EHL*: Memory-Budgeted Indexing for Ultrafast Optimal Euclidean Pathfinding

Jinchun Du, Bojie Shen, Muhammad Aamir Cheema

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

Abstract

The Euclidean Shortest Path Problem (ESPP) is a classic problem which requires finding the shortest path in a Euclidean plane with polygonal obstacles. The state-of-the-art solution, Euclidean Hub Labeling (EHL), offers ultra-fast query performance but comes with significant memory overhead, requiring up to tens of gigabytes of storage on large maps, limiting its use in memory-constrained environments like mobile phones. Additionally, EHL's memory usage can only be determined after index construction, and while it provides a memory-runtime tradeoff, it does not fully optimize memory utilization. In this work, we introduce an improved version of EHL, called EHL*, which overcomes these limitations. A key contribution of EHL* is its ability to create an index that adheres to a specified memory budget while optimizing query runtime performance. Moreover, EHL* can leverage pre-known query distributions, a common scenario in many real-world applications, to further enhance runtime efficiency. Our results show that EHL* can reduce memory usage by up to 10-20 times without much impact on query runtime performance compared to EHL, making it a highly effective solution for optimal pathfinding in memory-constrained environments. We also present a theoretical analysis comparing EHL* with EHL, providing insights into their indexing and query processing cost.

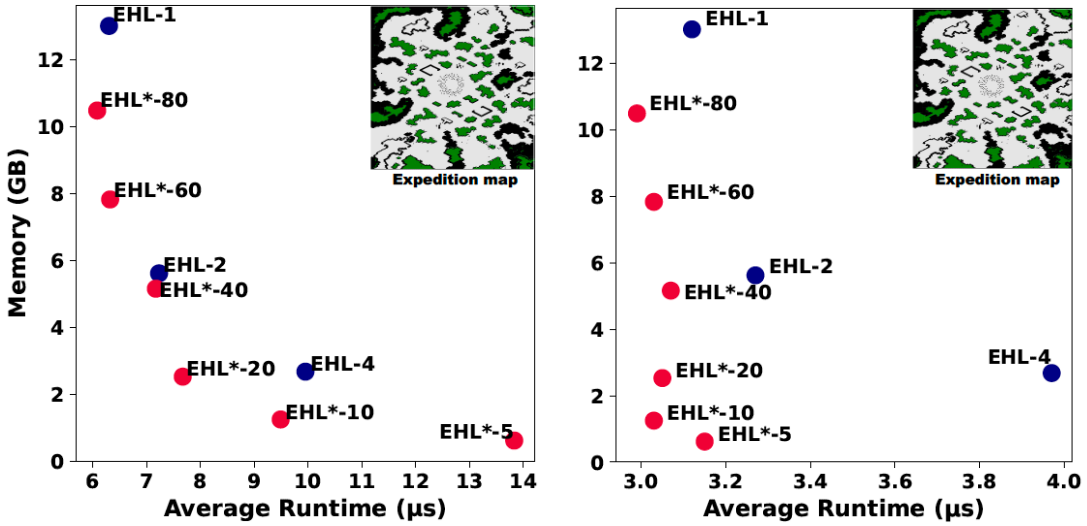
Introduction

The Euclidean Shortest Path Problem (ESPP) finds the shortest obstacle-avoiding path between a given source and target for an agent to travel. ESPP is a well-studied problem with various real-world applications, including computer games (Sturtevant 2012b), robotics (Mac et al. 2016), and indoor navigation (Cheema 2018). In many of these applications, it is crucial to compute the optimal shortest path as fast as possible, especially for large-scale deployments that require calculating tens of thousands of paths per second. This challenge has led to the development of numerous algorithms, such as navigation-mesh-based planners like Polyanya (Cui, Harabor, and Grastien 2017), variations of visibility graphs like hierarchical sparse visibility graphs (Oh and Leong 2017), and oracle-based approaches like End Point Search (EPS) (Shen et al. 2020) and Euclidean Hub Labeling (EHL) (Du, Shen, and Cheema 2023). Recent studies explore related directions: sub-optimal path planning on large-scale datasets (Funke et al. 2024), shortest-distance computation on visibility graphs (Funke et al. 2025), and heuristic methods to speed up Polyanya (Koch and Funke 2025).

Among the existing optimal algorithms, the state-of-the-art is EHL, which exploits Hub Labeling (Abraham et al. 2011), the leading shortest path approach for graphs such as road networks. During the preprocessing phase, EHL constructs a visibility graph on the convex vertices of the polygonal obstacles and precomputes the hub labeling on this graph. The precomputed hub labels of each vertex are copied to each grid cell of a uniform grid that is visible from the vertex. During the online phase, EHL considers the labels stored in the grid cells containing the start and target, respectively. These labels are joined to obtain the shortest path via a merge-join process.

EHL is 1-2 orders of magnitude faster than existing algorithms and performs best when grid cells are small (i.e., there are many grid cells) (Du, Shen, and Cheema 2023). However, in this case, it requires substantial memory to store the labels, limiting its use in memory-constrained environments. Although EHL offers a memory-runtime trade-off by increasing the grid cell size to reduce memory usage at the cost of longer running times, it still faces two limitations: 1) Memory required by EHL cannot be predicted in advance and is only known after the index is constructed. This is problematic for smaller devices with fixed memory budgets that need to create an index within the budget while optimizing runtime. 2) In many real-world cases, queries often follow a specific distribution, with certain areas of maps receiving more queries than others. While works in other domains (Sheng et al. 2023; Tzoumas, Yiu, and Jensen 2009) show that this information can be exploited to improve performance, EHL does not take advantage of this information, even when it is available.

In this paper, we address these limitations of EHL and propose EHL*, an improved version of EHL that works within a memory budget $\mathcal{B}$ to optimize the query runtime while still guaranteeing to find the optimal solution. Moreover, it is also able to exploit the expected query distribution if it is known beforehand. EHL* builds on EHL by incorporating a compression phase that merges the labels of adjacent grid cells into arbitrary-shaped regions. Each merging



(a) Unknown distribution

(b) Known distribution

Figure 1: Memory–runtime tradeoff for EHL and EHL*

effectively reduces the memory because the labels that were stored twice in two different cells/regions are now likely to be stored once in one larger region. The key is to carefully select which cells or regions to merge, ensuring that memory savings are achieved without significantly increasing query runtime. EHL* continues to merge chosen neighboring regions until the required memory drops below $\mathcal{B}$.

Figure 1 illustrates the memory-runtime trade-off achieved by EHL and EHL* on the Expedition map from the StarCraft benchmark (Sturtevant 2012a). Following the methodology of the original work (Du, Shen, and Cheema 2023), EHL-1 represents the algorithm with grid cells the same size as those in the benchmark. EHL-2 and EHL-4 correspond to versions where the grid cell sizes are increased by factors of 2 and 4 in each dimension, respectively. EHL*-x denotes EHL* with a memory budget $\mathcal{B}$ set to x% of the memory used by EHL-1. Figure 1a presents results assuming the query distribution is unknown. EHL* not only allows specifying a memory budget but also delivers a superior memory-runtime trade-off, e.g., EHL*-60 reduces memory usage by 60% compared to EHL-1 without significantly increasing runtime. Figure 1b shows results when queries are assumed to be clustered in two specific areas of the map, and EHL* leverages this information to strategically merge cells/regions. The results are even more impressive, with EHL*-5 reducing memory from over 12GB to around 600MB while maintaining similar query runtime. Alongside strong empirical performance, we also provide a comprehensive theoretical analysis comparing EHL* with EHL in terms of memory usage, index construction cost, and query processing cost.

Preliminaries

An obstacle is represented by a **polygon** that consists of a closed set of edges, each associated with two points at its ends, known as vertices. A **convex vertex** (resp. **non-convex vertex**) is a vertex located at a convex (resp. concave) corner of the polygon. Two points are said to be **visible** to each other (also known as **co-visible**) iff there exists a straight line between them that does not pass through any obstacle. A **path** $\mathcal{P}$ between a source s and target t is a sequence of points $\langle p_1, p_2, \cdots, p_n \rangle$ where $p_1 = s$, $p_n = t$ and every successive pair of points p_i and p_{i+1} ($i < n$) is co-visible. The **length** of a path $\mathcal{P}$ is the total sum of the Euclidean distances between each consecutive pair of points along the path, denoted as $|\mathcal{P}|$, i.e., $|\mathcal{P}| = \sum_{i=1}^{n-1} Edist(p_i, p_{i+1})$ where $Edist(p_i, p_{i+1})$ is the Euclidean distance between p_i and p_{i+1}. The **Euclidean Shortest Path Problem (ESPP)** computes the shortest obstacle-avoiding path between a given s and t with the minimum length, denoted as $sp(s, t)$. The shortest distance (i.e., the length of shortest path) between s and t, denoted as $d(s, t)$, i.e., $|sp(s, t)| = d(s, t)$.

Euclidean Hub Labeling (EHL)

This section introduces Euclidean Hub Labeling (EHL), the state-of-the-art algorithm we enhance in this work.

Offline Preprocessing

Building Hub Labelling on Visibility Graph: Visibility graph is a popular concept utilized in many pathfinding algorithms for Euclidean space, e.g., (Oh and Leong 2017; Shen et al. 2020). The graph $G = (V, E)$ consists of a set of convex vertices V and a set of edges E, where each edge $e \in E$ connects a pair of co-visible vertices in V. Given the constructed visibility graph $G = (V, E)$, Hub Labeling (HL) computes and stores, for each vertex $v_j \in V$, a set of hub labels denoted as $H(v_j)$. Each hub label in $H(v_j)$ is a tuple (h_i, d_{ij}) that includes: (i) a hub vertex $h_i \in V$; and (ii) the shortest distance d_{ij} between the hub vertex h_i and v_j. HL must satisfy the *coverage property* (Cohen et al. 2003), which means that for every pair of reachable vertices $v_j \in V$

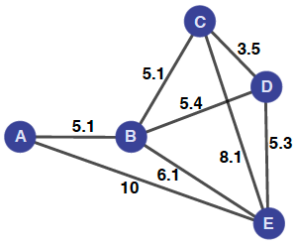


Figure 2: Visibility graph for the example in Figure 3

Vertex	Hub labels
A	(A, 0), (B, 5.1), (E, 10)
B	(B, 0)
C	(B, 5.1), (C, 0)
D	(B, 5.4), (D, 0)
E	(B, 6.1), (D, 5.3), (E, 0)

Table 1: Hub labeling for the graph in Figure 2

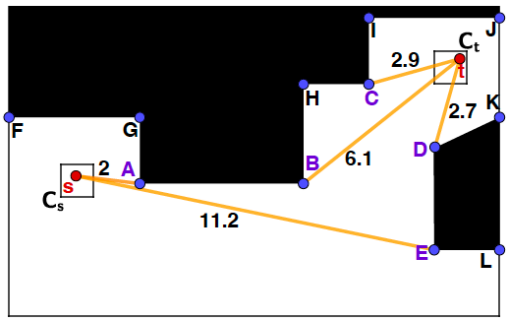


Figure 3: Euclidean plane with polygonal obstacles

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 efficiently determined as follow:

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

According to Eq. (1), the **shortest distance** between v_s and v_t can be calculated by performing a simple scan of the sorted label sets $H(v_s)$ and $H(v_t)$. The complexity is $O(|H(v_s)| + |H(v_t)|)$, where $|H(v_j)|$ represents the number of labels in $H(v_j)$. The shortest path can be retrieved by utilizing the successor node for each label. We refer the reader to (Li et al. 2017) for details.

Example 1. *Figure 2 shows the visibility graph constructed for the example in Figure 3, containing 5 convex vertices. Table 1 shows the hub labels for this visibility graph. The computation of d(E,A) involves scanning the hub labels for vertices E and A, during which two common hub nodes, B and E, are identified. Here, d(E,B) + d(B,A) = 6.1 + 5.1 = 11.2 and d(E,E) + d(E,A) = 0 + 10. Thus d(E,A) = 10.*

Computing Euclidean Hub Labelling: With the HL constructed on the visibility graph, we can efficiently solve ESPP when both the start s and target t are at convex vertices. However, since s and t can be arbitrary locations, EHL adapts HL to handle these cases, as follows. EHL overlays a uniform grid on the map, with grid size adjustable to manage memory usage. For each cell c, EHL computes:

- A visibility list L_c: This list consists of every convex vertex v such that at least part of c is visible from v.
- A via-labels list $VL(c)$: For each vertex v_j in the visibility list L_c, we consider each hub label $(h_i, d_{ij}) \in H(v_j)$ and insert a via-label $h_i{:}(v_j, d_{ij})$ into $VL(c)$. Intuitively, this label indicates that 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, with a distance of d_{ij} between them.

Within each cell c, there may be multiple via-labels containing the same hub nodes. We use $VL_{h_i}(c)$ to denote the set of all via-labels in c that have the same hub node h_i. $H(c)$ represents the set of unique hub nodes for the labels stored in c. For faster query processing, $H(c)$ is sorted by hub nodes h_i to efficiently join the hub labels of two different cells.

Example 2. *Table 2 presents the via-labels for two grid cells, c_s and c_t, as illustrated in Figure 3. To compute these via-labels, the EHL algorithm first identifies the visible convex vertices from each cell, forming visibility lists $L_{c_s} = \{A, B, E\}$ and $L_{c_t} = \{B, C, D\}$. Consider vertex A, with its hub label $(B, 5.1) \in H(A)$ as shown in Table 1. Since A is visible from c_s, a via-label $B : (A, 5.1)$ is added to indicate that there is a path from c_s to B via A, with the shortest distance $d(A, B) = 5.1$. The visibility list L_{c_s} includes A, B, and E, so the hub labels of these vertices are used to insert the via-labels for c_s: $A : (A, 0)$, $B : (A, 5.1)$, $E : (A, 10)$, $B : (B, 0)$, $B : (E, 6.1)$, $D : (E, 5.3)$, and $E : (E, 0)$. Similarly, the via-labels for c_t are generated.*

Online Query Processing

First, we introduce minimal via-distance and how EHL computes it using the stored labels.

Minimal Via-Distance: The via-distance $vdist(p, v_j, h_i)$ represents 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 $vdist(p, v_j, h_i) = Edist(p, v_j) + d_{ij}$ where $Edist(p, v_j)$ is the Euclidean distance between p and v_j. 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 minimum via-distance $vdist_{min}(p, h_i)$ between p and h_i as:

$$vdist_{min}(p, h_i) = \min_{h_i:(v_j, d_{ij}) \in VL_{h_i}(c)} vdist(p, v_j, h_i) \quad (2)$$

Computing Shortest Distance: Let c_s and c_t be the grid cells containing s and t, respectively. If s and t are co-visible, then the shortest distance $d(s, t)$ is simply the Euclidean distance between them (i.e., $d(s, t) = Edist(s, t)$). If they are not co-visible, $d(s, t)$ is computed as follows:

$$d(s, t) = \min_{h_i \in H(c_s) \cap H(c_t)} vdist_{min}(s, h_i) + vdist_{min}(t, h_i) \quad (3)$$

Once $d(s, t)$ is determined, the shortest path $sp(s, t)$ can be retrieved using the successor nodes. Please see (Du, Shen, and Cheema 2023) for details.

Example 3. *In Table 2, $H(c_s)$ and $H(c_t)$ share two common hub nodes, B and D. When B is identified, the $vdist_{min}(s, B)$ is computed using each via-label:*

$H(c_s)$	Via Labels $VL_{h_i}(c_s)$
A:	(A, 0)
B:	(A, 5.1), (B, 0), (E, 6.1)
D:	(E, 5.3)
E:	(A, 10), (E, 0)

$H(c_t)$	Via Labels $VL_{h_i}(c_t)$
B:	(B, 0), (C, 5.1), (D, 5.4)
C:	(C, 0)
D:	(D, 0)

Table 2: Via-labels for c_s and c_t shown in Figure 3

Algorithm 1: EHL*: Compression Phase

Input: EHL: Euclidean Hub Labeling, $\mathcal{B}$: memory budget.
Output: EHL*: the compressed EHL. M: a mapper that maps each grid cell to the merged region of EHL*.

1 `initializeScores`(EHL) ;
2 Initialize a min heap H with each grid cell of EHL;
3 Initialize a mapper M that maps each grid cell to its corresponding element in min heap H;
4 **while** *MemoryUsage* $> \mathcal{B}$ *and* $|H| > 1$ **do**
5 deheap an element e from H ;
6 $r \leftarrow$ `adjacentRegionSelection`(e,M) ;
7 $e \leftarrow e \cup r$; `// merge r into e`
8 update M by mapping each cell $c \in r$ to e ;
9 remove r from H and insert e in H ;
10 **return** EHL* and M;

$vdist(s, A, B) = EDist(s, A) + d(A, B) = 2 + 5.1 = 7.1$ *and* $vdist(s, E, B) = EDist(s, E) + d(E, B) = 11.2 + 6.1 = 17.3$. *The label* $B : (B, 0)$ *is ignored since B is not visible from s, resulting in* $vdist_{min}(s, B) = vdist(s, A, B) = 7.1$. *Similarly,* $vdist_{min}(t, B) = 6.1$ *is computed, and the distance is updated to* $dist = vdist_{min}(s, B) + vdist_{min}(t, B) = 13.2$. *For hub node D,* $vdist_{min}(s, D) + vdist_{min}(t, D) = 16.5 + 2.7 = 19.2$. *The algorithm returns the smaller value* $d(s, t) = 14$.

Euclidean Hub Labeling Star (EHL*)

Offline Preprocessing

The query performance of EHL depends on the number of labels stored in c_s and c_t – the fewer, the better. Thus, EHL performs better when the grid cells are small because smaller cells contain fewer labels. However, memory usage increases with smaller cells due to the likelihood of the same labels being stored in multiple cells. Memory usage in EHL can be significantly reduced without greatly impacting query performance by merging cells, as long as the number of labels per cell does not increase substantially. EHL* aims to achieve this as we detail next.

Recall that each via-label stored in a grid cell c is essentially a hub label copied from a convex vertex v, from which at least part of c is visible. As a result, neighboring cells often share many of the same hub labels. By merging adjacent cells that have a high overlap in hub labels, memory usage can be reduced without significantly impacting performance. EHL* leverages this insight by introducing a compression phase that iteratively checks adjacent grid cells and merges those with a high degree of hub label similarity.

Algorithm 1 details the compression phase. From this point on, we use "region" to refer to either a single grid cell or the shape resulting from merging two or more cells. The algorithm receives the constructed EHL and a memory budget $\mathcal{B}$ as input. It begins by calling the function `initializeScores`(EHL) to assign a score to each grid cell in the EHL (line 1). These scores help determine which regions to merge in each iteration. We will explain the score computation in the next section. Next, the algorithm initializes a min heap H by inserting each grid cell along with its corresponding score (line 2) and sets up a mapper M that links each grid cell to its region in H (line 3). This mapper allows for efficient tracking of the merged regions associated with each grid cell.

Once initialization is complete, the algorithm begins compressing the EHL through a while loop. In each iteration, the algorithm extracts the top element e from the heap H (line 5) and calls the function `adjacentRegionSelection`(e, M), which examines each adjacent region of e and chooses a region r to merge with e (line 6). Two regions are considered adjacent if they share a boundary. The mapper M is used to efficiently identify e's adjacent regions. Specifically, a region e' is adjacent to e if a cell c adjacent to e belongs to e' (as determined by the mapper M). The criteria for selecting the region r are discussed later. Once the region r is selected for merging with element e, they are merged at line 7 as follows.

The region of element e is expanded to include r. To maintain the correctness, the via-labels of r and e are merged by taking their union. Specifically, for each via-label $h_i{:}(v_j, d_{ij})$ of r, we add it to $VL(e)$ if $h_i{:}(v_j, d_{ij}) \notin VL(e)$; otherwise, we ignore it. Finally, the score of e is incremented by the score of r, i.e., $s(e) = s(e) + s(r)$, where $s(x)$ denotes score of x.

After merging region r into region e, the algorithm updates the mapper M by reassigning each grid cell $c \in r$ to the expanded region e, ensuring that adjacent regions are correctly tracked in future iterations (line 8). The region r is then removed from the heap H to prevent redundant merges, and the expanded region e is reinserted into H with its updated score (line 9). This process repeats until the memory usage is less than or equal to the memory budget $\mathcal{B}$ (line 4), at which point EHL* has been successfully constructed within the given budget. The algorithm also halts if all cells have merged into a single region (i.e., the heap contains only one element), which indicates that EHL* cannot be constructed within the memory limit. In our experiments, this situation only arises for certain small maps when the memory budget is set to 1%, which is already quite limited.

The algorithm initially calculates the total memory usage, and throughout the iterations, it tracks the reduction in mem-

$H(c_s)$	Via Labels $VL_{h_i}(c_s)$
A:	(A, 0)
B:	(A, 5.1), (B, 0), (E, 6.1), (D, 5.4)
D:	(E, 5.3), (D, 0)
E:	(A, 10), (E, 0)

Table 3: Via-labels for c_s after merging c_3 into it. Newly inserted labels are shown in red.

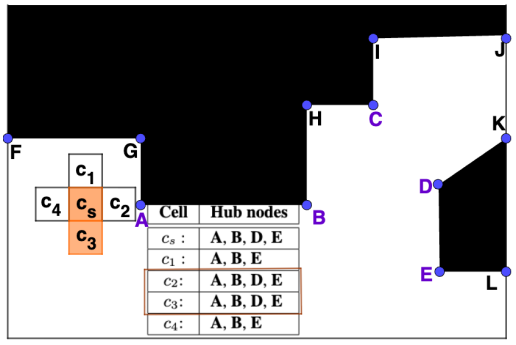


Figure 4: Grid cell c_s with its adjacent neighbors

ory by subtracting the amount saved through label merging. The final output of the algorithm includes the compressed EHL (i.e., EHL*) and the updated mapper M, which enables faster query processing as discussed later (line 10).

Initializing Scores: The scores assigned to grid cells play a crucial role in the compression phase, as cells with lower scores are more likely to be merged. These scores can be tailored based on the application's specific needs. For example, if certain areas of the map require highly efficient query runtimes, the cells in those areas can be given higher scores to reduce the likelihood of being merged, thereby retaining fewer labels and improving performance. In the absence of such requirements, EHL* assigns the same score to all cells, i.e., $s(c) = 1$ for each cell. This simple approach ensures that the merged regions have approximately equal sizes and similar number of hub labels, leading to relatively consistent query performance across different map areas.

Adjacent Region Selection: To reduce memory usage without significantly affecting query runtime performance, the key is to keep the number of labels per region as small as possible. This can be accomplished by merging regions with substantial overlap in their hub labels. EHL* does this by examining all adjacent regions R of e and selecting the region $r \in R$ whose hub labels exhibit the highest Jaccard similarity with those of e (ties are broken arbitrarily).

$$r = \arg\max_{r' \in R} \left(\frac{|H(r') \cap H(e)|}{|H(r') \cup H(e)|} \right) \tag{4}$$

Example 4. *Figure 4 shows a grid cell c_s that needs to be merged with one of its four neighboring cells: c_1 to c_4. The table in the figure lists the hub nodes for each of these cells. Cells c_1 and c_4 have a Jaccard similarity of 0.75 with c_s, while both c_2 and c_3 have a similarity of 1. c_2 and c_3 have the highest similarity and we arbitrarily select c_3 for merging with c_s. Table 3 shows the via labels for c_s after the merge with c_3. Two new via labels (highlighted in red) from c_3 are added to the label set of c_s as a result of the merge.*

Online Query Processing

EHL stores labels in uniform grid cells whereas EHL* differs by merging these cells into arbitrary-shaped regions. Despite this, both EHL and EHL* maintain the via-label list $VL(e)$ within each region e, so the query processing algorithm remains the same. Note that EHL* needs to first locate the regions e_s and e_t that contain s and t, respectively. This can be easily done in $O(1)$ by identifying the grid cells that contain s and t and then using the mapper M to find the corresponding regions e_s and e_t.

Query Workload-Aware EHL*

In many real-world scenarios, such as computer games, query workload can often be predicted because certain regions of a map are visited more frequently due to gameplay mechanics, player behavior patterns, or specific objectives. *Workload-Aware algorithms* (Sheng et al. 2023) can leverage such knowledge if, for example, expected query distribution is known or can be predicted. We adapt EHL* for workload-aware scenarios by modifying its scoring computation and adjacent region selection.

Initializing Scores: If the query distribution is known or can be predicted, EHL* can improve performance by keeping regions with higher expected workloads smaller, which in turn reduces the number of labels and enhances performance. To achieve this, EHL* prioritizes merging regions with lower expected query activity. Let w_c represent the expected workload of a cell c, defined as the anticipated number of queries where s or t falls within c. EHL* assigns an initial score to each cell as $s(c) = 1 + w_c$, ensuring that no cell has a zero score, and that cells with higher workloads have higher scores, making them less likely to be merged.

Adjacent Region Selection: EHL* still prioritizes similarity when selecting adjacent regions for merging. However, instead of solely considering similarity, we also factor in the expected workload, represented by the scores $s(r)$ for each region r. The selection of an adjacent region $r \in R$ is now determined by combining these two criteria:

$$r = \arg\max_{r' \in R} \left((1-\alpha) \left(\frac{|H(r') \cap H(e)|}{|H(r') \cup H(e)|} \right) + \alpha \left(\frac{1}{s(r')} \right) \right) \tag{5}$$

Note that regions with higher workloads $s(r')$ are less likely to be selected for merging. The weighting factor α, set to 0.2 based on experimental results, balances the influence of similarity and workload. Increasing or decreasing α degraded query performance, indicating that Jaccard similarity has a stronger impact on performance than workload weighting, even when the query distribution is known.

Theoretical Analysis

Key Terminology and Concepts: Let N denote the total number of grid cells in EHL and H[1] the average number of labels per cell. Since the memory is dominated by the total number of labels stored in the index, the space complexity for EHL is $O(HN)$. Let b be the memory reduction factor for EHL*, where $0 < b \leq 1$, i.e., if EHL uses memory M, then EHL* has a memory budget $\mathcal{B} = bM$.

[1]For analytical tractability, we adopt a simplified theoretical model in which each cell contains H labels, capturing average-case behavior rather than an exact per-cell value.

Recall that EHL* iteratively deheaps regions from a heap and merges each region with the neighboring region having the highest Jaccard similarity. Assume a region e is merged with the most similar neighboring region r. Let θ be the Jaccard similarity between the two regions being merged, i.e.,

$$\theta = \frac{|H(e) \cap H(r)|}{|H(e) \cup H(r)|} \tag{6}$$

where $|H(e)|$ and $|H(r)|$ denote the number of hub labels in e and r, respectively. For simplicity, assume both regions have the same number of labels X. Also, note that we have $|H(e) \cap H(r)| = |H(e)| + |H(r)| - |H(e) \cup H(r)| = 2X - |H(e) \cup H(r)|$. So, we have $\theta = \frac{2X - |H(e) \cup H(r)|}{|H(e) \cup H(r)|}$ which gives us $|H(e) \cup H(r)| = 2X/(1+\theta)$. In other words, if each region has X labels, then the number of labels in the merged region is $2X/(1+\theta)$.

Memory Reduction: The total number of labels stored in EHL is HN. EHL* iteratively merges regions with the neighboring regions to reduce the memory. Assume that, in round one, EHL* iteratively merges each cell with a neighboring cell with the highest similarity. Since each cell contains H labels, each merged region has $2H/(1+\theta)$ labels. Thus, at the conclusion of round one, there are a total of $N/2$ regions where each region (consisting of two merged cells each) has $2H/(1+\theta)$ labels. Thus, the total memory after the first round of merges is $\frac{2H}{(1+\theta)} \times \frac{N}{2} = HN/(1+\theta)$. Now, assume that, in the second round, all these regions containing two cells each are iteratively merged with the neighboring regions again. Each region has $2H/(1+\theta)$ labels. After merging two such regions, the merged region (containing 4 cells) will have $2^2H/(1+\theta)^2$ labels. Thus, after the second round of merges, the total number of merged regions is now $N/2^2$ (each region contains 4 cells), and the total memory is $\frac{2^2H}{(1+\theta)^2} \times \frac{N}{2^2} = HN/(1+\theta)^2$. Generalizing further, after i rounds of merges, there will be $N/2^i$ regions remaining, with each region containing $2^iH/(1+\theta)^i$ labels. Consequently, the total memory reduces to $HN/(1+\theta)^i$ after i rounds of merges[2].

The algorithm terminates when the memory of EHL* is reduced to $bM = bHN$, where $M = HN$ is the memory consumed by EHL. In other words, the algorithm terminates when $HN/(1+\theta)^i = bHN$. Solving for i, we get $i = \frac{\log 1/b}{\log(1+\theta)}$.

Index Construction Overhead: Note that EHL* first constructs the same index as EHL and then uses a compression phase to reduce the memory. In this section, we analyse the cost of this compression phase which represents the index construction overhead for EHL*.

[2]This analysis assumes that the Jaccard similarity θ remains unchanged as the region size increases. In practice, the Jaccard similarity decreases as the regions become bigger. To address this, we abuse the notation and redefine θ to be the average Jaccard Similarity for two regions at the i-th round of merging (i.e., for the largest regions). Defining θ in this way gives the worst-case behavior for EHL*.

	#Maps	# Queries	#Vertices	#CV
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
CITY	10	37,110	14990.9	7575.6

Table 4: Total # of maps and queries, and average # of vertices and convex vertices (#CV) in each benchmark.

The cost of merging two level-i regions can be computed as follows. Each level-i region contains 2^{i-1} cells. Consequently, the number of neighboring cells for this region is approximately $4 \cdot 2^{(i-1)/2}$ (because length of each side of the region is roughly $2^{(i-1)/2}$ cells). To determine each unique neighboring region, the algorithm processes each of these cells, in constant time, to determine which region it belongs to. Thus, finding all unique neighboring regions requires $O(2^{(i-1)/2})$. The total number of unique neighboring regions remains constant because, at higher levels, the regions themselves become larger. Each neighboring region has $2^iH/(1+\theta)^i$ labels, so computing the Jaccard similarity for all these neighboring regions requires $O(2^iH/(1+\theta)^i)$. After computing the similarities, the labels are merged with the one that has the highest similarity, using the merge operation similar to the one used in merge sort, which also costs $O(2^iH/(1+\theta)^i)$. Thus, the total cost of merging two regions at level i is $O(2^{(i-1)/2} + 2^iH/(1+\theta)^i)$.

Note that the total number of merge operations is $O(N)$ because there are $N/2$ merge operations in round one, $N/4$ in round two and so on. Thus, the total cost for all merges is $O((2^{(i-1)/2} + 2^iH/(1+\theta)^i)N)$. Additionally, during the compression phase, the algorithm uses a heap to prioritize grid cells or regions, which requires $O(\log N)$ per heapify operation to determine the next region to merge. Since the number of heapify operations is bounded by $O(N)$, the total cost of this is $O(N \log N)$. Thus, the total cost for the compression phase is $O(N(2^{(i-1)/2} + 2^iH/(1+\theta)^i + \log N))$ where $i = \frac{\log 1/b}{\log(1+\theta)}$.

Query Cost: As noted in the original paper that proposed EHL, the query complexity of EHL is primarily determined by the number of hub labels in the cells containing source and target locations. Thus, the query processing cost of EHL is roughly $O(H)$ whereas the query processing cost of EHL* is $O((2^iH/(1+\theta)^i)$ because each region in EHL* contains $2^iH/(1+\theta)^i$ labels. In other words, the query processing cost of EHL* is roughly $2^i/(1+\theta)^i$ times of that of EHL where $i = \frac{\log 1/b}{\log(1+\theta)}$.

It is worth noting that while the index construction cost and query processing cost grow exponentially with i (as expected when b approaches zero), in practice, i remains reasonably small for moderate values of b, as implied by the results shown in the next section.

Map	Query Set		EHL*						Competitors				
			80%	60%	40%	20%	10%	5%	EHL-1	EHL-2	EHL-4	EPS	Polyanya
DAO	Memory(MB)		105.5	78.8	52.4	26.3	13.6	7.5	134.8	57.8	27.6	0.21	-
	Build Time(Secs)		6.63	7.21	7.98	8.99	9.43	9.68	4.57	1.24	0.36	0.02	-
	Query(μs)	Unknown	2.02	2.08	2.20	2.49	3.12	4.40	1.84	2.28	3.19	25.94	176.50
		Cluster-2	1.00	0.99	0.98	0.98	1.03	1.15	0.98	1.27	1.88	7.22	17.51
		Cluster-4	1.22	1.21	1.20	1.23	1.33	1.59	1.21	1.55	2.23	10.01	32.66
		Cluster-8	1.35	1.34	1.35	1.43	1.58	2.00	1.33	1.69	2.43	11.69	39.60
DA	Memory(MB)		36.0	27.0	18.2	9.4	5.1	3.0	47.2	19.6	9.0	0.06	-
	Build Time(Secs)		2.76	2.88	3.05	3.26	3.37	3.43	1.67	0.51	0.18	0.11	-
	Query(μs)	Unknown	1.64	1.68	1.75	1.93	2.25	2.86	1.54	1.85	2.48	12.17	37.85
		Cluster-2	0.89	0.88	0.90	0.91	0.93	1.00	0.88	1.09	1.47	5.29	9.45
		Cluster-4	1.10	1.09	1.12	1.14	1.20	1.38	1.09	1.33	1.82	7.13	14.10
		Cluster-8	1.24	1.24	1.28	1.33	1.46	1.79	1.23	1.52	2.07	7.89	14.37
BG	Memory(MB)		150.8	112.8	75.0	37.6	19.3	10.3	193.0	71.2	31.8	0.12	-
	Build Time(Secs)		10.13	10.92	11.85	13.04	13.72	14.00	6.91	1.80	0.51	0.04	-
	Query(μs)	Unknown	1.34	1.37	1.43	1.55	1.79	2.25	1.27	1.48	1.93	9.77	13.25
		Cluster-2	0.75	0.74	0.78	0.80	0.79	0.82	0.73	0.85	1.11	4.32	6.84
		Cluster-4	0.96	0.97	1.03	1.02	1.04	1.11	0.95	1.10	1.42	6.32	10.12
		Cluster-8	1.09	1.11	1.17	1.19	1.23	1.37	1.08	1.23	1.59	7.29	11.66
SC	Memory(MB)		2060.7	1538.5	1017.3	503.9	253.6	130.9	2586.9	1096.4	522.5	2.3	-
	Build Time(Secs)		178.96	189.44	207.40	226.60	241.39	243.18	118.06	30.62	8.01	3.49	-
	Query(μs)	Unknown	3.43	3.50	3.96	4.29	5.05	6.47	3.34	4.13	5.40	68.20	261.16
		Cluster-2	1.83	1.93	1.79	1.94	1.81	1.89	1.92	2.22	2.93	30.56	83.23
		Cluster-4	2.48	2.43	2.49	2.41	2.43	2.63	2.57	2.96	3.89	43.74	104.74
		Cluster-8	2.95	2.97	3.02	3.19	3.06	3.47	3.08	3.59	4.66	51.94	120.16
CITY	Memory(MB)		11298.9	8389.5	5515.1	2715.1	1348.1	676.2	14211.1	5845.9	2716.6	5.7	-
	Build Time(Secs)		1894.12	1966.11	2090.59	2236.74	2347.68	2417.96	1693.2	425.5	110.4	10.9	-
	Query(μs)	Unknown	4.96	4.75	5.07	5.48	6.29	7.98	4.89	6.57	8.09	45.7	119.5
		Cluster-2	1.75	1.71	1.69	1.69	1.75	1.83	2.33	2.79	3.50	4.52	4.67
		Cluster-4	3.75	3.77	3.22	3.16	3.27	3.45	4.24	4.76	5.76	21.7	27.21
		Cluster-8	4.20	4.23	4.15	4.21	4.51	5.12	5.36	6.03	7.87	28.43	44.69

Table 5: Memory, build time and runtime comparison between EHL* and the competitors for different scenarios.

Experiments

Settings

We run our experiments on a 3.2 GHz Intel Core i7 machine with 64GB of RAM. The algorithms are all implemented in C++ and compiled with -O3 flag. Following existing studies, we conduct experiments on the widely used game map benchmarks (Sturtevant 2012a), which consist of four games: Dragon Age II (DA), Dragon Age: Origins (DAO), Baldur's Gate II (BG), and StarCraft (SC). Additionally, we enhance diversity by including maps from the CITY benchmark (Sturtevant 2012a), selecting the highest-resolution map for each of the 10 cities. These maps feature real-world urban obstacles, offering significantly higher grid resolution and distinct obstacle topologies compared to the game maps. Table 4 provides details of the benchmarks.

Queries. To represent scenarios where the query distribution is unknown, denoted as **Unknown**, we use the original dataset provided for each benchmark (Sturtevant 2012a). To simulate known query distributions, we generate synthetic query sets labeled **Cluster-x**, where x indicates the number of clustered regions (2, 4, or 8) within a map. These clusters are represented by rectangles, each generated by selecting a random central point within the traversable area of the map and ensuring the rectangle does not extend beyond map boundaries. The rectangles, with side lengths set to 10% of the width and height of the map, are positioned to ensure at least one other rectangle is reachable, preventing isolation. We generate source s and target t pairs as randomly selected locations within these rectangles ensuring that the path between s and t exists for each generated query. Using this approach, we generate (i) 500,000 s and t pairs as historical queries on the map, which EHL* uses to calculate the workload for each cell and build the index, and (ii) 2,000 s and t pairs to evaluate runtime performance. To evaluate runtime, each query is run five times, and the average is reported.

Algorithms Evaluated. We compare our approach with the state-of-the-art algorithm[3], EHL-x (Du, Shen, and Cheema 2023), where x denotes the cell size in EHL, varying x from 1 to 4 (similar to the original work). Additionally, we compare our method with Polyanya[4] (Cui, Harabor, and Grastien 2017), an optimal online pathfinding algorithm that runs on a navigation mesh, and EPS[5] (Shen et al. 2020), an optimal offline pathfinding algorithm that utilizes a Compressed Path Database (CPD) (Strasser, Harabor, and Botea 2014). We evaluate EHL* under different memory budgets, denoted as

[3]https://github.com/goldi1027/EHL

[4]https://bitbucket.org/mlcui1/polyanya

[5]https://github.com/bshen95/End-Point-Search

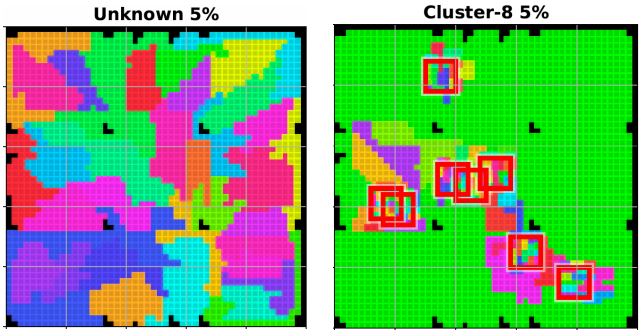


Figure 5: Distribution of merged grids for the arena map for different cluster scenarios at 5% memory. The red rectangles indicate cluster regions on the arena map, while black polygons represent obstacles.

EHL*-x, where x represents the percentage of memory used compared to EHL-1, e.g., the budget for EHL*-60 is 60% of the memory used by EHL-1. For reproducibility, implementation of EHL* is available online[6].

Experiment 1: Preprocessing Time and Space

Table 5 compares the average memory usage (in MB) and build time (in secs) of EHL* for different memory budgets with EHL, EPS and Polyanya. Compared to EHL, EHL* offers precise memory usage control, successfully reducing memory down to 5%, making it adaptable to different devices based on application needs. While EHL also offers a memory-runtime trade-off by increasing cell sizes, its memory usage cannot be predicted before index construction and lacks fine-tuned control. EPS and Polyanya, however, require significantly less memory but at the expense of a much higher query runtime as shown shortly. Polyanya's memory usage is not shown, as it is an online algorithm that requires no index, aside from a navigation mesh with a negligible memory footprint. Although EHL* requires nearly twice the preprocessing time compared to EHL-1 and significantly more preprocessing time than the other competitors, the overall build time remains reasonable and practical for most application needs.

Experiment 2: Memory-Runtime Tradeoff

Next, we discuss the memory-runtime tradeoff for each algorithm. Table 5 also shows the query runtime (in μs) for both the unknown and clustered queries. EPS and Polyanya offer lower memory usage but are 1-2 orders of magnitude slower than EHL and EHL*. EHL* reduces memory usage of EHL-1 from 80% to 20% with minimal impact on its runtime, but further reductions to 10% and 5% lead to noticeable increases in query time due to significantly larger merged regions leading to slower runtimes. Comparing with EHL on unknown query distributions, EHL*-80 and EHL*-60 use less memory than EHL-1 while keeping runtimes competitive, and EHL*-40 and EHL*-20, while similar in memory usage to EHL-2 and EHL-4, deliver better runtimes, thus outperforming EHL.

[6]https://github.com/goldi1027/EHL-Star

Dist		EHL* (Known)			EHL* (Unknown)			Competitors		
		80%	40%	20%	80%	40%	20%	EHL-1	EHL-2	EHL-4
100%	C-2	1.30	1.33	1.34	1.45	1.56	1.79	1.30	1.69	2.35
	C-4	1.44	1.48	1.50	1.58	1.76	1.87	1.45	1.79	2.50
	C-8	1.53	1.48	1.64	1.74	1.82	2.04	1.51	1.92	2.64
80%	C-2	1.48	1.54	1.64	1.54	1.65	1.75	1.38	1.75	2.41
	C-4	1.50	1.55	1.86	1.71	1.82	1.89	1.47	1.83	2.53
	C-8	1.63	1.73	2.03	1.75	1.87	1.97	1.54	1.91	2.63
50%	C-2	1.59	1.67	1.82	1.66	1.76	1.84	1.47	1.82	2.48
	C-4	1.68	1.80	2.34	1.70	1.81	1.95	1.52	1.87	2.53
	C-8	1.70	1.87	2.43	1.72	1.87	2.01	1.55	1.92	2.61
20%	C-2	1.75	1.85	2.20	1.87	1.95	1.98	1.53	1.89	2.57
	C-4	1.75	1.88	2.56	1.72	1.86	2.04	1.56	1.91	2.58
	C-8	1.70	1.96	3.09	1.79	1.94	2.08	1.57	1.94	2.62

Table 6: Running time comparison between EHL* and competitors for mixed distribution scenarios for DA benchmark.

When the query distribution is known in advance, EHL* leverages this information to significantly improve the performance. With a small number of cluster regions (e.g., Cluster-2 and Cluster-4), EHL* can reduce the memory budget from 80% to 5% while maintaining similar query runtimes, making the memory reduction nearly cost-free. Compared to EHL, EHL* remains competitive with EHL-1 across all memory budgets and consistently outperforms EHL-2 and EHL-4, demonstrating a superior memory-runtime tradeoff. As the number of cluster regions increases (e.g., Cluster-8), memory reduction becomes more challenging, leading to a more noticeable increase in query runtime. Nevertheless, the performance remains better than EHL for Cluster-8, e.g., when the memory budget is reduced to 5%, EHL*-5 still outperforms EHL-4 which consumes 3-5 times more memory.

Figure 5 illustrates how EHL* merges regions when the memory budget is set to 5%. The merged regions are displayed in different colors for various query sets on the arena map from the DA benchmark. When the query distribution is unknown, EHL* combines grid cells into regions of approximately equal size to efficiently handle queries that may arise at any random position. However, when the query distribution is known, EHL* prioritizes merging cells outside the cluster regions, leaving more granular grid cells within clusters to efficiently answer queries. This behaviour becomes clearer as the number of cluster regions increases. When the number of clusters is large (e.g., Cluster-8), merging cells outside the clusters may not satisfy the memory budget, forcing some merging within the clusters which leads to slower runtimes for the queries within the clusters.

Experiment 3: Query Distribution Deviations

In this section, we analyze the performance of EHL* for queries that diverge from the predicted distribution. We construct EHL* based on the assumed Cluster-x distribution,

but only $y\%$ of the queries adhere to this distribution, while the remaining $(100 - y)\%$ are generated randomly. We vary y from 100 to 20. Table 6 shows the average runtime (in μs) for the three cluster query sets in the DA benchmark. We compare our approaches, EHL* (known), which tries to exploit query distribution, and EHL* (unknown), which has no information on query distribution, with EHL-1, EHL-2, and EHL-4. As expected, when a larger number of queries deviate from the predicted distribution (i.e., smaller $y\%$), the performance of EHL* (known) drops significantly, especially when the budget is 20% of EHL-1 (although still competitive compared to EHL-4). This occurs because EHL* (known) tends to merge grid cells outside the cluster region when memory budget is constrained. As a result, when queries deviate from the cluster regions, larger regions with more vialabels are required to answer the queries, leading to slower query runtimes. In contrast, EHL* (unknown) and EHL remain stable as the number of deviated queries increases since they do not depend on pre-existing query distributions to build index. When most test queries follow the predicted distributions (i.e., 100% and 80%), EHL* (known) outperforms EHL* (unknown) and EHL. However, when more queries deviate from the prediction (i.e., 50% and 20%), EHL* (known) performs worse than both approaches.

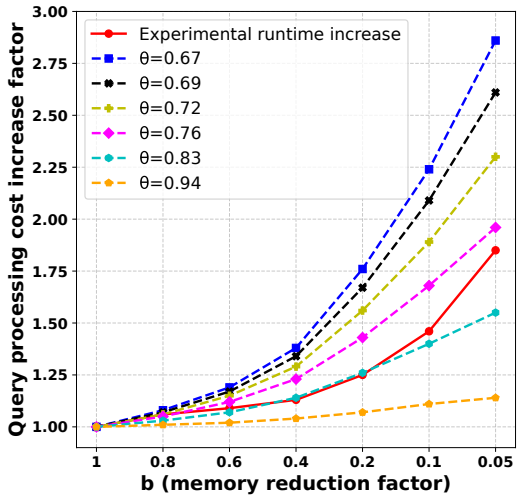


Figure 6: Comparing empirical performance with theoretical analysis for different values of b and θ

Experiment 4: Empirical Insights into Theoretical Analysis

In this section, we perform experiments to gain insights into the empirical value of Jaccard similarity for merging regions at different levels and validate our theoretical analysis by comparing it with the actual query performance.

Empirical value of θ. We conduct experiments on the DA benchmark for the case where the query distribution is unknown using $b = 0.05$, i.e., EHL* reduces the memory to 5% of the memory consumed by EHL. For all merges at each level, we compute and report average Jaccard similarity between the regions merged at that level. For $b = 0.05$, the highest level of merge is at level 6 (e.g., each region contains $2^5 = 32$ cells). Table 7 shows the average Jaccard similarity for each merge level. It can be seen that the average Jaccard similarity decreases as the region size increases. As remarked earlier, our analysis assumes the worst-case and considers θ to be the Jaccard similarity between regions at the highest level. While the average Jaccard similarity decreases as the region sizes increase, please note that the similarity is still quite high among the regions being merged. Furthermore, there are significantly larger number of merges at lower levels than at higher levels which also contributes to effective compression while maintaining comparable query cost.

Merge level	1	2	3	4	5	6
Jaccard Sim.	0.94	0.83	0.76	0.72	0.69	0.67

Table 7: Average Jaccard similarity across merge levels

Query cost. Our analysis demonstrates that the query cost of EHL* is $\frac{2^i}{(1+\theta)^i}$ times that of EHL. To validate this, we run experiments on the DA benchmark with an unknown query distribution, varying the value of b from 1 to 0.05, consistent with our main experiments.

Figure 6 presents the results, where the x-axis represents the memory reduction factor b, and the y-axis indicates the relative increase in runtime of EHL* compared to EHL, specifically the ratio r^*/r, where r^* and r denote the runtime of EHL* and EHL, respectively. The figure illustrates this runtime increase for different values of θ as listed in Table 7 (see dashed lines). The table also shows the actual performance differences observed in our experiments (see the solid line). As anticipated, the worst-case scenario for EHL* ($\theta = 0.67$) overestimates the query cost . On the other hand, using the Jaccard similarity for level 1 merges ($\theta = 0.94$) underestimates the runtime increase as it assumes all merged regions have such high similarity. In contrast, the values $\theta = 0.76$ and $\theta = 0.83$ align more closely with the experimental results, as they better account for variations in Jaccard similarity across different merge levels. Notably, the average of θ values for the six levels is 0.77, which produces a trend closely resembling that of $\theta = 0.76$.

Conclusion

We propose EHL*, an improved version of EHL that offers the flexibility to build the index within a specified memory budget while optimizing query runtime. Experiments show that EHL* reduces memory usage while maintaining competitive query runtimes. Additionally, when query distributions are known in advance, EHL* leverages this information to build an index that outperforms EHL in both memory efficiency and query runtime. Future work includes exploring arbitrarily shaped and ML-based regions for further improvement.

Acknowledgements

Muhammad Aamir Cheema is supported by Australian Research Council 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.

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.

Du, J.; Shen, B.; and Cheema, M. A. 2023. Ultrafast Euclidean Shortest Path Computation using Hub Labeling. In *AAAI*.

Funke, S.; Koch, D.; Proissl, C.; Schneewind, A.; Weiß, A.; and Weitbrecht, F. 2024. Scalable Ultrafast Almost-optimal Euclidean Shortest Paths. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, 6716–6723.

Funke, S.; Koch, D.; Proissl, C.; Staib, C.; and Weitbrecht, F. 2025. Fast and Stronger Lower Bounds for Planar Euclidean Shortest Paths. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-25. International Joint Conferences on Artificial Intelligence Organization. Main Track*.

Koch, D.; and Funke, S. 2025. Guiding the Search for the Euclidean Shortest Path Problem. In *Proceedings of the International Symposium on Combinatorial Search*, volume 18, 191–195.

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.

Sheng, Y.; Cao, X.; Fang, Y.; Zhao, K.; Qi, J.; Cong, G.; and Zhang, W. 2023. Wisk: A workload-aware learned index for spatial keyword queries. *Proceedings of the ACM on Management of Data*, 1(2): 1–27.

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 *Motion in Games - 5th International Conference, MIG 2012, Rennes, France, November 15-17, 2012. Proceedings*, volume 7660 of *Lecture Notes in Computer Science*, 1–6. Springer.

Tzoumas, K.; Yiu, M. L.; and Jensen, C. S. 2009. Workload-aware indexing of continuously moving objects. *Proceedings of the VLDB Endowment*, 2(1): 1186–1197.