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, bojie.shen, aamir.cheema}@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 2012a), 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) (Bot]a 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 $c_i$ if $c_i$ 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) if a straight line connecting them does not pass through any obstacle. A path $P$ between a source $s$ and a target $t$ is an ordered set of points $(p_1, p_2, \ldots, p_n)$ such that, for each
$p_i$ ($i < n$), $p_n$ and $p_{n+1}$ are co-visible where $p_1 = s$ and $p_n = t$. The length of a path $P$ is the cumulative Euclidean distance between every successive pair of points, denoted as $|P|$, i.e., $|P| = \sum_{i=1}^{k-1} Edist(p_i, p_{i+1})$ 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 $d(s, t)$. 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 $v_i \in V$, a set of hub labels denoted as $H(v_i)$. Each hub label is a tuple $(h_i, d_{ij}) \in H(v_i)$ 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 $H(v_i)$ 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 $s_{ix}$ in the label $(h_i, d_{ix}, s_{ix})$ is used to retrieve the first vertex on the shortest path. Then, the path from $s_{ix}$ to $h_i$ is recursively retrieved using the hub labels of $s_{ix}$ and $h_i$ until $h_i$ is reached. The path between $v_s$ 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 $v_j$. The shortest path from $v_i$ to $v_j$ is computed by recursive first-move extractions using the CPD until $v_j$ 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 experi-
ments 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 Polyanly-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 \( v \) is visible from \( v \). For example, for the two cells \( c_2 \) and \( c_3 \) shown in Figure 2, \( L_{c_2} = \{A, F, G\} \) and \( L_{c_3} = \{B, C, D, E\} \). In the next section, we explain how to efficiently compute the visibility lists using Polyanly-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 \((h_i, d_{ij}) \in H(c)\). 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 \( c_2 \) is visible from \( G \), we insert a label \((B, (G, 2.8)) \in H(c_2)\) indicating that there is a path from \( c_2 \) to \( B \) via \( G \) and \( d(G, B) = 2.8 \).

Since the visibility list of \( c_2 \) is \( L_{c_2} = \{A, F, G\} \), the labels of \( A, F \) and \( G \) from Table 1 are used to insert the following via labels in \( c_2 \): \( A: (A, 0) \), \( A: (F, 2.2) \), \( B: (F, 3.7) \), \( F: (F, 0) \), \( A: (G, 2) \), \( B: (G, 2.8) \), \( F: (G, 1.9) \), 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 \( c_2 \) contain three labels with \( A \) as the hub node. We use \( V L_{h_i}(c) \) 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 labels and via labels for both \( c_2 \) and \( c_3 \). In the next section, we explain how to efficiently compute the visibility lists using Polyanly-like search for each \( v \in V \).

**Online Query Processing**

We define \( \text{via-distance} \) \( \text{vdist}(p, v_j, h_i) \) 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 \((h_i: (v_j, d_{ij})) \in V L_{h_i}(c) \) and a point \( p \in c \), if \( v_j \) is visible from \( p \) then \( \text{vdist}(p, v_j, h_i) = \text{Edist}(p, v_j) + d_{ij} \). If \( v_j \) is not visible from \( p \), we assume \( \text{vdist}(p, v_j, h_i) = \infty \). Given a point \( p \in c \) and the via labels \( V L_{h_i}(c) \), we define the minimum via-distance \( \text{vdist}_{\min}(p, h_i) \) as:

\[
\text{vdist}_{\min}(p, h_i) = \min_{h_i: (v_j, d_{ij}) \in V L_{h_i}(c)} \text{vdist}(p, v_j, h_i) \tag{3}
\]

**Example 2.** Consider the via labels of \( A \) in \( H(c_2) \) shown in Table 2. \( \text{vdist}(s, F, A) = \text{Edist}(s, F) + d(F, A) = 2.6 + 2.2 = 4.8 \) and \( \text{vdist}(s, G, A) = \text{Edist}(s, G) + d(G, A) = 1 + 2 = 3 \). Since \( A \) is not visible from \( s \), \( \text{vdist}(s, A, A) = \infty \). Therefore, \( \text{vdist}_{\min}(s, A) = \text{vdist}(s, G, A) = 3 \). Similarly, it can be confirmed that \( \text{vdist}_{\min}(t, A) = \ldots \)
Algorithm 1: Computing minimum via-distance

Input: a point \( p \), \( V L_{h_i}(c) \) where \( c \) is the cell containing \( p \)

1. \( vdist_{min} = \infty \);
2. for each \( h_i; (v_j, d_{ij}) \in V L_{h_i}(c) \) 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 \( s \);
5. \( c_t \leftarrow \) the cell that contains \( t \);
6. incrementally scan \( H(c_s) \) and \( H(c_t) \) to find all common hub nodes \( H(c_s) \cap H(c_t) \);
7. for each \( h_i \in H(c_s) \cap H(c_t) \) do
8.    if \( vdist_{min}(s, h_i) + vdist_{min}(t, h_i) < dist \) then
9.        \( dist = vdist_{min}(s, h_i) + vdist_{min}(t, h_i) \)
10. return \( dist \);

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 \( dist = vdist_{min}(s, A) + vdist_{min}(t, A) = 3 + 7 = 10.1 \). 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., \( (h_i; d_{ih}) \) and \( (h_i; d_{ih'}) \). Thus, \( d(v_s, v_t) \) can be computed using the hub labels of \( v_s \) and \( v_t \), i.e., \( d(v_s, v_t) = d(h_s, h_i) + d(h_t, h_i') + d_{ih} + d_{ih'} = d_{is} + d_{it} \). 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 \( Edist(s, v_s) + d_{is} + Edist(t, v_t) + d_{it} \). Since \( d_{is} + d_{it} = d(v_s, v_t) \), we have \( dist = Edist(s, v_s) + d(v_s, v_t) + Edist(v_t, t) = d(s, t) \).

\[d(s, t) = \min_{h_i \in H(c_s) \cap H(c_t)} \text{vdist}_{min}(s, h_i) + \text{vdist}_{min}(t, h_i)\]

(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 Kappadai 2014). Specifically, the ray starts from the polygon in the mesh containing \( s \) and iteratively traverses the adjacent traversable polygons intersecting the ray until it 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.
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 \( DE \) and \( QE \). 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 \( v_j \). Figure 4 shows the via labels for \( c_s \) and \( c_t \) copied from Table 2. Therefore, \( c_t \) does not overlap with the taut region of \( c_s \), therefore \( A: (c_s, 0) \) 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 \( \Delta abv \) 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 \( p \rightarrow v_j \rightarrow s_{ij} \) 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 \( p \rightarrow v_j \rightarrow s_{ij} \) is non-taut for every \( p \in c \).

For example, consider the label \( A:(D, 4.1) \) for \( H(c_t) \) in Figure 4. The successor node on the shortest path from \( D \) to

### Table

<table>
<thead>
<tr>
<th>( H(c_s) )</th>
<th>Via Labels ( V L_{h_i}(c_s) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: ( (A, 0), (F, 2.2), (G, 2) )</td>
<td></td>
</tr>
<tr>
<td>B: ( (F, 3.7), (G, 2.8) )</td>
<td></td>
</tr>
<tr>
<td>C: ( (F, 0), (G, 1.9) )</td>
<td></td>
</tr>
<tr>
<td>D: ( (G, 0) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( H(c_t) )</th>
<th>Via Labels ( V L_{h_i}(c_t) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: ( (B, 1.6), (E, 6.2), (E, 4.1), (E, 5.1) )</td>
<td></td>
</tr>
<tr>
<td>B: ( (B, 0), (E, 4.6), (D, 2.5), (E, 3.5) )</td>
<td></td>
</tr>
<tr>
<td>C: ( (E, 0), (D, 2.7), (E, 2.9) )</td>
<td></td>
</tr>
<tr>
<td>D: ( (D, 0), (E, 1) )</td>
<td></td>
</tr>
<tr>
<td>E: ( (E, 0) )</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: 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.

\( A \) is \( B \) (see Figure 3). Since \( t \rightarrow D \rightarrow B \) 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 \( \\text{vdist}(p, v_j, h_i) \leq \text{vdist}(p, v_k, h_i) \) for every \( p \in c \).

**Proof.** Since the whole \( c \) is visible from \( v_j \), for every \( p \in c \), \( \text{vdist}(p, v_j, h_i) = Edist(p, v_j) + d_{ij} \leq \maxEdist(c, v_j) + d_{ij} \). Since \( \minEdist(c, v_k) + d_{ik} \leq \text{vdist}(p, v_k, h_i) \) and \( \maxEdist(c, v_j) + d_{ij} \leq \minEdist(c, v_k) + d_{ik} \), we have...
\( vdist(p, v_j, h_i) \leq vdist(p, v_k, h_i) \). \qed

Using the above lemma, the label \( h_i;(v_k, d_k) \) can be pruned because this label is not needed to compute \( vdist_{\min}(p, h_i) \) for any \( p \in c \). Consider \( c_o \) in Figure 3 and the labels \( A;(F, 2.2) \) and \( A;(G, 2) \) of \( H(c_o) \) in Figure 4. Here, \( max Edist(c_o, G) + d(G, A) = 1.6 + 2 = 3.6 \) is smaller than \( min Edist(c_o, F) + d(F, A) = 2 + 2.2 = 4.2 \). Thus, the label \( A;(F, 2.2) \) can be pruned from \( H(c_o) \) (see the red struckthrough label in Figure 4).

**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 \( u \) or \( v \) is an edge in the non-taut region of \( v \) or \( u \). Note that this label is not needed to remove a dead-end label because the only edge for any shortest path. This is because the triangle list is sorted according to the angles. To check whether a point \( p \) is in the taut region of \( v_j \), a binary search is conducted to find the triangle \( \triangle abv_j \) whose angle overlaps with the angle \( \overline{p vu} \). \( p \) 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 \( O(\log T) \).

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

**Lemma 2.** For every point \( p \in c \), \( min Edist(c, v_j) \leq d(p, v_j) \) for any vertex \( v_j \). Thus, \( mindist(c, v_j) \leq \sqrt{min Edist(c, v_j)^2 + d^2} \) which implies \( mindist(c, h_i) \leq vdist_{\min}(p, h_i) \).

**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. To compute the shortest distance, \( \beta \) takes \( O(|H(c_o)| + |H(c_t)|) \) to find all common hub nodes. For each common hub node \( h_i \), the algorithm computes \( vdist_{\min}(s, h_i) \) and \( vdist_{\min}(t, h_i) \). Then, for each via label \( v_j \), the algorithm computes \( vdist_{\min}(p, h_i) \) using Algorithm 1. Therefore, the total cost of the algorithm is \( O(|H(c_o)| + |H(c_t)| + |H(c_o) \cap H(c_t)| \times L \log T) \).

In practice, \( L \) is typically a small value especially when grid cells are small (see Table 6). Also, \( O(\log T) \) cost is only required for the labels for which the if condition at line 3 in Algorithm 1 is true.

<table>
<thead>
<tr>
<th>#Maps</th>
<th># Queries</th>
<th># Vertices</th>
<th>#CV</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAO</td>
<td>156</td>
<td>159,464</td>
<td>1727.6</td>
</tr>
<tr>
<td>DA</td>
<td>67</td>
<td>68,150</td>
<td>1182.9</td>
</tr>
<tr>
<td>BG</td>
<td>75</td>
<td>93,160</td>
<td>1294.4</td>
</tr>
<tr>
<td>SC</td>
<td>75</td>
<td>198,224</td>
<td>11487.5</td>
</tr>
</tbody>
</table>

Table 3: Total number of maps and queries, and average number of vertices and convex vertices (#CV) in each benchmark.
Table 4: Average cost (build time and memory) per map and maximum cost among all maps for each of the four benchmarks.

<table>
<thead>
<tr>
<th>Map</th>
<th>Build Time (Secs)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>EHL</td>
<td>EPS-CPD</td>
</tr>
<tr>
<td></td>
<td>Avg</td>
<td>Max</td>
</tr>
<tr>
<td>DAO</td>
<td>4.41</td>
<td>87.8</td>
</tr>
<tr>
<td>DA</td>
<td>1.52</td>
<td>4.95</td>
</tr>
<tr>
<td>BG</td>
<td>9.09</td>
<td>31.8</td>
</tr>
<tr>
<td>SC</td>
<td>128</td>
<td>603</td>
</tr>
</tbody>
</table>

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 4 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 5, we sort the queries by the number of nodes expansions needed by A* search to solve them.
proxy for how challenging a query is) and the x-axis corresponds 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 lower-bounds) 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)). EHL uses a lower bound of labels to solve these queries efficiently.

Query Performance: Table 6 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 6 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 #C_N = |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 common hub nodes for 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 lower-bounds) 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)). EHL uses a lower bound of labels to solve these queries efficiently.

Preprocessing Time and Space: Table 5 shows that removing PR1 significantly increases the build time. The 4x case 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 decided to 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.
Table 6: Effect of the pruning rules (PR) and the optimisation (OP) on query performance (see Table 5 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 $\mu$s) 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.

<table>
<thead>
<tr>
<th>Map</th>
<th>C</th>
<th>Query Performance ($\mu$s)</th>
<th>EHL (Final)</th>
<th>EPS-CPD</th>
<th>EPS-HL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>-PR1</td>
<td>-PR2</td>
<td>-PR3</td>
<td>-PR4</td>
</tr>
<tr>
<td>DAO</td>
<td>1x</td>
<td>2.18</td>
<td>2.17</td>
<td>2.22</td>
<td>2.05</td>
</tr>
<tr>
<td></td>
<td>4x</td>
<td>3.83</td>
<td>3.49</td>
<td>3.74</td>
<td>4.00</td>
</tr>
<tr>
<td></td>
<td>16x</td>
<td>11.07</td>
<td>8.58</td>
<td>8.80</td>
<td>11.76</td>
</tr>
<tr>
<td></td>
<td>64x</td>
<td>37.26</td>
<td>29.36</td>
<td>29.48</td>
<td>45.41</td>
</tr>
<tr>
<td>DA</td>
<td>1x</td>
<td>1.85</td>
<td>1.72</td>
<td>1.85</td>
<td>1.63</td>
</tr>
<tr>
<td></td>
<td>4x</td>
<td>3.30</td>
<td>2.61</td>
<td>2.91</td>
<td>2.92</td>
</tr>
<tr>
<td></td>
<td>16x</td>
<td>7.78</td>
<td>5.42</td>
<td>5.60</td>
<td>7.09</td>
</tr>
<tr>
<td></td>
<td>64x</td>
<td>20.49</td>
<td>14.73</td>
<td>15.06</td>
<td>20.90</td>
</tr>
<tr>
<td>BG</td>
<td>1x</td>
<td>1.78</td>
<td>1.82</td>
<td>1.83</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>4x</td>
<td>2.88</td>
<td>2.68</td>
<td>2.84</td>
<td>2.72</td>
</tr>
<tr>
<td></td>
<td>16x</td>
<td>7.39</td>
<td>5.96</td>
<td>6.27</td>
<td>7.41</td>
</tr>
<tr>
<td></td>
<td>64x</td>
<td>25.73</td>
<td>19.25</td>
<td>19.40</td>
<td>27.84</td>
</tr>
<tr>
<td>SC</td>
<td>1x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>4x</td>
<td>6.52</td>
<td>6.10</td>
<td>6.67</td>
<td>6.62</td>
</tr>
<tr>
<td></td>
<td>16x</td>
<td>20.02</td>
<td>14.86</td>
<td>16.43</td>
<td>20.61</td>
</tr>
<tr>
<td></td>
<td>64x</td>
<td>97.49</td>
<td>64.45</td>
<td>65.09</td>
<td>111.15</td>
</tr>
</tbody>
</table>

Table 6: Effect of the pruning rules (PR) and the optimisation (OP) on query performance (see Table 5 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 $\mu$s) 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.

Table 6 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 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


