1.1) for the insertion phase of ENLSVG. Here, we prune the dead-end and non-turn vertices to further improve the performance.

SUB-N-T: (N-level Subgoal graph) [24] is a suboptimal, off-line pathfinding algorithm. We run Theta-A* [25] on top of N-level subgoal graph, using the publicly available implementation.[4]

TRA*: (Triangulation Reduction A*) [6] is an anytime, optimal pathfinding algorithm that runs on navigation meshes.

For more details of the competitors, see Section 5. We implemented our algorithm in C++. All the experiments are performed on a 2.6 GHz Intel Core i7 machine with 16 GB of RAM and running OSX 10.14.6.

Experiment 1: CPD Statistics. Table 4 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

[1] https://github.com/nathansttt/hog2
[2] https://bitbucket.org/mlcui1
[3] https://github.com/Ohohcakester
[4] http://idm-lab.org/anyangle

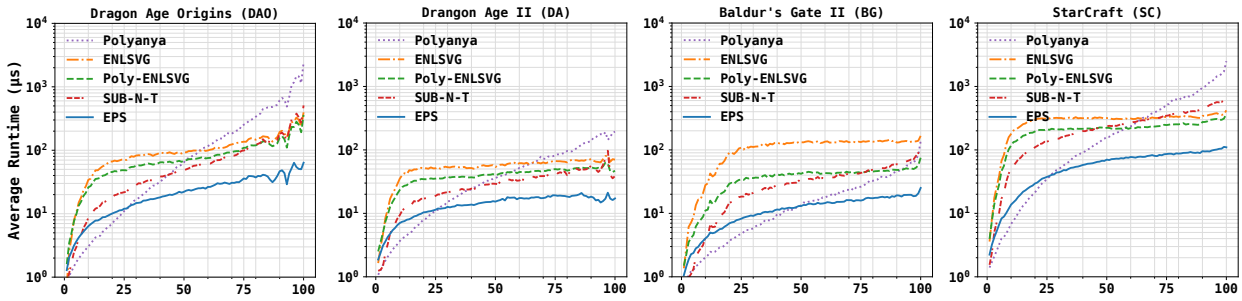


Figure 5: Runtime comparison on the four benchmarks. The x-axis shows the percentile ranks of queries in number of node expansions needed by A* search to solve them.

Map	Total		Poly-ENLSVG		EPS			
	$\|V_s\|$	$\|V_t\|$	$\|V_s\|$	$\|V_t\|$	$\|V_s\|$	$\|V_t\|$	#Paths	#FM
DAO	69.324	71.495	19.778	19.987	15.108	14.955	5.094	329.463
DA	46.171	45.707	13.202	12.922	10.779	10.755	2.470	144.872
BG	51.926	49.175	15.629	14.335	9.445	9.226	2.707	98.955
SC	180.013	178.874	45.889	45.356	29.819	29.707	4.028	976.194

Table 5: $|V_s|$ (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 (#Paths) and first move extractions (#FM) from the CPD.

per entry and hence very fast lookup times. In terms of compression, we also observed that the $\mathcal{E}$ symbol helps to reduce the size of CPDs by 20% – 50% on average and, for some of the maps, we observed that the $\mathcal{E}$ symbol reduces the size by a few factors. In games, it is common to treat all the maps for one game together. Although the raw memory of the first move table of one map may be affordable, it may not be feasible to store the raw tables of all the maps of a game in the main memory (e.g., the sum of raw memory for SC benchmark maps is around 14 GB in total). 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. 5, 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. 5 shows that EPS significantly outperforms the competitors on all four benchmarks especially when the queries are more challenging. Note that the y-axis scale is logarithmic. EPS is around 2-4 times faster than SUB-N-T (which does not guarantee optimal solutions) and 2-5 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 5 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 $|sp|$ to restrict the Polyanya search, it expands a smaller number of visible vertices 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. We remark that most of the first-move extractions (#FM) are incurred for non-taut pruning, e.g., to determine whether a path connecting $v' \in V_{opp}$ to $v \in V_{cur}$ is non-taut from the opposite end opp, we need to get the first-move v_i from v' to v and then check whether $\langle opp, v', v_i \rangle$ is

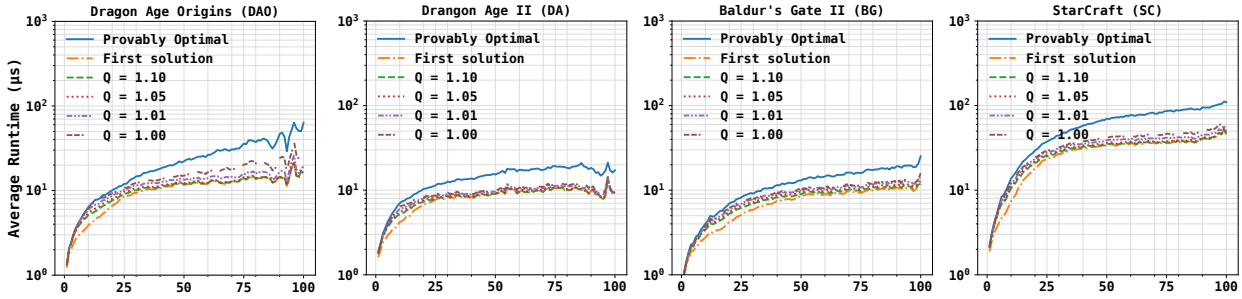
	EPS		No Caching & Pruning		Cost Caching Only		Pruning Only		Visible First	
Map	#FM	Time	#FM	Time	#FM	Time	#FM	Time	#FM	Time
DAO	329.463	23.905	23029.611	309.090	1981.007	50.342	363.923	24.512	335.019	29.674
DA2	144.872	14.186	6830.880	93.208	874.539	24.648	149.625	14.979	148.144	16.919
BG	98.955	12.299	2684.693	48.734	499.162	20.415	100.194	12.667	110.234	17.857
SC1	976.194	61.326	85617.378	1232.716	6298.061	177.680	994.956	62.375	1084.397	82.899

Table 6: Average query processing time (μs) and number of first move extractions (#FM) of our proposed EPS approach. We report results for each different pruning strategy.

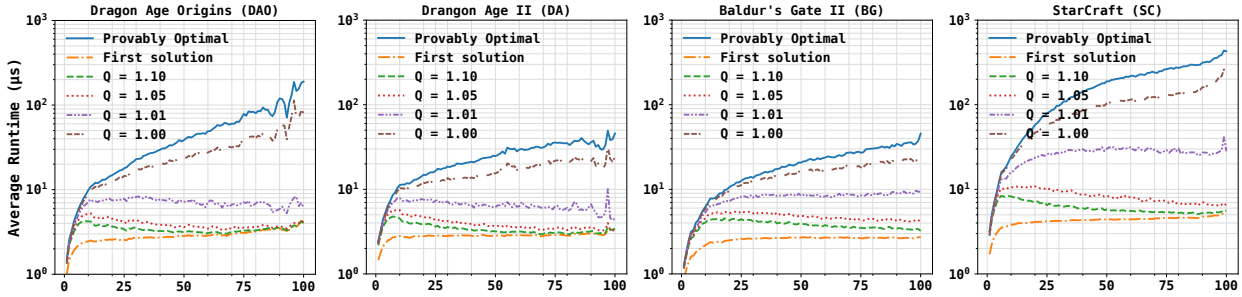
non-taut or not. Similarly, to check whether the path is non-taut from the *cur* end, we need to extract the first-move v_j from v to v' and check whether $\langle cur, v, v_j \rangle$ is non-taut or not. Although this non-taut pruning requires first-move extractions, overall it improves the performance as it reduces the number of paths extracted (#Paths) by EPS (which in turn reduces the overall number of first-move extractions needed to extract the paths). Also, we found that our previous implementation of EPS reported in [26] applied non-taut pruning only from the opposite end (*opp*). In this extended version, we updated the implementation by also applying non-taut pruning from the current end (*cur*) which significantly reduced both the number of paths and first-moves considered by EPS, resulting in an improved performance.

Experiment 3: Pruning Strategies. In Table 6, we show the average number of first move extractions and query processing time over four benchmark suites for: our final algorithm (EPS); we omit both pruning and caching (No Caching & Pruning); we use only cost caching but not pruning (Cost Caching Only); we use only pruning but not cost caching (Pruning Only); and we apply both pruning and caching but we first retrieve all visible vertices w.r.t. s and t (Visible First) instead of incrementally exploring them. Clearly, the reduction in search is significant and Pruning is the most important enhancement. Among the many ingredients of the Pruning techniques, we also observe that the non-taut path pruning is the most significant one. By performing non-taut path pruning on both ends, it allows the search to filter out most suboptimal candidate paths and quickly leads to the optimal one. On the other hand, Cost Caching Only essentially considers all the possible solutions and achieves a speedup vs. the baseline (i.e., No Caching & Pruning) by a few factors. Visible First requires the full insertions (i.e., finding all the visible vertices) which is time consuming, thus runs slower than EPS.

Experiment 4: 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. We begin with evaluating the anytime behaviour of EPS. In Fig. 6 (i), we show the average runtime of EPS to find: the first valid path (shown as First solution); the first path with length within a certain factor Q of the optimal path length (shown as $Q = 1.xx$); and the guaranteed optimal path (i.e., when EPS terminates). Here, $Q = 1.00$ is the time taken by EPS until it happens to discover the optimal path (although it cannot terminate because it cannot yet guarantee the optimality of this path). Recall that EPS makes use of pruning techniques to eliminate the candidate paths that are found to be non-optimal. Although this improves the runtime of EPS to find the optimal path, it adversely affects the performance of EPS for anytime search (e.g., the first valid path is found later as several candidate paths may have been pruned). This further motivates us to consider another variation of EPS for anytime search, where we consider all candidate paths and only use cost caching to improve the efficiency. We denote this as EPS (Cost Caching Only). Fig. 6 (ii) shows anytime search for EPS (Cost Caching Only). It is clear that EPS (Cost Caching Only) demonstrates excellent anytime behaviour, e.g., it finds the first valid path within 5μs and a path with $Q = 1.10$ (a path with length at most 10% longer than that of the optimal path) within 10μs. However, note that EPS (Cost Caching Only) understandably runs slower than EPS for the optimal path search (i.e., Provably Optimal) which demonstrates that the pruning rules provide a tradeoff

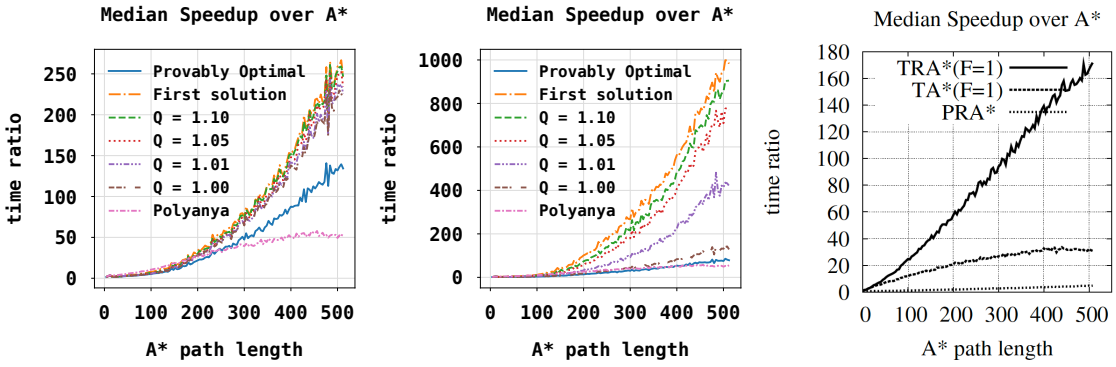


(i) EPS anytime behaviour



(ii) EPS (Cost Caching Only) anytime behaviour

Figure 6: (i) EPS and (ii) EPS (Cost Caching Only) anytime behaviour. The x-axis is the same as in Fig. 5. 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.



(i) EPS (ii) EPS (Cost Caching Only) (iii) TRA*

Figure 7: Speedup of (i) EPS and (ii) EPS (Cost Caching Only) over A* search for finding solutions of different quality on benchmark suite BG, and a reproduced graph for the same experiment for (iii) TRA* where ($F = 1$) represents the query performance that TRA* finds the first solution.

between optimal search and anytime search.

In Fig. 7 (i) and (ii), we show the speedup of anytime search compared to A* search for EPS and EPS (Cost Caching Only), respectively. Fig. 7 (iii) shows a graph reproduced from [6] showing similar comparison for TRA* anytime search, a popular mesh-based planner, which aims at finding the first solution fast. It can be seen that the speedup over A* search provided by EPS(Cost Caching Only) is significantly bigger

than those achieved by TRA*. For example, to find the first solution for queries with path length around 500, EPS(Cost Caching Only) is around 1000 times faster than A* whereas TRA* is around 180 times faster than A*. EPS provides a bigger speed up over A* compared to TRA* for queries with longer paths but a smaller speed up for the shorter queries. This is mainly because EPS focuses on finding the optimal solution fast and prunes many candidate paths which may result in a delay in finding the first solution.

4. Bounded Suboptimal Search

In this section, we present techniques to find a bounded suboptimal path. We focus on an absolute bound on the suboptimality, that is, $|bsp(s,t)| \leq |sp(s,t)| + \epsilon$ where ϵ is the error bound, $|bsp(s,t)|$ is the length of the bounded suboptimal path between s and t returned by our algorithm and $|sp(s,t)|$ is the shortest path distance between s and t. First, we show how to modify EPS to opportunistically terminate early when it finds a bounded suboptimal path. We then describe a search free method that builds a bounded suboptimal path using a larger CPD.

4.1. Bounded Suboptimal EPS Search

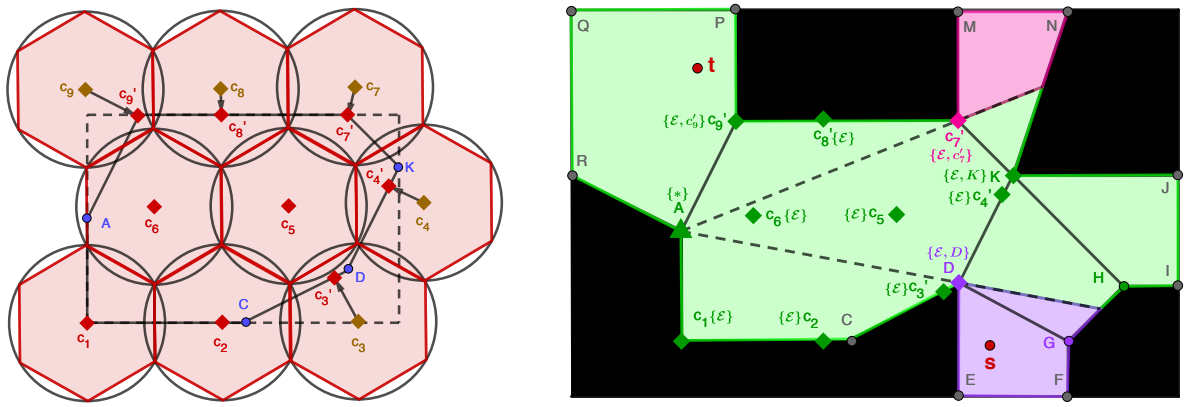
For efficient suboptimal search, it is important that the search can be terminated as soon as we find a candidate path that is within the bound $|sp(s,t)| + \epsilon$. Therefore, each candidate path found by the EPS becomes important which motives us to consider EPS (Cost Caching Only) for the suboptimal search; i.e., we consider all possible solutions found by EPS without applying our pruning rules and apply only the cost caching to improve the performance. Let $lb(s,t)$ be a lower bound on the length of the shortest path between s and t. EPS can terminate as soon as it finds a path $bsp(s,t)$ such that $|bsp(s,t)| \leq lb(s,t) + \epsilon$. Note that, given the lower bound, EPS can also be used to obtain a path with a relative bound by terminating EPS when $|bsp(s,t)| \leq lb(s,t) \times (1+\epsilon)$. One can use the Euclidean distance to obtain $lb(s,t)$ but it is not likely to be very effective. We propose a more effective lower bound as described below.

Recall that EPS uses CPD to compute the shortest distance between $v_s \in V_s$ and $v_t \in V_t$ denoted as $|cpd(v_s, v_t)|$ where v_s is visible from s and v_t is visible from t. Using the triangle inequality, it is easy to show that $|cpd(v_s, v_t)| - d(s, v_s) - d(t, v_t) \leq |sp(s,t)|$, i.e., $|cpd(v_s, v_t)| - d(s, v_s) - d(t, v_t)$ is a valid lower bound on the shortest distance. Whenever EPS uses CPD to compute the shortest distance between a pair of vertices (at line 8 in Algorithm 1), we compute the lower bound for the pair. The algorithm maintains $lb(s,t)$ to be the largest lower bound computed so far. Specifically, we initialise the $lb(s,t)$ to be the Euclidean distance $d(s,t)$, let P_v denotes the set of (v_s, v_t) pairs for which EPS has computed the shortest distance using CPD. Then, $lb(s,t) = max(lb(s,t), argmax_{(v_s,v_t)\in P_v} |cpd(v_s, v_t)| - d(s, v_s) - d(t, v_t))$.

Consider the example of Fig. 8 (ii) and assume that EPS uses CPD to compute the shortest distances between the pair of vertices (D,c'_9) and (E,P). Then, $lb(s,t) = |cpd(P,E)| - d(s,E) - d(t,P)$ because the pair (E,P) generates a better (larger) lower bound than that of (D,c'_9). For the rest of the paper, we denote the bounded (S)uboptimal EPS as SEPS.

4.2. Centroid-based Path Extraction

Although our SEPS may terminate earlier to find a path within a given bound ϵ, it still requires searching for the endpoints and computing distances between multiple pairs of vertices using CPD, which may be inefficient. This motivates the Centroid-based Path Extraction (CPE) algorithm, a search-free algorithm that provides a bounded suboptimal path by using the CPD to extract the path between only a single pair of points. The key idea of this algorithm is to fill in the navigation mesh with a set of candidate nodes, called centroids, such that for every point p in the space there exists a centroid visible from p and within distance $\delta = \epsilon/4$. Then, first-move tables are created using these centroids which are compressed to obtain a centroid-based CPD. To compute the shortest path, we find a centroid c_s (resp. c_t) that is within distance δ from s (resp. t) and use the CPD to compute the shortest path between c_s and c_t. We show that the path $\langle s, c_s, c_t, t\rangle$ is within the absolute bound ϵ. The path is further refined using string pulling.



(i) (ii)

Figure 8: (i) shows an example of our hexagonal tiling approach for a given polygon, where c_1 - c'_9 (shown as red) are the centroids selected for building CPD; and (ii) shows a partial example with centroids created for central polygon according to (i). 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. pink) area is `D` (resp. c'_7).

Algorithm 2: Centroid-based CPD Construction

Input : Navigation mesh
Output : A Centroid-based CPD
Initialization: $CN \leftarrow \emptyset$
1 **for** *each traversable polygon* P *of the mesh* **do**
2 $R \leftarrow$ minimum bounding rectangle of P;
3 **for** *each hexagon of the hexagonal tiling of* R **do**
4 $c_i \leftarrow$ centroid of the hexagon;
5 **if** c_i *is inside* P **then**
6 $CN \leftarrow CN \cup c_i$;
7 **if** *the hexagon overlaps* P *but* c_i *is outside* P **then**
8 $c'_i \leftarrow$ closest point of P from c_i;
9 $CN \leftarrow CN \cup c'_i$;
10 insert convex vertices of P in CN if not already present;
11 construct a CPD using CN as candidate nodes and **return**;

4.2.1. Building the Centroid-based CPD

We need to fill each polygon of the navigation mesh with centroids such that the circles centred at these centroids with radii δ cover every point in the polygon. The main objective here is to cover the polygon with a minimal number of such circles. This problem belongs to the class of covering problems many of which are shown to be NP-Complete [27]. Next, we describe a greedy algorithm (Algorithm 2) based on hexagon tiling.

First, we create a set of candidate nodes CN which will be used for CPD construction. CN is initially empty and the algorithm iteratively accesses each traversable polygon P of the navigation mesh (line 1) to populate CN as follows. Let R be the minimum bounding rectangle of the polygon. We use hexagonal tiling to cover the rectangle R such that the circumradius of each hexagon (i.e., the radius of its circumscribing circle) is δ, e.g., see the hexagon tiling in Fig. 8 (i). Since these hexagons completely cover the polygon, their circumscribing circles also completely cover the polygon. For each hexagon, we insert its centroid c_i in CN if the centroid is within the polygon P (line 6), e.g., in Fig. 8 (i), c_1, c_2, c_5, and c_6 are inside P and inserted in CN. If the hexagon overlaps P but its centre c_i is outside P, we find the closest point from c_i on P (denoted c'_i) and insert it in CN, e.g., c'_3, c'_4, c'_7, c'_8 and c'_9 in Fig. 8 (i) are inserted in CN. It is easy to

Algorithm 3: Centroid-based Path Extraction

Input : s: start, t: target, CPD: compressed-path-database
Output : a bounded suboptimal path from s to t
Initialization: $cp \leftarrow \emptyset$
1 $c_s \leftarrow$ getClosestCentroid(s);
2 $c_t \leftarrow$ getClosestCentroid(t);
3 $sp(c_s, c_t) \leftarrow cpd(c_s, c_t)$;
4 **refine** the path $sp(s, c_s) \oplus sp(c_s, c_t) \oplus sp(c_t, t)$ and **return**

show that the circles centred at these moved centroids still completely cover the polygon. Finally, all convex vertices of P are inserted in CN if not already present (line 10). In Fig. 8 (ii), A, D and K are inserted.

Once all traversable polygons are processed as described above, a CPD is constructed using the set of candidate nodes CN (line 11) by following the same procedure as described in Section 3.1.2. E.g., in Fig. 8 (ii) $CN = \{c_1, c_2, c'_3, c'_4, c_5, c_6, c'_7, c'_8, c'_9, \mathtt{A}, \mathtt{D}, \mathtt{K}\}$. Hereafter, we use "centroid" to refer to candidate nodes in CN. Note that this CPD stores first moves from each centroid to every other centroid in CN, and only the convex vertices can be the first moves between any pair of centroids. Also, if two centroids are visible from each other, $\mathcal{E}$ may be used as the first move. As shown later in our experimental study (Table 7), this results in an excellent compression and reduces the space by up to two orders of magnitude. This is because the symbols in the first move tables are either convex vertices of the navigation mesh or $\mathcal{E}$.

4.2.2. Online Search

Algorithm 3 outline the pseudocode of our CPE algorithm. First, we find the centroids c_s and c_t which are the closest centroids to s and t, respectively. Different approaches may be used to achieve this. We implement a grid-based fetching approach. Specifically, during pre-processing, we use a grid for each polygon in the navigation mesh where each cell of the grid stores every centroid of the polygon whose circle overlaps it. During online search, we identify the grid cell containing s (resp. t) and find the closest centroid recorded in this cell which is in the same polygon as s (resp. t). The lookup time is linear in the number of centroids that overlap this grid cell.

Once c_s and c_t are obtained, we compute the shortest path between c_s and c_t using the CPD. Since both c_s and c_t are nodes in CPD, the number of CPD look ups required to extract this path is linear in the number of vertices on the shortest path between c_s and c_t. Then, we obtain $sp(s, c_s) \oplus sp(c_s, c_t) \oplus sp(c_t, t)$ where $sp(s, t)$ denote the shortest path from s to t and $\oplus$ is a concatenation operator.

Theorem 2. *If the centroid-based CPD is constructed using $\delta = \epsilon/4$, then the length of $sp(s, c_s) \oplus sp(c_s, c_t) \oplus sp(c_t, t)$ is at most $|sp(s, t)| + \epsilon$.*

Proof. We show that the length of the path, $|bsp(s, t)| = d(s, c_s) + |cpd(c_s, c_t)| + d(c_t, t)$, is at most $|sp(s, t)| + \epsilon$. Due to the triangle inequality, we have $|cpd(c_s, c_t)| \leq d(c_s, s) + |sp(s, t)| + d(t, c_t)$. Therefore, we have $|bsp(s, t)| \leq 2 \cdot d(s, c_s) + |sp(s, t)| + 2 \cdot d(c_t, t)$. The hexagon tiling ensures that the circles centred at all centroids with radii δ cover every point in the map. Hence, for any points p, its closest centroid is no further than δ, i.e., $d(s, c_s) \leq \delta$ and $d(c_t, t) \leq \delta$. Therefore, we have $|bsp(s, t)| \leq |sp(s, t)| + 4\delta \leq |sp(s, t)| + \epsilon$. □

Consider the example of Fig. 9. We first get the closest centroids c_{18} and c_{13} from s and t, respectively. Then, the centroid-based CPD is used to extract an optimal path from c_{18} to c_{13} (i.e., $\langle c_{18}, \mathtt{O}, c_{13} \rangle$). The bounded suboptimal path is then $\langle s, c_{18}, \mathtt{O}, c_{13}, t \rangle$ (shown using blue lines in the figure).

Path Refinement: Although the path generated by the approach described earlier is bounded, we can further improve the path quality by string pulling. We describe how to refine the path from the source end. Let v be the first vertex after c_s on the unrefined path $sp(s, c_s) \oplus sp(c_s, c_t) \oplus sp(c_t, t)$. We use string pulling to refine the subpath $\langle s, c_s, v \rangle$. To implement the string pulling, we use a combination of the funnel algorithm [28] and our CPD. Specifically, we use the funnel algorithm to find the first turning point p on the refined path from s to v. Since p must be a vertex in the CPD, we can use the CPD to obtain the optimal

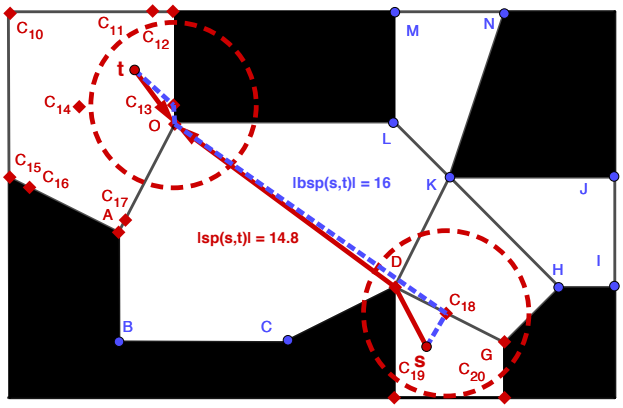


Figure 9: An example of Cenroid Path Extraction (CPE) that runs on a centroid-based CPD with $\delta = 3$. The bounded suboptimal path found by CPE is shown as blue. The final refined path of CPE, as well as the optimal path between s to t, are shown as red.

path from p to v. Thus, the subpath $\langle s, c_s, v\rangle$ is refined as $sp(s,p) \oplus sp(p,v)$. After the path is refined, we use a similar approach to refine the path from the target end.

To find the first turning point p from s to v, the funnel algorithm iteratively accesses the mesh edges that intersect with the subpath $\langle s, c_s, v\rangle$ in this order (from s to v). It also maintains a maximal visible interval $I = [v_l, v_r]$ where v_l and v_r correspond to the left and right most vertices, respectively, visible from s through the accessed edge. The algorithm terminates when the accessed edge is completely invisible from s via I and returns v_l (resp. v_r) if the invisible edge is on the left (resp. right) of $I = [v_l, v_r]$.

In the example of Fig. 9, the blue path is refined as follows. The first vertex on the blue path after c_{18} is O. Therefore, we use string pulling to refine the subpath $\langle s, c_{18}, \mathtt{O}\rangle$. The funnel algorithm follows the subpath $\langle s, c_{18}, \mathtt{O}\rangle$ and finds intersected mesh edges DG, DK and AO in this order. The algorithm initializes the maximal visible (from s) interval I as [D, G]. When it processes the next intersecting edge DK, it updates I to be [D, K] since the whole edge is visible from s. It then proceeds to process the edge AO. Since AO is completely invisible from s via I = [D, K] and AO is towards the left of I (i.e., node D), the funnel algorithm returns D as the first turning point on the path from s to O. We use the CPD to compute the path from D to O which happens to be $\langle \mathtt{D}, \mathtt{O}\rangle$. Therefore, the subpath $\langle s, c_{18}, \mathtt{O}\rangle$ is refined to $\langle s, \mathtt{D}, \mathtt{O}\rangle$. Similarly, the subpath from the target end (i.e., $\langle t, c_{13}, \mathtt{O}\rangle$) is refined to $\langle t, \mathtt{O}\rangle$. Thus, the final refined path is $\langle s, \mathtt{D}, \mathtt{O}, t\rangle$ (shown using red lines in the figure).

4.3. Experiments

We run experiments on the same benchmarks and machine as described in Section 3.3. We compare our suboptimal algorithms (SEPS and CPE) with SUB-N-T [24] (briefly described in Section 3.3). We also compare with SUB-N-A [24] which is the same as SUB-N-T except that it uses A* instead of Theta-A*. For our SEPS, we refine the path by using the same path refinement technique of CPE. i.e., given a path $sp(s, v_s) \oplus sp(v_s, v_t) \oplus sp(v_t, t)$ returned by SEPS and let v be the first turning vertex after v_s, we refine the subpath $\langle s, v_s, v\rangle$ if it is not taut. We refine the path in a similar way for the target end. For both SUB-N-T and SUB-N-A, as suggested in [24], we use path smoothing to improve the path quality.

Experiment 1: Centroid-based CPD Statistics. In order to construct them faster, our centroid-based CPDs were built on 32 cores Nectar research cloud with 64 GB of RAM and running Ubuntu 18.04 LTS (Bionic) amd64. Table 7 shows the average and maximal number of centroids, the raw size of the first move tables (e.g., without compression), as well as the size of the CPD with (i.e., M) or without $\mathcal{E}$ (i.e., $M \setminus \mathcal{E}$) symbols, and its building time for the four benchmark suites. Clearly, our centroid-based CPDs are cheap to build and memory efficient for large radii (i.e., $\delta = 4, 8$). On the other hand, when the radius is small (i.e., $\delta = 1, 2$), our centroid-based CPDs require more pre-processing time and space. In later experiments,

		CPE(δ=1)		CPE(δ=2)		CPE(δ=4)		CPE(δ=8)	
Map	Stat	Avg	Max	Avg	Max	Avg	Max	Avg	Max
DAO	C	14294.705	87898	5577.500	34404	3129.910	19059	2341.621	14079
	T	1.109	29.474	0.178	4.662	0.061	1.486	0.036	0.873
	R	1961.839	30904.233	293.112	4734.540	90.923	1452.981	50.574	792.872
	M\$\mathcal{E}$	122.745	1900.786	15.885	208.670	3.831	53.326	1.688	23.585
	M	27.318	536.143	6.467	120.143	2.308	38.942	1.213	19.043
DA2	C	10557.328	29008	4094.597	11701	2211.462	7343	1608.164	5959
	T	0.251	2.120	0.042	0.431	0.014	0.149	0.009	0.130
	R	651.578	3365.856	97.233	547.653	28.752	215.678	15.664	142.038
	M\$\mathcal{E}$	40.726	189.284	5.561	21.793	1.382	5.593	0.627	2.768
	M	7.921	30.164	2.033	7.231	0.762	3.018	0.418	1.780
BG	C	36570.453	94750	11590.760	31316	4511.653	15998	2449.186	11423
	T	2.763	26.548	0.250	2.080	0.047	0.589	0.019	0.332
	R	6100.325	35910.250	619.045	3922.767	102.595	1023.744	37.330	521.939
	M\$\mathcal{E}$	723.109	2049.922	142.178	1943.864	15.081	153.247	2.858	14.774
	M	74.253	442.378	14.857	67.309	3.807	19.536	1.390	10.161
SC	C	149261.360	469291	52441.946	173836	25505.546	92309	17263.480	66775
	T	130.440	1013.990	13.255	98.862	2.836	25.950	1.284	13.265
	R	114718.511	880936.170	14302.496	120875.819	3476.542	34083.805	1638.570	17835.602
	M\$\mathcal{E}$	1061.652	2099.440	502.036	2017.235	92.047	397.907	31.269	133.774
	M	635.166	3966.987	143.700	850.968	45.803	257.136	21.087	105.182

Table 7: Number of (C)entroids, building (T)ime in minutes, (R)aw memory (first-move tables without compression) in MB, and (M)emory usage without/with ($\mathcal{E}$) symbol in MB, for different radius δ over four benchmarks.

we will show that the variation of radius in the centroid-based CPDs has only a minor effect on the running time of our CPE algorithm, but may result in worse path quality for larger radius. Therefore, there is a trade-off between the sub-optimality requirements and the memory limitations and a decision regarding the radius can be made considering the specific requirements of the application.

Compared with the raw memory, our CPD compression reduces the size of the first move tables by up to five hundred times, and allows CPDs even with a small radius (i.e., $\delta = 1$ or 2) to fit into the main memory of modern devices. In addition, we also observe that the $\mathcal{E}$ symbols significantly improve the size by a few factors on average, especially when the radius is small (i.e., $\delta = 1$ or 2), because there exists many centroids that are co-visible and without the $\mathcal{E}$ symbols, they do not compress well.

Experiment 2: Query Processing Time. In Table 8, we compare the average query processing time for our CPE algorithm using different radii (i.e., δ = 1, 2, 4, and 8) against the competitors. Table 8 shows that CPE significantly outperforms all competitors in all settings. Specifically, CPE is around one order of magnitude faster than SUB-N-T, SUB-N-A and optimal EPS, and is several times faster than the suboptimal EPS, i.e., SEPS(ϵ=32), SEPS($\epsilon = 8\%$). The SEPS with relative bound $\epsilon = 8\%$ matches the same sub-optimality with SUB-N-T and SUB-N-A, because the shortest grid paths are at most 8% longer than the optimal any-angle path [4]. We also observe that the query processing time of CPE is not significantly affected with the change in δ. This is because CPE needs to extract only one path using the CPD and the size of the CPD does not significantly affect the performance because the number of first move extractions is small. Also, the query time of CPE is dominated by the cost of getting the closest centroids from s and t. Note that the path refinement is cheap for both CPE and SEPS which is mainly because, we only use string pulling to find the first turning vertex, and make use of CPDs to quickly recover the rest of the path. The refinement time for CPE is even cheaper because, in most of the cases, the first vertex after c_s (resp. before c_t) is visible from s (resp. t) resulting in cheap string pulling that behaves like a simple line-of-sight check.

In Fig. 10, we extend the comparison to show the cactus plots for each competing algorithm over four different benchmarks. Note that for our CPE and SEPS algorithms, we also include the path refinement

	CPE(δ=1)		CPE(δ=2)		CPE(δ=4)		CPE(δ=8)		SEPS(ϵ=32)		SEPS(ϵ=8%)		EPS	SUB-N-T	SUB-N-A
Map	Q	R	Q	R	Q	R	Q	R	Q	R	Q	R	Q	Q	Q
DAO	2.244	0.767	2.138	0.758	2.130	0.786	2.166	0.807	4.536	1.459	5.141	1.410	23.905	81.518	31.820
DA2	2.212	0.654	2.083	0.643	2.047	0.657	2.048	0.681	4.121	1.283	4.977	1.215	14.186	31.296	15.458
BG	1.977	0.697	1.912	0.737	1.882	0.690	1.848	0.697	4.051	1.089	4.922	1.026	12.299	33.830	14.328
SC	3.985	1.509	3.523	1.379	3.459	1.547	3.462	1.618	22.049	1.731	14.937	1.766	61.326	252.962	83.379

Table 8: Runtime comparison on the four benchmarks: we show the average (Q)uery processing time (µs) and path (R)efinement time (µs) for Centroid-based Path Extraction (CPE), and compare with: the Suboptimal EPS (SEPS) with an absolute bound ϵ=32 (the same sub-optimality bound as $\delta = \epsilon/4 = 8$) and with a relative bound $\epsilon = 8\%$ (the same sub-optimality bound as SUB-N-T and SUB-N-A); the optimal EPS; and two subgoal graph methods SUB-N-T and SUB-N-A.

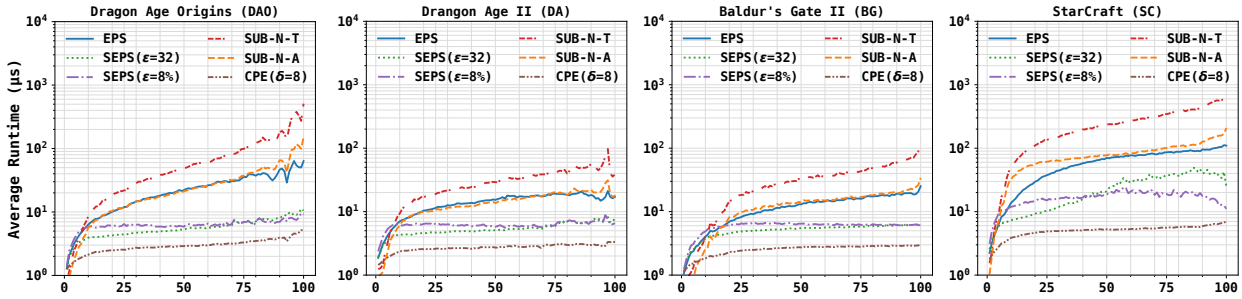


Figure 10: Runtime comparison on the four benchmarks. The x-axis shows the percentile ranks of queries in number of node expansions needed by A* search to solve them.

time. Fig. 10 shows that both CPE and SEPS are significantly faster than EPS, SUB-N-T and SUB-N-A. CPE is the best performing algorithm and scales really well with the increasing difficulty of the queries.

Experiment 3: Path Quality. In this experiment, we compare the path quality for CPE and the suboptimal EPS against the competitors. Our evaluation is based on the following three measures: (1) The (O)ptimal ratio (%) that corresponds to the percentage of instances when an algorithm retrieved optimal paths, i.e., number of queries for which an algorithm returns an optimal path $\times$ 100 /total number of queries; (2) The (S)uboptimality ratio (%) which is the difference between the length of the retrieved path and the optimal path divided by the length of optimal path (i.e., (cost(retrieved path) $-$ cost(optimal path)) $\times$ 100 / cost(optimal path)); (3)The path (D)ifference which is the difference between the length of the retrieved path and optimal path. (i.e., cost(retrieved path) $-$ cost(optimal path)).

Effectiveness of path refinement in CPE and SEPS: In Table 9, we compare the path quality before and after path refinement for our CPE and SEPS algorithms. Clearly, our path refinement strategy significantly improves the path quality on all three measures. Before the path refinement, the optimality ratio for both CPE and SEPS are low because: the retrieved path by CPE can never be optimal unless the closest centroids of both s and t are either turning points on the path or s and t lie on centroids; and SEPS essentially terminates as soon as it satisfies the suboptimality constraint. However, it is clear that the optimality ratio is significantly improved after the path refinement and, in fact, most of the paths returned by CPE are optimal. In addition, we observe that the path refinement also significantly reduces the average suboptimality ratio (S) and difference (D). Thus, the path refinement is cheap (see Table 8) and very effective in improving the path quality. Finally, note that the average path difference of our CPE both before and after path repair is much smaller than the theoretical bound (i.e., $4 \times \delta$). Similarly for SEPS, the theoretical error bound ϵ is also larger than the average suboptimality ratio and path difference reported.

Comparison of different algorithms: Table 10 compares the algorithms on the optimal ratio (O) as

		CPE(δ=1)		CPE(δ=2)		CPE(δ=4)		CPE(δ=8)		SEPS(ϵ=32)		SEPS(ϵ=8%)	
Map	Stat	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After
DAO	O(%)	3.110	95.291	3.140	89.409	3.061	80.893	3.077	70.393	3.931	39.283	9.067	44.413
	S(%)	1.021	0.001	1.953	0.008	3.398	0.029	5.235	0.072	14.332	1.431	2.888	0.366
	D	1.030	0.002	1.990	0.009	3.565	0.031	5.744	0.088	12.771	1.525	10.241	1.126
DA2	O(%)	2.315	96.486	2.342	91.925	2.253	84.061	2.335	74.797	5.625	47.126	12.013	54.842
	S(%)	1.036	0.001	1.982	0.005	3.597	0.022	5.661	0.066	14.563	1.340	2.672	0.272
	D	1.059	0.001	2.053	0.006	3.714	0.024	5.974	0.074	12.339	1.198	6.750	0.663
BG	O(%)	6.053	97.988	6.053	95.392	6.047	91.105	6.080	85.045	9.153	49.024	16.868	56.305
	S(%)	0.861	<0.001	1.671	0.002	3.159	0.008	5.352	0.030	12.964	1.292	3.191	0.383
	D	1.082	0.001	2.105	0.003	4.020	0.015	7.020	0.052	15.131	1.899	8.478	1.058
SC	O(%)	2.161	97.570	2.171	94.275	2.170	88.506	2.169	79.565	7.272	26.844	8.252	27.222
	S(%)	0.473	<0.001	0.915	0.001	1.715	0.005	3.088	0.020	6.856	1.447	3.205	0.703
	D	1.117	0.001	2.166	0.004	4.104	0.016	7.372	0.059	13.842	3.073	16.921	3.457

Table 9: Our Centroid-based Path Extraction (CPE) and Suboptimal EPS (SEPS), before and after path refinement. We show (O)ptimal ratio (%): i.e., # optimal path × 100 / # queries; average (S)uboptimality ratio (%): i.e., (cost(retrieved path) − cost(optimal path)) × 100 / cost(optimal path) and (D)ifference: i.e., cost(retrieved path) − cost(optimal path).

		CPE(δ=1)		CPE(δ=2)		CPE(δ=4)		CPE(δ=8)		SEPS(ϵ=32)		SEPS(ϵ=8%)		SUB-N-T		SUB-N-A	
Map	Stat	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
DAO	O(%)	95.291		89.409		80.893		70.393		39.283		44.413		56.303		35.767	
	S(%)	0.001	12.759	0.008	19.731	0.029	141.421	0.072	89.442	1.431	1623	0.366	7.922	0.093	4.875	0.155	4.945
	D	0.002	1.817	0.009	4.007	0.031	6.878	0.088	11.819	1.525	31.834	1.126	53.105	0.504	7.247	0.732	9.642
DA2	O(%)	96.486		91.925		84.061		74.797		47.126		54.842		63.949		48.312	
	S(%)	0.001	7.701	0.005	6.893	0.022	20.893	0.066	78.046	1.340	470	0.272	7.973	0.096	3.760	0.129	5.475
	D	0.001	1.926	0.006	2.985	0.024	6.916	0.074	10.441	1.198	30.491	0.633	63.288	0.354	6.015	0.438	7.295
BG	O(%)	97.988		95.392		91.105		85.045		49.024		56.305		82.958		66.343	
	S(%)	<0.001	14.412	0.002	14.412	0.008	14.412	0.030	29.724	1.292	570	0.383	7.797	0.061	5.580	0.115	5.580
	D	0.001	1.349	0.003	5.468	0.015	6.436	0.052	15.443	1.899	31.622	1.058	35.113	0.188	14.118	0.348	16.217
SC	O(%)	97.570		94.275		88.506		79.565		26.844		27.222		54.780		27.952	
	S(%)	<0.001	4.696	0.001	8.270	0.005	17.082	0.020	49.998	1.447	554	0.703	7.848	0.076	5.156	0.175	5.156
	D	0.001	2.576	0.004	3.948	0.016	10.126	0.059	18.062	3.073	30.939	3.457	109.386	0.526	17.580	1.120	19.800

Table 10: Comparing CPE, SEPS, SUB-N-T and SUB-N-A on (O)ptimal ratio (%) as well as average/maximum (S)uboptimality ratio (%) and (D)ifference.

well as average and maximum suboptimality ratio (S) and path difference (D). Clearly, CPE demonstrates excellent path quality for all different benchmarks on all measures, e.g., the optimal ratio, average suboptimality ratio and average path difference are better than those of SUB-N-T, SUB-N-A, SEPS($\epsilon = 32$) and SEPS($\epsilon = 8\%$) for different values of δ and remarkably better for smaller δ. SUB-N-A and SUB-N-T are better in terms of *maximum* suboptimality ratio. Since CPE returns a path within an absolute bound, the path quality in terms of suboptimality ratio may be very poor when the start and target are close to each other but δ is large (i.e., 4 or 8). On the other hand, CPE usually has a smaller *maximum* path difference (D) compared to SUB-N-T for $\delta = 1$, 2 or 4. For the SEPS with an absolute bound, SEPS($\epsilon = 32$), we see that the maximum path difference is always close to the theoretical bound ϵ as our SEPS greedily terminates the search as soon as it explores a path that satisfies the constraint (note that the refinement does not improve the path quality if the returned path is taut on both ends). This also causes the suboptimality ratio of the absolute bounded SEPS to be large, especially in terms of maximum suboptimality ratio (which is mainly due to the queries when start and target are close to each other). The SEPS with a relative

bound, SEPS($\epsilon = 8\%$), has the advantage that it allows the search to control the suboptimality ratio within a certain percentage. However, in this case, it results in large absolute differences in the worst-case (for queries when start and target are far from each other).

5. Related work

In this section, we categorise the related works into three four categories: any-angle pathfinding; visibility graph-based path planning; and mesh-based path planning; and centroids-based path planning.

Any-angle Pathfinding: A grid map is a map representation that is commonly used in computer games and robotics. The grid map discretises the set of obstacles using a fixed resolution grid and represents the environment as blocked and unblocked cells. Typically, pathfinding on grid maps results in suboptimal and unrealistic paths as movement is only allowed in certain directions (i.e., horizontal and vertical in a four-connected grid, or in addition diagonal movement in an eight-connected grid). Any-angle pathfinding avoids this restriction by allowing movement in any direction and, consequently, finds the paths that are substantially shorter.

The state of art any-angle pathfinder, ANYA [5], is a fast and online pathfinding algorithm that finds optimal any-angle path. Anya is an interval-based searching algorithm that instantiates A*. It scans the grid row by row and finds an optimal path by expanding the most promising intervals in the open list.

Apart from the optimal algorithm Anya, there is also a range of algorithms that find suboptimal any-angle paths. The state-of-art any-angle suboptimal algorithm is the N-level subgoal graph [24]. The subgoal graph [12] is constructed by placing subgoals on the convex corners of obstacles. Instead of connecting each pair of subgoals that are visible to each other, the subgoal graph only connects a small subset of subgoals that are *direct-h-reachable*. Two subgoals are *h-reachable* if there exists a traversal path equal to the octile distance between them, and they are *direct-h-reachable* iff all shortest trajectories (octile distance paths) are traversable with no subgoals between them. The N-level subgoal graph builds, on top of this simple subgoal graph, a hierarchy that is similar to a contraction hierarchy [29]. Searching in the N-level subgoal graph only requires connecting the start and target to the corresponding direct-h-reachable subgoals, and then identifying the reachable subgoals from the start to the target through the ascending edges (edges from a subgoal to its higher-level subgoals). Using the graph consisting of hierarchy of subgoals, one can simply apply an A* algorithm to find a path quickly, or alternatively apply the Theta* [30] algorithm to search for a path with better path quality as the edges in this graph may be non-taut. Other way of improving path quality is to extend the subgoal graph by considering 2^k grid neighbours [31], where the path quality can be controlled using the value of k. The N-level subgoal graph outperforms a range of A*-like suboptimal algorithms for any-angle pathfinding such as: Theta* [30] which improves the path quality by performing a line-of-sight check when expanding a search node; Field A* [32] which generalises the ideas from Field D* [33], allowing a straight-line trajectory from a search node to any of vertices on the boundary of its adjacent grids, and updates g-value by using linear interpolation; and Block A* [34] which performs a blocks-based A* search with Local Distance Database, an auxiliary data structure that partitions grid map into $m \times n$ equal size of blocks and stores the distance of optimal paths between any two boundary vertices inside each block.

Visibility Graphs (VG): A visibility graph [35] is constructed by connecting any pairs of co-visible vertices appearing on the corners of the obstacles. In contrast to grid-based approaches that discretise the obstacles using a grid, the visibility graph allows obstacles to be precisely represented in Euclidean space. Hence, it finds the Euclidean optimal path precisely (which is guaranteed to be equal to or shorter than the any-angle path on grid maps). Search in a visibility graph only requires connecting the start and target to their set of visible vertices before an A* algorithm can be applied to find the Euclidean optimal path. However, the visibility graph suffers from two major issues: (1) it requires high memory consumption because, in the worst-case, the size of the graph is quadratic in the number of corners of the obstacles; and (2) branching factors of nodes in the graph is high and, consequently, the search is unlikely to be efficient.

Sparse visibility graphs (SVG) [8] focus on addressing the above mentioned two major issues and improve over the original visibility graph by removing unnecessary edges (i.e., a non-turn edge $v_i v_j$ where the

angle from v_i to v_j does not allow turning around the polygon defining vertex v_j) and non-convex vertices. This modification allows the graph to fit into the memory even for the large maps. It also reduces the branching factor on graph nodes and makes search more efficient. Edge N-Level Sparse Visibility Graphs (ENLSVG) [8] build a hierarchy by iteratively removing the non-taut paths. Such a hierarchy partitions the SVG into multiple levels where, in each level, edges only have taut neighbouring paths to the edges in the higher levels. ENLSVG restricts the search to only consider the edges that increase the levels from both ends. This results in searching in a smaller taut-path graph which improves the performance further.

Some existing works such as SUB-N-T/SUB-N-A [24, 12] and ENLSVG [8] improve the underlying graph by building a hierarchy similar to the contraction hierarchies (CH) [29], and enhance the performance by allowing the search to consider "shortcuts" between different levels of the hierarchy. As shown in a recent paper [36] on road networks, constructing a CPD on top of these hierarchies improves the performance as this allows the first-move to be a shortcut which effectively reduces the number of first-move extractions needed to recover the whole path. We believe that such a hierarchy-compressed CPD should also improve our EPS for optimal and suboptimal search and CPE for suboptimal search. We leave it as future work.

Mesh-based Planners: Mesh-based planners work by pre-processing the non-obstacle regions of the map into a set of convex polygons, called a *navigation mesh*. Such techniques combine the strengths of any-angle pathfinding and visibility graph. The state-of-art mesh-based path planner is Polyanya [7] which is a fast, online, and optimal pathfinding algorithm that extends and generalises Anya. We present the details of Polyanya in Section 2.2. Polyanya outperforms a range of pathfinding algorithms that work on Constrained Delaunay Triangulation [37] (CDT), a type of navigation mesh where the non-obstacle regions are modelled as triangles. We discuss a number of these methods below.

Channel search [38] finds a shortest channel between start and target from the CDT by using a modified A* algorithm. The search begins from the triangle that contains the start, and always considers the midpoint of the non-constrained edges as neighbours to expand the search nodes. The search terminate when it reaches the target and applies the funnel algorithm [28] to retrieve a local optimal path within the channel. Channel search is fast and easy to implement but can return suboptimal paths. Triangulation A* (TA*) [6] works in a way similar to the channel search. However, instead of terminating the search immediately, TA* takes the cost of the best path found so far as an upper bound. It continuously explores the most promising channel and updates the upper bound. TA* terminates when either the search is exhausted or the lower bound (i.e., f-value) of the search becomes greater than this upper bound. Thus, it guarantees to find the Euclidean optimal path for any given start and target. Triangulation Reduction A* (TRA*) [6] enhances TA* by preprocessing the CDT into an abstract graph which is small, but allows the search to find the most promising channel quickly. Note that both TA* and TRA* are any-time algorithms since these algorithms can return suboptimal paths encountered during the search. Both TA* and TRA* are significantly outperformed by Polyanya [7], therefore, we do not directly compare with these in this paper.

Hechenberger et al. propose RayScan [22], a ray-casting based ESPP algorithm that is purely online and requires no additional structure to support the search (not even a navigation mesh). The main idea of the algorithm is to use ray-casting from the current search node towards the target until an obstacle impedes the ray in which case the algorithm scans the perimeter of the obstacle to find the turning points. RayScan runs slower than Polyanya if the navigation mesh is already available. However, RayScan is useful when the navigation mesh is not available (or when the environment is dynamically updated which makes the existing mesh invalid requiring a repair of the mesh).

Centroids: The use of centroids to generate bounded suboptimal paths has been previously explored for grid maps [39]. There the focus is avoiding creating an enormous CPD for large grid maps, by using centroids to act as abstract target points, and the main technical emphasis is on using "reverse" CPDs and producing these only for centroid targets. Unlike the traditional CPD which compresses first-moves table for a given source s containing first-moves from s to every $t \in V$, a reverse CPD compresses first-moves table for a given target t containing first-moves to t from every $s \in V$. In contrast to [39], we use centroids to partition Euclidean space, and build a traditional CPD on the graph of centroids to allow fast bounded suboptimal

path planning, by connecting start and target points to the centroid graph.

6. Conclusions

We introduce new approaches to Euclidean path finding based on Compressed Path Databases (CPD). Our optimal algorithm, End Point Search (EPS), substantially improves the state-of-the-art for optimal Euclidean shortest paths and also has 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. The bounded suboptimal variant, Centroid-based Path Extraction (CPE), is several times faster than EPS for finding (absolute) bounded suboptimal paths. It allows us to tradeoff the suboptimality bound versus the size of the resulting CPD. In practice its behaviour is much better than the theoretical bound, with ≈90% of paths found being optimal for $\delta = 2$.

Acknowledgements

Muhammad Aamir Cheema is supported by Australian Research Council (ARC) FT180100140 and DP180103411. Daniel D. Harabor is supported by ARC DP190100013.

References

[1] T. T. Mac, C. Copot, D. T. Tran, R. De Keyser, Heuristic approaches in robot path planning: A survey, Robotics and Autonomous Systems 86 (2016) 13–28.
[2] M. A. Cheema, Indoor location-based services: challenges and opportunities, SIGSPATIAL Special 10 (2) (2018) 10–17.
[3] N. R. Sturtevant, Moving path planning forward, in: International Conference on Motion in Games, Springer, 2012, pp. 1–6.
[4] A. Nash, S. Koenig, Any-angle path planning, AI Magazine 34 (4) (2013) 9.
[5] D. D. Harabor, A. Grastien, D. Öz, V. Aksakalli, Optimal any-angle pathfinding in practice, Journal of Artificial Intelligence Research 56 (2016) 89–118.
[6] D. Demyen, M. 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, AAAI Press, 2006, pp. 942–947.
[7] M. Cui, D. D. Harabor, A. 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, ijcai.org, 2017, pp. 496–502.
[8] S. Oh, H. W. 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, AAAI Press, 2017, pp. 64–72.
[9] E. A. Hansen, R. Zhou, Anytime heuristic search, Journal of Artificial Intelligence Research 28 (2007) 267–297.
[10] A. 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.
[11] B. Strasser, D. Harabor, A. 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.
[12] T. Uras, S. Koenig, C. Hernández, Subgoal graphs for optimal pathfinding in eight-neighbor grids, in: Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013, AAAI, 2013.
[13] AI Game Programming Wisdom 4, Charles River Media, 2008.
[14] M. Kallmann, M. Kapadia, Navigation meshes and realtime dynamic planning for virtual worlds, in: ACM SIGGRAPH 2014 Courses, ACM Press, 2014, p. 3.
[15] M. Kallmann, Path Planning in Triangulations, in: IJCAI Workshop on Reasoning Representation and Learning in Computer Games, 2005.
[16] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE transactions on Systems Science and Cybernetics 4 (2) (1968) 100–107.
[17] M. Chiari, S. Zhao, A. Botea, A. E. Gerevini, D. Harabor, A. Saetti, M. Salvetti, P. 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, AAAI Press, 2019, pp. 106–113.

[18] B. Strasser, A. Botea, D. Harabor, Compressing optimal paths with run length encoding, Journal of Artificial Intelligence Research 54 (2015) 593–629.
[19] Y.-H. Liu, S. Arimoto, Path Planning Using a Tangent Graph for Mobile Robots Among Polygonal and Curved Obstacles, International Journal of Robotics Research 11 (1992) 376–382.
[20] T. Young, Optimizing Points-of-Visibility Pathfinding, in: Game Programming Gems 2, Charles River Media, 2001, pp. 324–329.
[21] R. C. Holte, A. Felner, G. Sharon, N. 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, AAAI Press, 2016, pp. 3411–3417.
[22] R. Hechenberger, P. J. Stuckey, D. Harabor, P. L. Bodic, M. A. Cheema, Online computation of euclidean shortest paths in two dimensions, in: Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020, AAAI Press, 2020, pp. 134–142.
[23] N. R. Sturtevant, Benchmarks for grid-based pathfinding, IEEE Transactions on Computational Intelligence and AI in Games 4 (2) (2012) 144–148.
[24] T. Uras, S. 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, AAAI Press, 2015, pp. 234–238.
[25] A. Nash, K. Daniel, S. Koenig, A. 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, AAAI Press, 2007, pp. 1177–1183.
[26] B. Shen, M. A. Cheema, D. Harabor, P. J. Stuckey, Euclidean pathfinding with compressed path databases, in: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, ijcai.org, 2020, pp. 4229–4235.
[27] R. J. Fowler, M. Paterson, S. L. Tanimoto, Optimal packing and covering in the plane are np-complete, Inf. Process. Lett. 12 (3) (1981) 133–137.
[28] J. Hershberger, J. Snoeyink, Computing minimum length paths of a given homotopy class, Comput. Geom. 4 (1994) 63–97.
[29] R. Geisberger, P. Sanders, D. Schultes, D. Delling, Contraction hierarchies: Faster and simpler hierarchical routing in road networks, in: Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings, Vol. 5038 of Lecture Notes in Computer Science, Springer, 2008, pp. 319–333.
[30] A. Nash, K. Daniel, S. Koenig, A. 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, AAAI Press, 2007, pp. 1177–1183.
[31] N. Hormazábal, A. Díaz, C. Hernández, J. A. Baier, Fast and almost optimal any-angle pathfinding using the 2^k neighborhoods, in: Proceedings of the Tenth International Symposium on Combinatorial Search, SOCS 2017, 16-17 June 2017, Pittsburgh, Pennsylvania, USA, AAAI Press, 2017, pp. 139–143.
[32] T. Uras, S. 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, AAAI Press, 2015, pp. 206–211.
[33] D. Ferguson, A. Stentz, Field d*: An interpolation-based path planner and replanner, in: Robotics Research: Results of the 12th International Symposium, ISRR 2005, October 12-15, 2005, San Francisco, CA, USA, Vol. 28 of Springer Tracts in Advanced Robotics, Springer, 2005, pp. 239–253.
[34] P. Yap, N. Burch, R. C. Holte, J. Schaeffer, Block a*: Database-driven search with applications in any-angle path-planning, in: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011, AAAI Press, 2011.
[35] T. Lozano-Pérez, M. A. Wesley, An algorithm for planning collision-free paths among polyhedral obstacles, Commun. ACM 22 (10) (1979) 560–570.
[36] B. Shen, M. A. Cheema, D. D. Harabor, P. J. Stuckey, Contracting and compressing shortest path databases, in: Proceedings of the International Conference on Automated Planning and Scheduling, Vol. 31, 2021, pp. 322–330.
[37] L. P. Chew, Constrained delaunay triangulations, Algorithmica 4 (1) (1989) 97–108.
[38] M. Kallmann, Path planning in triangulations, in: Proceedings of the IJCAI workshop on reasoning, representation, and learning in computer games, 2005, pp. 49–54.
[39] S. Zhao, M. Chiari, A. Botea, A. E. Gerevini, D. Harabor, A. Saetti, P. J. Stuckey, Bounded suboptimal path planning with compressed path databases, in: Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020, AAAI Press, 2020, pp. 333–342.