Ultrafast Euclidean Shortest Path Computation using Hub Labeling

Jinchun Du, Bojie Shen, Muhammad Aamir Cheema

Faculty of Information Technology, Monash University, Melbourne, Australia
{jinchun.du, aamir.cheema, bojie.shen}@monash.edu

Abstract

Finding shortest paths in a Euclidean plane containing polygonal obstacles is a well-studied problem motivated by a variety of real-world applications. The state-of-the-art algorithms require finding obstacle corners visible to the source and target, and need to consider potentially a large number of candidate paths. This adversely affects their query processing cost. We address these limitations by proposing a novel adaptation of hub labeling which is the state-of-the-art approach for shortest distance computation in road networks. Our experimental study conducted on the widely used benchmark maps shows that our approach is typically 1-2 orders of magnitude faster than two state-of-the-art algorithms.

Introduction

Given a source s and a target t in a Euclidean plane containing polygonal obstacles, we study Euclidean Shortest Path Problem (ESPP) which is to compute the shortest obstacle avoiding path between s and t. Due to its applications in robotics (Mac et al. 2016), indoor location-based services (Cheema 2018) and computer games (Sturtevant 2012b), efficiently computing ESPP has been extensively studied. Some of the most notable works include any-angle pathfinding approaches such as Anya (Harabor et al. 2016), navigation-mesh-based techniques such as Polyanya (Cui, Harabor, and Grastien 2017), advanced visibility graph techniques such as hierarchical sparse visibility graph (Oh and Leong 2017), and the techniques based on distance oracles such as End Point Search (EPS) (Shen et al. 2020).

EPS is the state-of-the-art ESPP algorithm and provides very efficient query processing by exploiting a *compressed path database* (CPD) (Botea 2011). However, as we explain in the next section, it still requires Polyanya to find the obstacle corners (i.e., vertices) visible from s and t, and needs to consider $|V_s| \times |V_t|$ candidate paths where $|V_s|$ and $|V_t|$ are the number of vertices visible from s and t, respectively.

We present an approach addressing the above-mentioned limitations. Our approach utilises hub labeling (Abraham et al. 2011) which computes a set of labels for each vertex in a graph such that the shortest distance between any two vertices can be computed efficiently (specifically, in time linear to the number of labels stored for the two vertices). Hub labeling is the state-of-the-art approach for efficiently computing shortest distances in graphs such as road networks (Li et al. 2017) but it has not been previously used for solving ESPP. Our experimental study shows that a straightforward application of hub labeling for ESPP does not improve EPS as the algorithm still needs to consider $|V_s| \times |V_t|$ candidate paths. We overcome this limitation by a novel adaptation of hub labeling and create labels for Euclidean plane to allow efficient query processing. Below we provide an overview of our approach called Euclidean Hub Labeling (EHL).

- During the preprocessing, we construct a visibility graph on the convex corners/vertices of the polygonal obstacles. Hub labels for each vertex in this graph are computed. We superimpose a uniform grid on the Euclidean plane and copy the labels of each vertex v onto each cell c if c is visible from v. Several novel pruning rules are applied to significantly reduce the number of labels stored.
- During online query processing, we use the labels stored in the grid cells c_s and c_t containing s and t, respectively. These labels are joined in time linear in the number of labels in the two cells and the shortest distance is obtained. Additional auxiliary information stored in each label is used to efficiently recover the shortest path.

We evaluate EHL using a variety of widely used grid map benchmarks (Sturtevant 2012a) and compare against Polyanya and EPS, the two state-of-the-art algorithms. Our results indicate that EHL provides 1-2 orders of magnitude faster query performance compared to Polyanya and EPS at the expense of larger (but reasonable) preprocessing time and memory. We also show that the grid size used by EHL provides a trade-off between the preprocessing cost (build time and memory) and query performance.

Preliminaries

A **polygon** representing an obstacle is defined by a set of points called vertices. A **convex vertex** is a vertex that is located at a convex corner of the polygon. A **non-convex vertex** is located at a concave corner. A pair of points in the plane are **visible** to each other (also called **co-visible**) iff a straight line connecting them does not pass through any obstacle. A **path** $\mathcal{P}$ between a source s and a target t is an ordered set of points $\langle p_1, p_2, \cdots, p_n\rangle$ such that, for each

p_i ($i < n$), p_i and p_{i+1} are co-visible where $p_1 = s$ and $p_n = t$. The **length** of a path $\mathcal{P}$ is the cumulative Euclidean distance between every successive pair of points, denoted as $|\mathcal{P}|$, i.e., $|\mathcal{P}| = \sum_{i=1}^{k-1} Edist(p_i, p_{i+1})$ where $Edist(x, y)$ is the Euclidean distance between x and y. Given s and t in a Euclidean plane containing polygonal obstacles, the **Euclidean Shortest Path Problem (ESPP)** is to compute an obstacle-avoiding path between s and t with the minimum length, denoted as $sp(s, t)$. The shortest distance between s and t, denoted as $d(s, t)$, is the length of the shortest path, i.e., $d(s, t) = |sp(s, t)|$.

Related Work

Hub Labeling: Hub Labeling (Abraham et al. 2011) is the state-of-the-art approach for computing shortest distances in graphs such as road networks. Assume an undirected graph $G = (V, E)$ with vertices V (vertices are also called nodes in this paper) and edges $E \subseteq V \times V$. Hub labeling computes and stores, for each vertex $v_j \in V$, a set of hub labels denoted as $H(v_j)$. Each hub label is a tuple $(h_i, d_{ij}) \in H(v_j)$ containing: (i) a hub vertex $h_i \in V$; and (ii) the shortest distance d_{ij} between the hub vertex h_i and v_j. The hub labels in each $H(v_j)$ are kept sorted according to the hub vertices. The hub labeling must satisfy the *coverage property*, i.e., for every pair of reachable vertices $v_j \in V$ and $v_k \in V$, $H(v_j)$ and $H(v_k)$ must contain at least one hub vertex h_i on the shortest path from v_j to v_k. Thus, the shortest distance between any $v_s \in V$ and any $v_t \in V$ can be computed as:

$$d(v_s, v_t) = \min_{h_i \in H(v_s) \cap H(v_t)} (d_{is} + d_{it}) \quad (1)$$

Computing the smallest hub labeling while ensuring the coverage property is NP-hard (Cohen et al. 2003). Therefore, heuristics are often used to compute hub labeling.

Computing Shortest Distance. The shortest distance between v_s and v_t can be calculated by Eq. (1) which only requires a simple scan over the sorted label sets $H(v_s)$ and $H(v_t)$. The complexity is $O(|H(v_s)| + |H(v_t)|)$, where $|H(v_j)|$ denotes the number of labels in $H(v_j)$.

Example 1. *Table 1 shows hub labels for the graph in Figure 1. To compute $d(E, G)$, the hub labels of E and G are scanned and two common hub vertices A and B are found. Here, $d(E, A) + d(A, G) = 5.1 + 2 = 7.1$ and $d(E, B) + d(B, G) = 3.5 + 2.8 = 6.3$. Thus, $d(E, G) = 6.3$.*

Computing Shortest Path. For a hub label $(h_i, d_{ij}) \in H(v_j)$, we use s_{ij} to denote its successor – the first vertex after v_j on the shortest path from v_j to h_i. During the hub labeling computation, the successor nodes are also stored in the labels, i.e., (h_i, d_{ij}, s_{ij}). To compute the path from v_s to v_t, first the shortest distance is computed using Eq. (1) and then the successor node s_{is} in the label (h_i, d_{is}, s_{is}) is used to retrieve the first vertex on the shortest path. Then, the path from s_{is} to h_i is recursively retrieved using the hub labels of s_{is} and h_i until h_i is reached. The path between v_t and h_i is retrieved similarly.

Polyanya (Cui, Harabor, and Grastien 2017): Polyanya is the state-of-the-art online algorithm for ESPP requiring minimal preprocessing time and memory. It employs a search similar to A* search on the navigation mesh which divides the traversable area in the plane into a set of convex traversable polygons. Each search node n is a tuple $n = (I, r)$ where r is the *root* of the search node and I is a contiguous interval on an edge of the navigation mesh such that I is completely visible from r. Polyanya prioritises the search using f-values of nodes where $f(n) = g(n) + h(n)$. Here $g(n)$ is the shortest distance from s to the root r of the node n and $h(n)$ is a lower-bound distance from r to the target t passing through the interval I. When a search node $n = (I, r)$ is expanded, Polyanya generates its successors $n' = (I', r')$ by "pushing" the interval I away from r across the face of adjacent traversable polygon in the navigation mesh. Polyanya terminates when the target is expanded or the open list is empty.

End Point Search (EPS) (Shen et al. 2022): EPS is the state-of-the-art ESPP algorithm. It requires computing a *Compressed Path Database (CPD)* (Botea 2011) during the offline preprocessing. Assume a $V \times V$ table which records the first move (i.e., the first vertex on the shortest path) from every $v_i \in V$ to every $v_j \in V$ where V is the set of all convex vertices in the plane. A CPD reduces the size of this table by compressing each of its row using run-length encoding (Strasser, Harabor, and Botea 2014). Given this CPD, the first move from v_i to v_j can be extracted in $O(\log |r|)$ where $|r|$ is the size of the compressed row of v_i. The shortest path from v_i to v_j is computed by recursive first-move extractions using the CPD until v_j is reached. The cost of computing the shortest path/distance using the CPD is $O(P \log |r|)$ where P is the number of edges on the shortest path.

EPS adapts Polyanya to incrementally obtain convex vertices visible from s and t denoted as V_s and V_t, respectively. If s and t are not co-visible, the shortest distance between s and t can be computed as follows.

$$d(s, t) = \min_{(v_s, v_t) \in V_s \times V_t} Edist(s, v_s) + d(v_s, v_t) + Edist(v_t, t) \quad (2)$$

Here, $d(v_s, v_t)$ is computed using the CPD. Once the shortest distance is computed, the shortest path can be easily retrieved using the CPD. Eq. (2) requires exploring $|V_s| \times |V_t|$ paths where $|V_s|$ and $|V_t|$ are the number of vertices visible from s and t, respectively. Thus, the worst-case cost for EPS is $O(|V_s| \times |V_t| \times P \log |r|)$. EPS employs various effective pruning strategies and optimisations to improve the performance. Despite this, the total number of first-move extractions still remains considerably high, e.g., for queries on the widely-used DAO benchmark, on average, V_s and V_t contain around 16 vertices each and EPS requires over 400 CPD first-move extractions (see Table 2 in Appendix provided in the supplementary files). Unless clear by context, we call this approach EPS-CPD hereafter.

EPS-HL. Our experimental study also considers EPS-HL, a variant of EPS-CPD which uses hub labeling (HL) instead of a CPD. Specifically, hub labels are constructed on the convex vertices V, and $d(v_s, v_t)$ in Eq. (2) is computed using these labels instead of the CPD. Our implementation of EPS-HL employs all possible pruning rules and optimisations that are used by EPS-CPD. We show in experiments that EPS-HL is slower than EPS-CPD indicating that a straightforward application of hub labeling is not helpful.

Our Solution: Euclidean Hub Labeling (EHL)

Our solution consists of two phases: an offline preprocessing; and an online query processing algorithm. In this section, we present the basic ideas and, in the next section, we discuss optimisations to improve the performance.

Offline Preprocessing

The preprocessing phase consists of five steps detailed shortly. First, we use existing techniques to compute a visibility graph and hub labeling on this graph (steps 1-2). Then, we use these labels to compute Euclidean hub labeling for the Euclidean plane (steps 3-5).

1) A visibility graph $G = (V, E)$ is constructed where V is the set of convex vertices and E consists of edges between each pair of co-visible vertices. This is done by running a Polyanya-like depth-first search for each $v \in V$ as explained in (Shen et al. 2020). Figure 1 shows the visibility graph for the convex vertices (A to G) of the map shown in Figure 2.

2) Hub labels are computed on this visibility graph G. Although any hub labeling approach can be used, we use SHP (Li et al. 2017) which is among the state-of-the-art hub labeling approaches for road networks. Table 1 shows hub labeling for the visibility graph of the map in Figure 2. In our implementation, we also store the successor node for each hub label for efficient shortest path retrieval once the distance has been computed. However, for the ease of illustration, we only show (h_i, d_{ij}) for each label in $H(v_j)$.

3) We superimpose a uniform grid covering the whole map. The size of the grid is a parameter and we evaluate its effect in our experiments. For each grid cell c, we store a list L_c, called its visibility list, which consists of every convex vertex v such that at least some part of c is visible from v. For example, for the two cells c_s and c_t shown in Figure 2, $L_{c_s} = \{A, F, G\}$ and $L_{c_t} = \{B, C, D, E\}$. In the next section, we explain how to efficiently compute the visibility lists using Polyanya-like search for each $v \in V$.

4) For each cell c, we create its labels (called via labels) using its visibility list L_c. Specifically, for each vertex $v_j \in L_c$ and for each hub label $(h_i, d_{ij}) \in H(v_j)$, we insert a via label $h_i{:}(v_j, d_{ij})$ in $H(c)$. Intuitively, this label indicates that the cell c is visible from a via vertex v_j and there is a potential path from c to the hub node h_i via v_j and the distance between h_i and v_j is d_{ij}. Consider the vertex G and its hub label $(B, 2.8) \in H(G)$ (see Table 1). Since c_s is visible from G, we insert a label $B{:}(G, 2.8)$ in $H(c_s)$ indicating that there is a path from c_s to B via G and $d(G, B) = 2.8$. Since the visibility list of c_s is $L_{c_s} = \{A, F, G\}$, the labels of A, F and G from Table 1 are used to insert the following via labels in c_s: $A{:}(A, 0)$, $A{:}(F, 2.2)$, $B{:}(F, 3.7)$, $F{:}(F, 0)$, $A{:}(G, 2)$, $B{:}(G, 2.8)$, $F{:}(G, 1.9)$, and $G{:}(G, 0)$.

5) Hub labels of a cell c may have several via labels containing the same hub node, e.g., as noted above, the via labels for c_s contain three labels with A as the hub node. We use $VL_{h_i}(c)$ to denote the set of all via labels in c that have the same hub node h_i. We use $H(c)$ to denote the set of unique hub nodes for labels stored in c. We sort $H(c)$ according to hub nodes h_i to efficiently join the hub labels of two different cells. Table 2 shows hub nodes and via labels for both c_s and c_t, e.g., the hub nodes for c_s are $H(c_s) = \{A, B, F, G\}$ and the via labels for these hub nodes are shown in the corresponding rows.

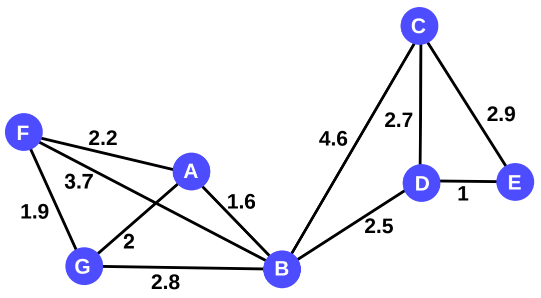

Figure 1: Visibility graph for the example in Figure 2

Vertex	Hub labels
A	(A, 0)
B	(A, 1.6), (B, 0)
C	(A, 6.2), (B, 4.6), (C, 0)
D	(A, 4.1), (B, 2.5), (C, 2.7), (D, 0)
E	(A, 5.1), (B, 3.5), (C, 2.9), (D, 1), (E, 0)
F	(A, 2.2), (B, 3.7), (F, 0)
G	(A, 2), (B, 2.8), (F, 1.9), (G,0)

Table 1: Hub labeling for the graph in Figure 1

Most of the preprocessing steps can be easily parallelized. Specifically, each of step 1 and step 3 can be parallelized because the Polyanya-like searches for each $v \in V$ to construct the visibility graph and visibility lists are independent. We can also parallelize the labels computation for each cell c in steps 4 and 5 because constructing the labels only requires the hub labels computed at step 2 and the visibility list L_c.

Online Query Processing

We define *via-distance* $vdist(p, v_j, h_i)$ as the length of the shortest path between a point p and h_i passing through a convex vertex v_j visible from p. Given a via label $h_i{:}(v_j, d_{ij}) \in VL_{h_i}(c)$ and a point $p \in c$, if v_j is visible from p then $vdist(p, v_j, h_i) = Edist(p, v_j) + d_{ij}$. If v_j is not visible from p, we assume $vdist(p, v_j, h_i) = \infty$. Given a point $p \in c$ and the via labels $VL_{h_i}(c)$, we define the minimum via-distance $vdist_{min}(p, h_i)$ between p and h_i as:

$$vdist_{min}(p, h_i) = \min_{h_i:(v_j, d_{ij}) \in VL_{h_i}(c)} vdist(p, v_j, h_i) \quad (3)$$

Example 2. *Consider the via labels of A in $H(c_s)$ shown in Table 2. $vdist(s, F, A) = EDist(s, F) + d(F, A) = 2.6 + 2.2 = 4.8$ and $vdist(s, G, A) = EDist(s, G) + d(G, A) = 1 + 2 = 3$. Since A is not visible from s, $vdist(s, A, A) = \infty$. Therefore, $vdist_{min}(s, A) = vdist(s, G, A) = 3$.*

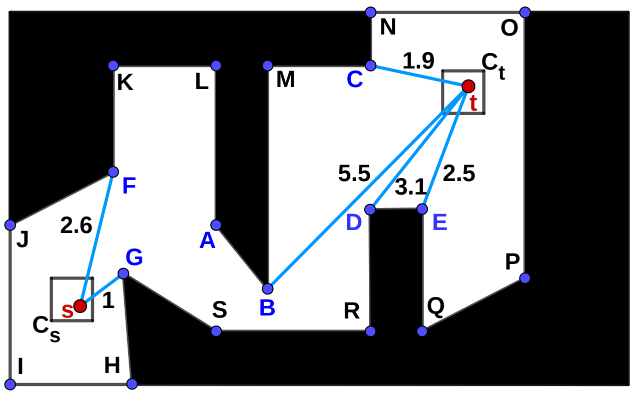

Figure 2: Euclidean plane with polygonal obstacles

$H(c_s)$	Via Labels $VL_{h_i}(c_s)$
A:	(A, 0), (F, 2.2), (G, 2)
B:	(F, 3.7), (G, 2.8)
F:	(F, 0), (G, 1.9)
G:	(G, 0)

$H(c_t)$	Via Labels $VL_{h_i}(c_t)$
A:	(B, 1.6), (C, 6.2), (D, 4.1), (E, 5.1)
B:	(B, 0), (C, 4.6), (D, 2.5), (E, 3.5)
C:	(C, 0), (D, 2.7), (E, 2.9)
D:	(D, 0), (E, 1)
E:	(E, 0)

Table 2: Via labels for c_s and c_t shown in Figure 2

Similarly, it can be confirmed that $vdist_{min}(t, A) = vdist(t, B, A) = 7.1$, $vdist_{min}(s, B) = vdist(s, G, B) = 3.8$ *and* $vdist_{min}(t, B) = vdist(t, B, B) = 5.5$.

Algorithm 1 shows the details of how to compute $vdist_{min}(p, h_i)$. Note that we check whether v_j and p are co-visible only if $Edist(p, v_j) + d_{ij}$ is smaller than the current $vdist_{min}$ (because checking visibility is typically more expensive). We explain how to efficiently check whether v_j is visible from p or not in the next section when we introduce optimisations. Next, we explain how to compute $d(s, t)$.

Let c_s and c_t be the grid cells containing s and t, respectively. If s and t are visible to each other, $d(s, t) = Edist(s, t)$. Otherwise, $d(s, t)$ can be computed as follows.

$$d(s,t) = \min_{h_i \in H(c_s) \cap H(c_t)} vdist_{min}(s, h_i) + vdist_{min}(t, h_i) \quad (4)$$

Algorithm 2 shows the details of our query processing algorithm. It initializes $dist$, which corresponds to the shortest distance found so far, to infinity. It returns $d(s, t) = Edist(s, t)$ if s and t are co-visible. Co-visibility can be efficiently checked by shooting a ray from s to t which we implement on top of a navigation mesh (Kallmann and Kapadia 2014). Specifically, the ray starts from the polygon in the mesh containing s and iteratively traverses the adjacent

Algorithm 1: Computing minimum via-distance

```
Input: a point $p$; $VL_{h_i}(c)$ where $c$ is the cell containing $p$
1 $vdist_{min} = \infty$;
2 for each $h_i{:}(v_j, d_{ij}) \in VL_{h_i}(c)$ do
3     if $Edist(p, v_j) + d_{ij} < vdist_{min}$ then
4         if $v_j$ is visible from $p$ then
5             $vdist_{min} = Edist(p, v_j) + d_{ij}$
6 return $vdist_{min}$;
```

Algorithm 2: Shortest Distance $d(s, t)$ Computation

```
1 $dist = \infty$;
2 if $s$ and $t$ are co-visible then
3     return $Edist(s, t)$
4 $c_s \leftarrow$ the cell that contains $s$; $c_t \leftarrow$ the cell that contains $t$
5 incrementally scan $H(c_s)$ and $H(c_t)$ to find all common
    hub nodes $H(c_s) \cap H(c_t)$
6 for each $h_i \in H(c_s) \cap H(c_t)$ do
7     if $vdist_{min}(s, h_i) + vdist_{min}(t, h_i) < dist$ then
8         $dist = vdist_{min}(s, h_i) + vdist_{min}(t, h_i)$
9 return $dist$
```

traversable polygons intersecting the ray until the ray either enters an obstacle (i.e., t is not visible) or it hits t (i.e., t is visible). If s and t are not co-visible, $d(s, t)$ is computed using Eq. (4) by using the common hub nodes in $H(c_s)$ and $H(c_t)$ which can be found similar to the merge phase of the sort-merge join. Once $d(s, t)$ is found, the shortest path $sp(s, t)$ can be retrieved using the successor nodes as described earlier in Preliminaries.

Example 3. *In our running example,* $H(c_s)$ *and* $H(c_t)$ *have two common hub nodes* A *and* B *(see Table 2). When the common hub node* A *is found, dist is updated to be* $dist = vdist_{min}(s, A) + vdist_{min}(t, A) = 3 + 7.1 = 10.1$. *When the common hub node* B *is accessed, dist is updated to be* $dist = vdist_{min}(s, B) + vdist_{min}(t, B) = 3.8 + 5.5 = 9.3$. *The algorithm returns* $d(s, t) = 9.3$.

Theorem 1. *Algorithm 2 returns the shortest distance* $d(s, t)$.

Proof. If s and t are co-visible, our algorithm correctly returns $d(s, t) = Edist(s, t)$. Otherwise, let $sp(s, t)$ be $\langle s, v_s, \cdots, v_t, t\rangle$ where v_s and v_t are convex vertices visible from s and t, respectively. The shortest distance is then $Edist(s, v_s) + d(v_s, v_t) + Edist(v_t, t)$. Note that v_s and v_t may correspond to the same vertex, i.e., $d(v_s, v_t) = 0$. Hub labeling guarantees that the labels of v_s and v_t contain at least one common hub node h_i that is on the shortest path between v_s and v_t, i.e., (h_i, d_{is}) and (h_i, d_{it}). Thus, $d(v_s, v_t)$ can be computed using the hub labels of v_s and v_t, i.e., $d(v_s, v_t) = d(v_s, h_i) + d(v_t, h_i) = d_{is} + d_{it}$. Since at least some part of the cells c_s and c_t are visible from v_s and v_t, respectively, the via labels $h_i{:}(v_s, d_{is})$ and $h_i{:}(v_t, d_{it})$ are inserted for c_s and c_t, respectively, during the preprocessing. During query processing, these labels with common hub node h_i are found and $dist$ is updated to be $Edist(s, v_s) + d_{is} + Edist(t, v_s) + d_{it}$. Since $d_{is} + d_{it} = d(v_s, v_t)$, we have

$dist = Edist(s, v_s) + d(v_s, v_t) + Edist(v_t, t) = d(s, t)$. Note that the proof holds when v_s and v_t correspond to the same vertex denoted as $v_{s/t}$ for the rest of the proof. This is because each vertex has a label containing itself as a hub (i.e., $v_{s/t}$ has $(v_{s/t}, 0)$) which is inserted to the via labels of both c_s and c_t and the shortest distance $d(s,t) = Edist(s, v_{s/t}) + 0 + Edist(v_{s/t}, t)$ is returned. □

Optimisations

Improving Preprocessing

The cost of our algorithm is directly proportional to the number of hub nodes and via labels for the cells c_s and c_t. Now, we present several pruning rules (PR) to significantly reduce the number of labels stored for each grid cell. This does not only reduce the query processing cost but also reduces the storage and construction cost for our hub labeling approach.

PR1: Pruning using non-taut regions: First, we define non-taut region (Oh and Leong 2017) of a convex vertex using the example of vertex E in Figure 3. Two incident obstacle edges of E are $\overline{DE}$ and $\overline{QE}$. Assume we shoot a ray from D to E that hits an obstacle at X. Similarly, the ray shot from Q to E hits the boundary of the map at a point Y. The region of the map that is within the angle $\angle YEX$ from E is called the non-taut region of E (see the gray area). The taut region of E is the area which is visible from E and is not the non-taut region (see the green area). Note that E cannot be an intermediate vertex on any shortest path from a point p in its non-taut region to any other point p' (where $p' \neq E$) because any path from such p to p' via E will be non-taut.

Based on the idea above, for a cell c, we include a vertex v_j in its visibility list L_c only if c overlaps with its taut region. In other words, we prune every via label h_i:(v_j, d_{ij}) if c does not overlap with the taut region of v_j. Table 3 shows the via labels for c_s and c_t copied from Table 2. Note that c_t does not overlap with the taut regions of both E and C. Thus, all via labels in $H(c_t)$ with E and C as via nodes are pruned (see the blue struckthrough labels). Similarly, c_s does not overlap with the taut region of A, therefore A:$(A, 0)$ is removed from $H(c_s)$.

To efficiently compute the taut region of a vertex v, we modify Polyanya (Cui, Harabor, and Grastien 2017) such that it only generates visible successors and trims each successor by removing the part that lies in non-taut angle range. When Polyanya discovers a successor ab on an edge of the obstacle or boundary, a triangle $\triangle abv$ is stored. When Polyanya terminates, the union of all such triangles represents the taut region of v (see the triangulated green region in Figure 3). For each cell c, we check whether c overlaps with the taut region and insert v in L_c only if it does.

PR2: Pruning non-taut labels: Consider a via label h_i:(v_j, d_{ij}). Let the successor node of v_j (i.e., node after v_j) on the shortest path from v_j to the hub node h_i be denoted as s_{ij}. Consider a point p which is visible from v_j. The shortest path between p and h_i cannot pass through v_j if the subpath $p \to v_j \to s_{ij}$ is non-taut. Thus, the via label h_i:(v_j, d_{ij}) can be pruned from the label set of a cell c if $p \to v_j \to s_{ij}$ is non-taut for every $p \in c$.

For example, consider the label A:$(D, 4.1)$ for $H(c_t)$ in

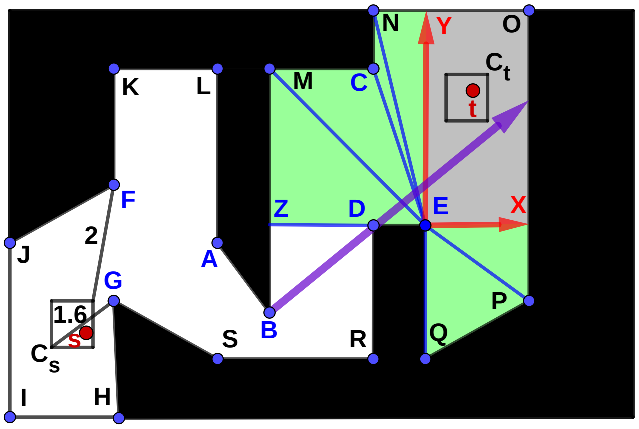

Figure 3: Illustrating pruning rules

$H(c_s)$	Via Labels $VL_{h_i}(c_s)$
~~A~~:	~~(A, 0)~~, ~~(F, 2.2)~~, ~~(G, 2)~~
B:	~~(F, 3.7)~~, (G, 2.8)
F:	(F, 0), ~~(G, 1.9)~~
G:	(G, 0)

$H(c_t)$	Via Labels $VL_{h_i}(c_t)$
A:	(B, 1.6), ~~(C, 6.2)~~, ~~(D, 4.1)~~, ~~(E, 5.1)~~
B:	(B, 0), ~~(C, 4.6)~~, ~~(D, 2.5)~~, ~~(E, 3.5)~~
~~C~~:	~~(C, 0)~~, ~~(D, 2.7)~~, ~~(E, 2.9)~~
D:	(D, 0), ~~(E, 1)~~
~~E~~:	~~(E, 0)~~

Table 3: Labels pruned by different pruning rules (PR) are shown in different colors. PR1: blue, PR2: orange, PR3: red, PR4: purple. The hub nodes with all labels pruned are struckthrough gray.

Table 3. The successor node on the shortest path from D to A is B (see Figure 3). Since $t \to D \to B$ is non-taut, D cannot be on the shortest path from t to A. We shoot a ray from the successor node B to D (see the purple ray) and since the whole cell c_t lies above this ray, D cannot be on the shortest path from any $p \in c_t$ to A. Thus, the label A:$(D, 4.1)$ can be safely pruned from $H(c_t)$. Similarly, B:$(D, 2.5)$ and C:$(D, 2.7)$ can be pruned from $H(c_t)$ and F:$(G, 1.9)$ can be pruned from $H(c_s)$ (see the orange struckthrough labels).

PR3: Pruning using distance bounds: Let $minEdist(c, v)$ and $maxEdist(c, v)$ denote the minimum and maximum Euclidean distance, respectively, from a vertex v to any point of a cell c, e.g., $minEdist(c, v) = \min_{p \in c} Edist(p, v)$.

Lemma 1. *Consider a cell c and its two via labels for the same hub node: h_i:(v_j, d_{ij}) and h_i:(v_k, d_{ik}). If the whole cell c is visible from v_j and $maxEdist(c, v_j) + d_{ij} \leq minEdist(c, v_k) + d_{ik}$ then $vdist(p, v_j, h_i) \leq vdist(p, v_k, h_i)$ for every $p \in c$.*

Proof. Since the whole c is visible from v_j, for every $p \in c$, $vdist(p, v_j, h_i) = Edist(p, v_j) + d_{ij} \leq maxEdist(c, v_j) +$

d_{ij}. Since $minEdist(c, v_k) + d_{ik} \leq vdist(p, v_k, h_i)$ and $maxEdist(c, v_j) + d_{ij} \leq minEdist(c, v_k) + d_{ik}$, we have $vdist(p, v_j, h_i) \leq vdist(p, v_k, h_i)$. □

Using the above lemma, the label $h_i{:}(v_k, d_{ik})$ can be pruned because this label is not needed to compute $vdist_{min}(p, h_i)$ for any $p \in c$. Consider c_s in Figure 3 and the labels $A{:}(F, 2.2)$ and $A{:}(G, 2)$ of $H(c_s)$ in Table 3. Here, $maxEdist(c_s, G) + d(G, A) = 1.6 + 2 = 3.6$ is smaller than $minEdist(c_s, F) + d(F, A) = 2 + 2.2 = 4.2$. Thus, the label $A{:}(F, 2.2)$ can be pruned from $H(c_s)$ (see the red struckthrough label in Table 3).

PR4: Pruning dead-end labels: We say that an edge $\overline{uv}$ between two vertices u and v is a dead-end edge if either u is in the non-taut region of v or v is in the non-taut region of u. It is called a dead-end edge because it can only be the first or the last edge on any shortest path, i.e., this edge cannot be an *intermediate* edge for any shortest path. For example, in Figure 3, B is in the non-taut region of F and the edge $\overline{FB}$ is a non-taut edge. Note that $\overline{FB}$ cannot be an intermediate edge for any shortest path.

A label $h_i{:}(v_j, d_{ij})$ is called a dead-end label if the first or the last edge on the shortest path between h_i and v_j is a dead-end edge. For example, in Table 3, $B{:}(F, 3.7)$ is a dead-end label because the only edge $\overline{FB}$ on the shortest path is a dead-end edge. Note that this label is not needed to compute the shortest path between any s and t and thus can be removed. This holds true even when s and/or t are on v_j or h_i (which can be proved using arguments similar to those presented in (Oh and Leong 2017)). When we construct the hub labels of the visibility graph at Step 2 (see Offline Preprocessing), we maintain the first and last edges on the shortest paths and remove the labels for which the first or the last edge is a dead-end edge. In our running example, the labels $E{:}(C, 2.9)$, $F{:}(B, 3.7)$ and $G{:}(A, 2)$ are removed from Table 1. Consequently, these labels are not included in the visibility list of any cell c, i.e., these labels are pruned for all cells of the grid.

Storing minimum distances to hub nodes: We define a lower bound on the minimum distance from a cell c to a hub node h_i using the via labels $VL_{h_i}(c)$ as follows.

$$mindist(c, h_i) = \min_{h_i:(v_j, d_{ij}) \in VL_{h_i}(c)} minEdist(c, v_j) + d_{ij} \tag{5}$$

During preprocessing, for each unique hub node $h_i \in H(c)$, we also store $mindist(c, h_i)$ computed using Eq. (5). Later, we show how to utilize this during query processing.

Query Processing

The pruning rules presented above significantly reduce the number of labels stored for each cell (e.g., see Table 3) thus reduce the query processing cost. Recall that, to compute $vdist_{min}(p, h_i)$, Algorithm 1 ignores a label $h_i{:}(v_j, d_{ij})$ if v_j is not visible from p. As stated earlier in the pruning rule 1 (PR1), v_j cannot be on the shortest path if p is in its non-taut region. Thus, instead of checking whether p is visible from v_j or not, we check whether p is in the taut region of v_j or not. We efficiently check this as follows. During the preprocessing, we maintain a triangle list for each convex vertex v_j which represents the taut region of v_j (see the triangulated green region in Figure 3). The triangle list is kept sorted according to the angles. To check whether a point p is in taut region of v_j, a binary search is conducted to find the triangle $\triangle abv_j$ whose angle range overlaps with the angle of $\overline{pv_j}$. p is in the taut region if and only if such a triangle exists and p is found to be inside the triangle. Let T be the number of triangles in the triangle list. The complexity of checking whether p is in the taut region or not is $O(\log T)$.

	#Maps	# Queries	#Vertices	#Convex Vertices
DAO	156	159,464	1727.6	926.5
DA	67	68,150	1182.9	610.8
BG	75	93,160	1294.4	667.7
SC	75	198,224	11487.5	5792.7

Table 4: Total number of maps and queries, and average number of vertices and convex vertices in each benchmark.

Optimisation: The optimisation is based on the lower bound (see Eq. (5)) we stored during preprocessing.

Lemma 2. $\forall p \in c$, $mindist(c, h_i) \leq vdist_{min}(p, h_i)$.

Proof. For every point $p \in c$, $minEdist(c, v_j) \leq d(p, v_j)$ for any vertex v_j. Thus, $minEdist(c, v_j) + d_{ij} \leq d(p, v_j) + d_{ij}$ which implies $mindist(c, h_i) \leq vdist_{min}(p, h_i)$. □

Lemma 3. *For a source $s \in c_s$ and a target $t \in c_t$ and a common hub node h_i, $mindist(c_s, h_i) + mindist(c_t, h_i) \leq vdist_{min}(s, h_i) + vdist_{min}(t, h_i)$.*

In Algorithm 2, we sort the common hub nodes $H(c_s) \cap H(c_t)$ in ascending order of $mindist(c_s, h_i) + mindist(c_t, h_i)$ and iteratively process them in this order (at line 6). When the algorithm accesses a common hub node h_i for which the currently found shortest distance $dist$ is less than or equal to $mindist(c_s, h_i) + mindist(c_t, h_i)$, the algorithm terminates returning $dist$ as the shortest distance. This is because $d(s, h_i, t)$ for every common hub node h_i not yet processed by the algorithm is guaranteed to be at least equal to the shortest distance $dist$ found already by the algorithm.

Complexity: Let $|S|$ denote the number of elements in a set S. Algorithm 1 (computing $vdist_{min}(p, h_i)$) requires $O(L \log T)$ where $L = |VL_{h_i}(c)|$ is the number of via labels for the hub node h_i in the cell c that contains p and $O(\log T)$ is the cost to check whether p is in the taut region of a vertex v_j or not. Here, L is bounded by the number of convex vertices from which c is visible[1]. To compute the shortest distance, Algorithm 2 takes $O(|H(c_s)| + |H(c_t)|)$ to find all common hub nodes. For each common hub node h_i, the algorithm computes $vdist_{min}(s, h_i)$ and $vdist_{min}(t, h_i)$ using Algorithm 1. Therefore, the total cost of the algorithm is $O(|H(c_s)| + |H(c_t)| + |H(c_s) \cap H(c_t)| \times L \log T)$.

[1]In practice, L is typically a small value especially when grid cells are small (see Table 7). Also, $O(\log T)$ cost is only required for the labels for which the if condition at line 3 in Algorithm 1 is true

Map	Build Time (Secs)												Memory (MB)											
	EHL								Competitors				EHL								Competitors			
	1x		4x		16x		64x		EPS-CPD		EPS-HL		1x		4x		16x		64x		EPS-CPD		EPS-HL	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
DAO	4.41	87.8	0.35	7.1	0.05	0.72	0.02	0.24	0.11	1.86	0.05	0.72	137	2596	27.5	559.3	7.1	145	2.4	37.2	0.21	3.48	0.78	9.36
DA	1.52	7.45	0.13	0.6	0.02	0.08	0.01	0.07	0.02	0.15	0.02	0.21	48.9	203	9.0	41.1	2.2	10.7	0.9	4.0	0.06	0.26	0.04	0.15
BG	6.99	31.8	0.51	2.4	0.06	0.45	0.02	0.20	0.04	0.81	0.03	0.30	197	1144	31.7	242.8	7.7	64.0	2.2	19.6	0.12	1.38	0.07	0.74
SC	128	603	8.47	42	0.85	4.11	0.25	1.18	3.49	51.5	0.71	3.68	2644	13430	523	2749	142	742	41	204	2.30	13.3	9.30	47.4

Table 5: Average cost (build time and memory) per map and maximum cost among all maps for each of the four benchmarks.

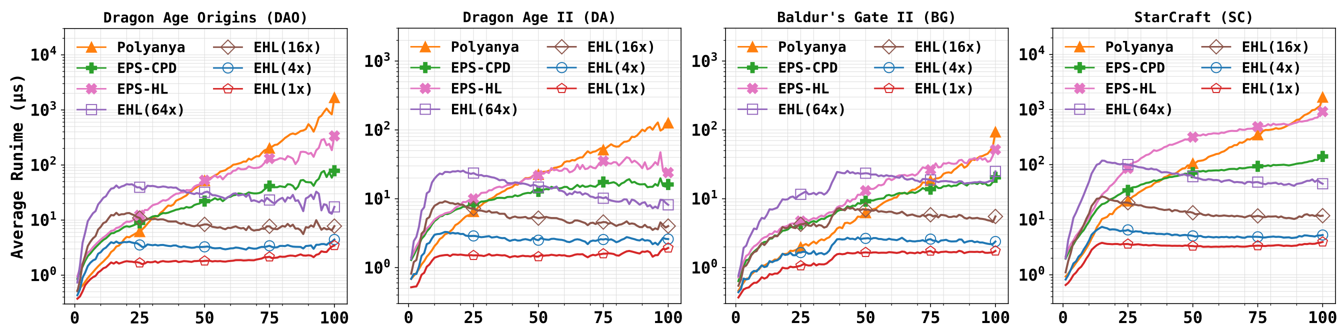

Figure 4: Runtime comparison between EHL and competitors. The x-axis shows the percentile ranks of queries in number of node expansions needed by A* search to solve them.

Experiments

Similar to the existing studies, we conduct experiments on the widely used game map benchmarks[2], described in (Sturtevant 2012a), on a total of 373 game maps (see Table 4). Each map has an underlying grid and we refer to the cell of this grid as the base cell. We evaluate the effect of size of the uniform grid used in our Euclidean Hub Labeling (EHL) approach by using grids where each cell is N times bigger than the base cell in each dimension. We vary N from 1 to 64 (e.g., cell sizes vary from 1x to 64x of the base cell size in each dimension) and employ SHP (Li et al. 2017) as the underlying hub labeling approach using the implementation taken from the public repository[3]. For reproducibility, implementation of EHL is available online[4].

We compare our approach with two state-of-the-art algorithms, EPS-CPD[5] (Shen et al. 2022) and Polyanya[6] (Cui, Harabor, and Grastien 2017), using the implementations provided by the authors. We also compare with EPS-HL (a variant of EPS-CPD as introduced in the Related Work section). An advantage of using grid benchmarks is that it enables a direct comparison with any-angle grid-based pathfinding algorithms. Note that it was shown in the previous works (Cui, Harabor, and Grastien 2017; Shen et al. 2020) that Polyanya and EPS-CPD significantly outperform all existing any-angle grid-based pathfinding algorithms. Each individual query is run 5 times and we report the average query time for each algorithm to return the complete shortest path. All the algorithms are implemented in C++ and complied with -O3 flag. We run experiments on a 3.2 GHz Intel Core i7 machine with 32 GB of RAM.

Preprocessing Time and Space: Table 5 compares the preprocessing time and memory of EHL for different grid sizes with EPS-CPD and EPS-HL (we do not compare with Polyanya because it is an online algorithm and employs a navigation mesh requiring insignificant preprocessing time and storage). All experiments are run on a 12 core machine and the build time would be better/worse if more/less processors are available. Also, the reported cost for EHL is the total cost including the cost for constructing and storing SHP hub labels and visible/taut regions for each vertex. The build time and memory required by EHL are in general larger than EPS-CPD and EPS-HL especially for smaller grid cells. However, these are practical even when the cell size is 1x especially considering the main memory available in modern computers. Also, the build time and memory of EHL reduce significantly as the cell sizes increase, e.g., as the cell size increases from 1x to 64x, build time and memory requirements decrease by over 100 times and over 50 times, respectively. This is because EHL requires processing fewer cells when cells are larger. However, this comes at the expense of querying cost as shown next.

Query Performance: Similar to previous works (Cui, Harabor, and Grastien 2017; Shen et al. 2022), in Figure 4, we sort the queries by the number of nodes expansions required by the standard A* search to solve them (which is a proxy for how challenging a query is) and the x-axis corre-

[2]https://github.com/nathansttt/hog2
[3]http://degroup.cis.umac.mo/sspexp
[4]https://github.com/goldi1027/EHL
[5]https://github.com/bshen95/End-Point-Search
[6]https://bitbucket.org/mlcui1/polyanya

Map	C	Build Time (Secs)							Memory (MB)						
		-PR1	-PR2	-PR3	-PR4	-OP	-All	Final	-PR1	-PR2	-PR3	-PR4	-OP	-All	Final
DAO	1x	13.282	4.476	4.440	5.391	4.430	-	4.418	182.049	203.399	247.715	206.159	113.766	-	136.139
	4x	0.968	0.355	0.351	0.434	0.349	0.999	0.352	36.318	30.289	38.980	47.864	24.470	176.850	26.584
	16x	0.143	0.055	0.050	0.060	0.049	0.104	0.051	8.176	6.222	6.547	12.140	5.930	19.934	6.162
	64x	0.075	0.025	0.022	0.021	0.021	0.029	0.022	1.715	1.444	1.444	2.928	1.406	3.683	1.444
DA	1x	4.552	1.442	1.442	1.651	1.398	4.879	1.873	64.846	69.267	69.509	70.219	39.684	493.856	48.521
	4x	0.389	0.132	0.127	0.146	0.123	0.337	0.196	11.410	9.636	10.726	14.786	7.779	40.601	8.608
	16x	0.089	0.030	0.024	0.025	0.021	0.051	0.024	2.381	1.862	1.884	3.456	1.757	4.928	1.854
	64x	0.062	0.017	0.010	0.009	0.009	0.020	0.012	0.550	0.478	0.478	0.913	0.459	1.090	0.474
BG	1x	21.851	6.490	6.689	7.731	6.666	-	6.988	269.217	332.058	298.593	266.981	157.088	-	196.831
	4x	1.421	0.498	0.496	0.578	0.485	1.676	0.597	40.575	39.932	44.113	50.017	27.633	240.559	31.065
	16x	0.126	0.059	0.055	0.064	0.054	0.167	0.059	8.894	7.325	7.874	12.584	6.682	23.812	7.045
	64x	0.028	0.020	0.018	0.017	0.017	0.055	0.018	1.887	1.595	1.599	2.948	1.542	3.778	1.592
SC	1x	-	-	-	-	123.108	-	128.353	-	-	-	-	2190.222	-	2633.301
	4x	27.278	8.348	8.517	10.580	8.230	-	8.471	717.278	703.762	874.349	927.945	469.553	-	512.359
	16x	2.047	0.841	0.853	0.985	0.833	2.873	0.851	178.001	138.485	162.484	258.009	126.257	601.141	130.858
	64x	0.395	0.253	0.257	0.199	0.252	0.480	0.251	38.463	30.419	30.551	62.555	29.820	86.746	30.394

Table 6: Effect of pruning rules (PR) and the optimisation (OP) on build time and memory of EHL. Final is our final algorithm. -PRx refers to the version where only the pruning rule x is removed from Final, -OP is when only the optimisation is removed and -All is when all pruning rules and the optimisation are removed. We exclude the cost for constructing/storing SHP hub labels and visible regions as these are common to all versions. We show "-" for the cases when we ran out of memory.

sponds to the percentile ranks of queries in this order. Note that the y-axis is in log-scale. EHL(1x) and EHL(4x) significantly outperform Polyanya, EPS-CPD and EPS-HL for almost all cases. Specifically, EHL(1x) typically outperforms Polyanya and EPS-HL by around two orders of magnitude and EPS-CPD by more than an order of magnitude. Although EHL with larger cell sizes (i.e., 64x and 16x) outperforms Polyanya, EPS-CPD and EPS-HL for the challenging queries, for the less challenging queries (when s and t are close but not co-visible), EHL runs slower because it needs to access a large number of hub nodes and labels in order to solve these queries. Querying time for EHL stabilises (or decreases) as the queries become more challenging (in terms of A* node expansions) because the cost of EHL mainly depends on the number of labels stored in c_s and c_t (and not much on $d(s,t)$), and the most challenging queries (in terms of A* node expansions) belong to the large maps without much open space which results in fewer vertices visible from c_s and c_t resulting in smaller number of labels in general.

Ablation Study : In Table 6 and Table 7, we show the effectiveness of the four pruning rules and the optimisation (e.g., by removing these one at a time) on build time, memory and query performance.

Preprocessing Time and Space: Table 6 shows that removing PR1 significantly increases the build time, because without removing the non-taut regions, EHL needs more time to compute the full visible area for each convex vertex which also results in a larger visible list to be processed in preprocessing. In addition, all pruning rules show positive effects on memory (and in some cases on build time). However, including the optimisation (OP) has a negative effect on build time and memory because computing and storing lower-bounds incur additional cost.

Query Performance: Table 7 shows that OP is the most important enhancement for query performance as it stores lower-bounds which helps EHL skip some common hub nodes during the query time. Next in line for effectiveness, especially for bigger grid cells, are PR1 and PR4 which show remarkable improvement in query performance by removing a notable amount of labels that are inside the non-taut region or are dead-end. Though not as significant, PR2 and PR3 also play an important role in terms of improving performance, and together with the other pruning rules and the optimisation, we achieve speed up against the baseline version of EHL (where all PRs and OP are removed) by up to an order of magnitude.

Table 7 also gives insights into why EHL significantly outperforms EPS-CPD especially for smaller grid cells by showing some important statistics from their respective complexity analysis. Although there are many hub nodes in c_s and c_t (i.e., $|H(c_s)|$ and $|H(c_t)|$), the number of common hub nodes $\#CN = |H(c_s) \cap H(c_t)|$ and the average number of via labels per hub node L are quite small for smaller grid cells. Consequently, EHL is very efficient especially for smaller grid cells as it needs to access a small number of via labels to solve a query. On the other hand, although EPS-CPD has a smaller number of vertices visible from s and t (i.e., $|V_s|$ and $|V_t|$), it has to extract many first moves (#FM) from the CPD each requiring a binary search on a compressed row of the CPD. Furthermore, EPS-CPD requires finding the vertices visible from s and t using Polyanya which incurs additional cost.

Remarks on EPS-CPD and EPS-HL: EPS-HL is outperformed by EPS-CPD mainly because, unlike EPS-CPD,

Map	C	Query Performance (μs)							EHL (Final)				EPS-CPD				EPS-HL			
		-PR1	-PR2	-PR3	-PR4	-OP	-All	Final	|H(c_s)|	|H(c_t)|	#CN	L	$\|V_s\|$	$\|V_t\|$	#FM	Time	$\|V_s\|$	$\|V_t\|$	#AL	Time
DAO	1x	2.18	2.17	2.22	2.05	2.38	-	1.97	80	79	10	1.80	17	16	416	27.40	17	16	27023	82.62
	4x	3.83	3.49	3.74	4.00	5.38	17.86	3.49	113	112	17	5.23								
	16x	11.07	8.58	8.80	11.76	13.30	31.30	8.91	160	162	30	13.21								
	64x	37.26	29.36	29.48	45.41	35.06	75.72	30.95	285	288	78	23.01								
DA	1x	1.85	1.72	1.85	1.63	1.75	6.40	1.55	44	42	5	1.72	11	11	161	12.62	11	11	5615	22.65
	4x	3.30	2.61	2.91	2.92	3.36	8.53	2.70	60	58	8	4.76								
	16x	7.78	5.42	5.60	7.09	7.24	13.98	5.69	86	83	14	10.61								
	64x	20.49	14.73	15.06	20.90	16.76	30.56	15.55	157	155	40	15.97								
BG	1x	1.78	1.82	1.83	1.67	1.93	-	1.66	59	56	12	1.51	11	11	132	11.53	11	11	5795	21.92
	4x	2.88	2.68	2.84	2.72	4.03	19.06	2.59	77	73	18	3.99								
	16x	7.39	5.96	6.27	7.41	10.80	30.41	6.46	107	103	30	10.78								
	64x	25.73	19.25	19.40	27.84	27.19	61.96	21.01	176	170	65	20.61								
SC	1x	-	-	-	-	4.61	-	3.45	167	163	22	1.34	32	32	1108	70.99	32	32	132645	359.48
	4x	6.52	6.10	6.67	6.62	10.03	-	5.46	236	232	34	3.17								
	16x	20.02	14.86	16.43	20.61	32.23	132.24	15.05	364	360	62	8.80								
	64x	97.49	64.45	65.09	111.15	105.20	257.94	67.19	638	633	139	18.16								

Table 7: Effect of the pruning rules (PR) and the optimisation (OP) on query performance (see Table 6 for naming of different versions). We only consider the queries for which s and t are not co-visible because PRs and OP are applicable only for such queries (144,011 for DAO, 62,861 for DA, 69,104 for BG, and 179,036 for SC). For EHL, we also report the average # of hub nodes $|H(c_s)|$ (resp. $|H(c_t)|$) in c_s (resp. c_t), the average # of common hub nodes #CN=$|H(c_s)| \cap H(c_t)|$ and the average # of via labels per hub node (L) in c_s and c_t. For EPS-CPD and EPS-HL, we report average runtime (in μs) as well as average $|V_s|$ (resp. $|V_t|$) denoting the average # of vertices visible from s (resp. t). For EPS-CPD, we also show the average # of total first move (#FM) extractions and, for EPS-HL, we report the average # of total accessed labels (#AL) required to answer the query.

EPS-HL is unable to exploit caching and cannot effectively use partial paths for pruning. Although HL can typically compute the shortest distance between a given pair of vertices faster than a CPD, our experimental results (e.g., see Table 7) show that EPS-HL runs significantly slower than EPS-CPD. This is due to the two major reasons:

1. EPS-CPD and EPS-HL both require computing distances between many pairs of vertices in $V_s \times V_t$ and CPD can significantly benefit from caching. Specifically, to compute the distance, CPD iteratively retrieves first moves until the target vertex is reached. EPS-CPD caches the distances on the intermediate nodes (Shen et al. 2022) and, in future iterations, uses these cached values to quickly obtain the distances. On the other hand, EPS-HL is unable to benefit from caching (at least trivially).

2. When a first move is extracted by CPD from a vertex in V_s towards a vertex in V_t (or vice versa), non-taut paths can be pruned if the first move is a non-taut move (see (Shen et al. 2022) for details). Thus, for many pairs, EPS-CPD does not need to recover the whole path if the first move is non-taut. In contrast, EPS-HL requires computing the shortest distance before it can prune the path, and computing the shortest distance between two vertices requires accessing all hub labels stored in the two vertices.

Table 7 shows the average total number of first move extractions (#FM) by EPS-CPD and the average total number of accessed labels (#AL) by EPS-HL to answer a query. Since EPS-CPD employs caching and can prune a non-taut path after only one first move extraction from each end, it requires a much smaller number of first move extractions. On the other hand, EPS-HL needs to access a large number of labels resulting in a significantly higher query cost.

The above explains that the performance improvement of our algorithm, EHL, does not come from simply using HL instead of CPD but because EHL is inherently different and addresses two major limitations of the EPS framework: i) EHL avoids the $|V_s| \times |V_t|$ pairwise complexity of the EPS-CPD and EPS-HL by creating "via labels" for grid cells obtained by projecting the hub labels from the vertices of the graph to the cells visible from them; and ii) during query processing, both EPS-CPD and EPS-HL require finding the vertices visible from s or t by employing Polyanya which incurs additional cost. On the other hand, EHL avoids such computation because the pre-computed hub labels of c_s and c_t contain the vertices visible from these cells.

Conclusions and Future Work

We present a new Euclidean pathfinding approach based on hub labeling, called Euclidean Hub Labeling (EHL), which significantly outperforms the state-of-the-art algorithms in terms of query time. To compute hub labels, we used SHP (Li et al. 2017) which is one of the most efficient distance computation techniques for road networks. It needs to be investigated whether other techniques specifically designed to compute labels for the Euclidean plane can provide further performance improvement. Also, in future, we plan to focus on improving the pre-processing cost of EHL.

Acknowledgements

Muhammad Aamir Cheema is supported by Australian Research Council FT180100140 and DP230100081.

References

Abraham, I.; Delling, D.; Goldberg, A. V.; and Werneck, R. F. 2011. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, 230–241. Springer.

Botea, A. 2011. 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.

Cheema, M. A. 2018. Indoor location-based services: challenges and opportunities. *SIGSPATIAL Special*, 10(2): 10–17.

Cohen, E.; Halperin, E.; Kaplan, H.; and Zwick, U. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.*, 32(5): 1338–1355.

Cui, M.; Harabor, D. D.; and Grastien, A. 2017. 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*, 496–502. ijcai.org.

Harabor, D. D.; Grastien, A.; Öz, D.; and Aksakalli, V. 2016. Optimal Any-Angle Pathfinding In Practice. *Journal of Artificial Intelligence Research*, 56: 89–118.

Kallmann, M.; and Kapadia, M. 2014. Navigation Meshes and Realtime Dynamic Planning for Virtual Worlds. In *ACM SIGGRAPH 2014 Courses*, 3. ACM Press.

Li, Y.; U, L. H.; Yiu, M. L.; and Kou, N. M. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *Proceedings of the VLDB Endowment*, 11(4): 445–457.

Mac, T. T.; Copot, C.; Tran, D. T.; and De Keyser, R. 2016. Heuristic approaches in robot path planning: A survey. *Robotics and Autonomous Systems*, 86: 13–28.

Oh, S.; and Leong, H. W. 2017. 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*, 64–72. AAAI Press.

Shen, B.; Cheema, M. A.; Harabor, D.; and Stuckey, P. J. 2020. Euclidean Pathfinding with Compressed Path Databases. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 4229–4235. ijcai.org.

Shen, B.; Cheema, M. A.; Harabor, D. D.; and Stuckey, P. J. 2022. Fast optimal and bounded suboptimal Euclidean pathfinding. *Artificial Intelligence*, 302: 103624.

Strasser, B.; Harabor, D.; and Botea, A. 2014. 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.

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

Sturtevant, N. R. 2012b. Moving path planning forward. In *International Conference on Motion in Games*, 1–6. Springer.