Key-Interval A* and Automatic Subspace Decomposition

Anonymous submission

Abstract

We present two novel contributions to grid-based path planning: Key-Interval A* and Automatic Subspace Decomposition. Key-Interval A* introduces the concepts of intervals and key points, which are extracted through horizontal and vertical scans of the environment. By identifying key intervals and evaluating key points, the algorithm significantly reduces the search space while preserving path optimality. Building upon the properties of intervals, we further propose an Automatic Subspace Decomposition method that partitions the environment into a set of connected subspaces by analyzing the connectivity between intervals. The resulting hierarchical structure is well-suited for integration with hierarchical navigation algorithms. Experimental results demonstrate that Key-Interval A* achieves substantial improvements in computational efficiency without sacrificing path quality.

Introduction

Path planning is a fundamental problem in robotics, game AI, and autonomous systems. In grid-based environments, A* (Hart, Nilsson, and Raphael 1968) remains the cornerstone algorithm for computing optimal and complete paths. However, as the size and complexity of environments increase, A* and its derivatives face challenges in computational cost and memory usage.

To address these challenges, various acceleration techniques have been proposed. Jump Point Search (JPS) (Harabor and Grastien 2011) exploits grid symmetries to reduce node expansions but is restricted to uniform-cost grids. Anya (Harabor and Grastien 2013) offers optimal any-angle paths by operating on line-of-sight intervals, but requires sophisticated geometric reasoning. Hierarchical approaches such as HPA* (Botea, Müller, and Schaeffer 2004) and Block A* (Yap et al. 2011) abstract the map into fixed-size clusters to improve planning efficiency. Subgoal Graphs (Uras, Koenig, and Hernandez 2013) preprocess the map into a graph of key subgoals placed at obstacle corners, find a high-level path over subgoals and refines it into a complete low-level path. Compressed Path Databases (CPD) (Botea 2011) precompute all-pairs optimal paths to eliminate runtime search, at the cost of significant memory overhead.

However, many of these methods either rely on uniform cost assumptions, are restricted to 8-connected grids, or require extensive preprocessing and storage. In this work, we propose two novel contributions that exploit the underlying map geometry to accelerate pathfinding while avoiding the limitations above. Our methods support diverse grid connectivity patterns and require only minimal preprocessing:

- **Key-Interval A*** identifies structurally significant *key intervals* and *key points* through interval scanning. By restricting search to key intervals, the algorithm substantially reduces the search space.
- **Automatic Subspace Decomposition** partitions the map into a hierarchy of connected subspaces based on interval connectivity, enabling integration with hierarchical planning pipelines without relying on fixed-size clustering.

Our experiments show that the proposed methods achieve considerable speed-ups in planning without sacrificing solution quality, making them practical for large-scale grid-based navigation tasks.

Preliminaries and Notation

We consider a grid-based pathfinding problem defined on a finite two-dimensional map G of size $M \times N$. Each cell $(x, y) \in \mathbb{Z}^2$ represents a vertex, where x denotes the row index (increasing downward) and y denotes the column index (increasing to the right). The origin $(0, 0)$ corresponds to the top-left corner of the grid. A cell is either *traversable* (free) or *non-traversable* (occupied).

Let $V \subseteq G$ denote the set of all traversable vertices. Two vertices $(x_1, y_1), (x_2, y_2) \in V$ are considered adjacent under the 4-connectivity model if and only if $|x_1 - x_2| + |y_1 - y_2| = 1$. Although the proposed Key-Interval A* algorithm can be extended to support any-angle pathfinding, in this work we restrict our attention to a *4-connected grid* for clarity and simplicity.

The input to the pathfinding problem consists of a grid map G, a start vertex $s \in V$, and a target vertex $t \in V$. The objective is to compute a path $\pi = \langle v_0, v_1, \ldots, v_k \rangle$ such that $v_0 = s$, $v_k = t$, each $v_i \in V$, and every consecutive pair (v_i, v_{i+1}) is adjacent under the 4-connected rule.

We use dot notation to access components of structured objects. For example, for a vertex $v = (x, y)$, we write $v.x = x$ and $v.y = y$. Similarly, for a tuple $T = \langle a, b \rangle$, $T.a$ and $T.b$ refer to its components a and b.

Preprocess

To enable efficient pathfinding, we conduct a one-time preprocessing phase on the input grid map. This phase identifies intervals, extracts critical structural change points (termed *key points*), and designates the intervals containing these key points as *Key Intervals* for search. Each Key Interval stores both vertical and horizontal key points, and maintains a list of neighboring Key Intervals that can be reached via direct transitions. These components form the structural backbone of the Key-Interval A* algorithm.

Interval

Definition (Interval) An **interval** is defined as a maximal contiguous sequence of traversable grid cells along a single row (for horizontal intervals) or column (for vertical intervals), bounded by obstacles or the map boundary.

A vertical interval on column y is represented as:

$$I = \langle y, x_{\text{start}}, x_{\text{end}}, T_{\text{start}}, T_{\text{end}} \rangle, \tag{1}$$

where $\forall x \in [x_{\text{start}}, x_{\text{end}}]$, the cell (x, y) is passable, and both $(x_{\text{start}} - 1, y)$ and $(x_{\text{end}} + 1, y)$ are either obstacles or out of bounds. The cells (x_{start}, y) and (x_{end}, y) are referred to as the start and end vertices of the interval I.

Similarly, a horizontal interval on row x is represented as:

$$I = \langle x, y_{\text{start}}, y_{\text{end}}, T_{\text{start}}, T_{\text{end}} \rangle, \tag{2}$$

where $\forall y \in [y_{\text{start}}, y_{\text{end}}]$, the cell (x, y) is passable, and both $(x, y_{\text{start}} - 1)$ and $(x, y_{\text{end}} + 1)$ are either obstacles or out of bounds. The cells (x, y_{start}) and (x, y_{end}) are referred to as the start and end vertices of the interval I.

T_{s} and T_{e} are temporary variables introduced for the scanning phase and will be explained later. While the full interval representation includes a trend component, we occasionally omit it and use the shorthand form $I = \langle x, y_s, y_e \rangle$ or $I_k = \langle x_k, y_{s_k}, y_{e_k} \rangle$ with a subscript when the trend is not relevant to the context.

Key-Interval A* can be implemented using either vertical or horizontal intervals. Due to symmetry, we describe only the vertical case; the horizontal variant follows analogously.

Definition (Interval Connectivity) Given a vertical interval $I_k = \langle y_k, x_{s_k}, x_{e_k} \rangle$, we define its set of **direct neighbors** as:

$$\mathcal{N}_{\text{dir}}(I_k) = \{ I_j \mid |y_k - y_j| = 1 \wedge [x_{s_k}, x_{e_k}] \cap [x_{s_j}, x_{e_j}] \neq \emptyset \}$$

If one vertical interval is a direct neighbor of another, we say that the two intervals are *connected*. Specifically, if $y_j = y_k - 1$, then I_j is referred to as the *left direct neighbor* of I_k; similarly, if $y_j = y_k + 1$, then I_j is the *right direct neighbor* of I_k.

Key Point Extraction

We extract a set of key points designed to capture structural changes in the layout of intervals. These points are identified by scanning the grid and analyzing how interval boundaries evolve across adjacent columns and rows. Key points are located at positions where the reachable area undergoes a contraction followed by an expansion, indicating a potential decision point during the search process.

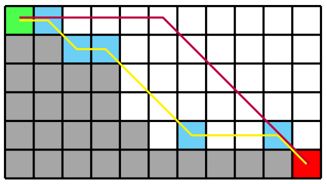

Figure 1: Example of Preprocessing

Vertical Scan In the vertical scan, we iterate over columns y from left to right. For each vertical interval $I = \langle y, x_{\text{s}}, x_{\text{e}}, T_{\text{s}}, T_{\text{e}} \rangle$ in column y, let S^l denote the set of vertical intervals in column $y - 1$ that are connected to I. Then, we determine the trend of the start and end vertices of I as follows:

Let

$$I^l_{\text{min}} := \arg\min_{I^l \in S} I^l.x_s$$

Then T_s is computed as:

$$T_s = \begin{cases} \text{UNCHANGED} & \text{if } S = \emptyset \\ \text{DECREASING} & \text{if } x_s > I_{\text{min}}.x_s \\ \text{INCREASING} & \text{if } x_s < I_{\text{min}}.x_s \\ I_{\text{min}}.T_s & \text{otherwise} \end{cases}$$

Let

$$I^l_{\text{max}} := \arg\max_{I^l \in S} I^l.x_e$$

Then T_e is computed as:

$$T_e = \begin{cases} \text{UNCHANGED} & \text{if } S = \emptyset \\ \text{INCREASING} & \text{if } x_e > I_{\text{max}}.x_e \\ \text{DECREASING} & \text{if } x_e < I_{\text{max}}.x_e \\ I_{\text{max}}.T_e & \text{otherwise} \end{cases}$$

Horizontal Scan The horizontal scan follows the same principle as the vertical scan but processes each row x from top to bottom. For each horizontal interval in row x, we compare it with connected intervals in the previous row $x - 1$ to detect trend changes in the left and right boundaries.

Definition (Key Point) After the scanning phase, we identify *key points* that reflect structural changes across adjacent columns. Let $I = \langle y, x_s, x_e, T_s, T_e \rangle$ be a vertical interval in column y, and let S^r be the set of vertical intervals in the next column $y + 1$ that are directly connected to I. Define:

$$I^r_{\text{min}} := \arg\min_{I' \in S^r} I'.x_s, \quad I^r_{\text{max}} := \arg\max_{I' \in S^r} I'.x_e$$

A vertex (x, y) is defined as a *key point* if either of the following conditions holds:

- $T_s = \text{DECREASING}$ and $I^r_{\text{min}}.T_s = \text{INCREASING}$, in which case (x_s, y) is a key point.
- $T_e = \text{DECREASING}$ and $I^r_{\text{max}}.T_e = \text{INCREASING}$, in which case (x_e, y) is a key point.

All vertical intervals are examined for the above conditions to extract vertical key points. A similar procedure is applied to horizontal intervals to identify horizontal key points. Each detected key point (x, y) is associated with the corresponding vertical interval it lies within; that is, it is stored with the interval $I = \langle y, x_s, x_e, T_s, T_e \rangle$ such that $x \in [x_s, x_e]$ and $y = I.y$.

Key Interval

Definition (Key Interval) A *key interval* is defined as a vertical interval that contains at least one key point, either from the vertical or horizontal scan. Formally, a key interval K is represented as:

$$K = \langle I, P_h, P_v, N, T \rangle,$$

where:

- I is the underlying vertical interval.
- P_h is the set of key points obtained from the horizontal scan.
- P_v is the set of key points obtained from the vertical scan. Since a vertical interval has exactly one start and one end vertex, it can contain at most one UP key point and one DOWN key point. We use P_v.up and P_v.down to refer to them, respectively.
- N is the set of neighboring key intervals directly connected to K.
- T is a set of *transition vertices*, where each $v \in T$ is a vertex associated with a pair of neighboring intervals $(K_i, K_j) \in N \times N$, such that v enables direct traversal between K_i and K_j. This set may be empty.

A key interval must contain at least one key point, i.e., $P_h \neq \emptyset$ or $P_v \neq \emptyset$.

Theorem 1. *If a vertical interval has more than one left neighbor or more than one right neighbor, then this interval must be a key interval.*

Proof. Without loss of generality, we consider the case where a vertical interval has more than one left neighbor; the case of multiple right neighbors follows by symmetry.

Let $I = \langle y+1, x_s, x_e \rangle$ be a vertical interval with n left neighbors in column y, denoted as

$$I_1 = \langle y, x_s^{(1)}, x_e^{(1)} \rangle, \ldots, I_n = \langle y, x_s^{(n)}, x_e^{(n)} \rangle,$$

where $x_s^{(1)} < x_s^{(2)} < \cdots < x_s^{(n)}$. Consider any two adjacent intervals I_k and I_{k+1}. Since there is no vertical interval between I_k and I_{k+1}, all cells in $[x_e^{(k)}, x_s^{(k+1)}]$ in column y must be occupied by obstacles.

Now consider the horizontal scanning process. During the scan from row $x_e^{(k)} - 1$ to $x_s^{(k+1)} + 1$:

1. At position $x_e^{(k)} - 1$, there exists a horizontal interval $H_1 = \langle x_e^{(k)} - 1, y_s^{(1)}, y_e^{(1)} \rangle$, where $y_s^{(1)} \leq y$ and $y_e^{(1)} \geq y+1$. This is due to the existence of I_k and I.
2. At position $x_e^{(k)}$, there exists a horizontal interval $H_2 = \langle x_e^{(k)}, y_s^{(2)}, y_e^{(2)} \rangle$, where $y_s^{(2)} = y+1$ and $y_e^{(2)} \geq y+1$. According to the scanning rules, the end trend of H_2 is marked as DECREASING.
3. During the scan from $x_e^{(k)}$ to $x_s^{(k+1)}$, all horizontal intervals that contains vertex with column $y+1$ have their start at row $y+1$, so their end trends inherit the DECREASING trend.
4. At position $x_s^{(k+1)} + 1$, there exists a horizontal interval $H_3 = \langle x_s^{(k+1)} + 1, y_s^{(3)}, y_e^{(3)} \rangle$, where $y_s^{(3)} \leq y$. According to the scanning rules, the end trend of H_3 is marked as INCREASING.

Thus, at position $x_s^{(k+1)}$, the end trend changes from DECREASING to INCREASING, indicating that point $(x_s^{(k+1)}, y+1)$ is a key point. Since this key point lies within the interval I, we conclude that I must be a key interval.

Hence, if a vertical interval has more than one left neighbor or more than one right neighbor, it must be a key interval. □

Key Interval Neighbors Given a key interval I_k, for each direct neighbor I_d of I_k, initialize $I_{\text{curr}} \leftarrow I_d$. Its neighbors are determined as follows:

1. If I_{curr} is a key interval, add it to the neighbor set $\mathcal{N}(I_k)$.
2. Otherwise, iterate through $\mathcal{N}_{\text{dir}}(I_{\text{curr}})$. If there exists an interval I such that $I.y - I_{\text{curr}}.y = I_{\text{curr}}.y - I_k.y$, then update $I_{\text{curr}} \leftarrow I$ and go to step 1. If no such I exists, no neighbor is found in this direction.

If the procedure yields a key interval, it is recorded as the neighbor of I_k in the direction of the direct neighbor I_d. According to Theorem 1, for each interval I_{curr}, either it is itself a key interval, or there exists at most one direct neighbor I such that $I.y - I_{\text{curr}}.y = I_{\text{curr}}.y - I_k.y$. Therefore, the above procedure guarantees that in each direction of a direct neighbor, either no neighbor is found or a unique one is determined.

Transition Vertex Let I_k be a key interval. Consider any pair of direct neighbors $I_a, I_b \in \mathcal{N}_{\text{dir}}(I_k)$ such that both lie on the same side of I_k, i.e., $I_k.y - I_a.y = I_k.y - I_b.y$.

Suppose there exist key intervals I_{N_a} and I_{N_b} such that I_{N_a} is a neighbor of I_k in the direction of I_a, and I_{N_b} is a neighbor in the direction of I_b. Then, any key point $p = (x, y) \in I_k$ satisfying:

$$\max(I_a.x_e, I_b.x_e) \leq x \leq \min(I_a.x_s, I_b.x_s)$$

is defined as a *transition vertex* between the key intervals I_{N_a} and I_{N_b}, via I_k. If the number of direct neighbors on one side of I_k is at most one, or if I_k has no key interval neighbor in the direction of either I_a or I_b, then a transition vertex between I_{N_a} and I_{N_b} does not exist.

According to Theorem 1, such a key point p must exist in I_k, and is identified during the horizontal scan phase.

Key-Interval A*

Key-Interval A* is a pathfinding algorithm that operates on a higher-level abstraction of the environment, replacing individual grid cells with structurally significant regions called key intervals. Unlike traditional A* which expands individual cells, Key-Interval A* navigates through a reduced search space composed of key intervals, leveraging precomputed structural information—key points and neighbor relations.

Search Node Structure

Each node in the search frontier represents a key interval and maintains the following attributes:

- **K**: The key interval currently represented by this node.
- **g**: The cost from the start vertex to the current interval.
- **f**: The total estimated cost, computed as $f = g + h$.
- **parent**: The parent key interval from which the current key interval was reached.
- **waypoints**: An ordered list of vertices that the final path must traverse.
- **up, down**: Auxiliary key points used to maintain vertical visibility constraints.
- **isTarget**: A Boolean flag indicating whether the target has been reached.

Algorithm Description

The pseudocode for the Key-Interval A* algorithm is presented in Algorithm 1. For clarity, several low-level implementation details are omitted, including the internal representation of preprocessed key intervals and standard A* mechanisms such as g-value updates, open list operations, and duplicate detection. These aspects follow conventional A* implementations and are not central to the novel contributions of this work.

The Key-Interval A* algorithm begins by identifying the key intervals that contain or are closest to the start and target vertices, referred to as the *start intervals* and *target intervals*, respectively. This identification is performed using the same recursive neighbor traversal procedure introduced during the construction of key interval neighbors.

While locating the nearest key intervals for the start vertex, if the current interval I_{curr} directly contains the target vertex, the start interval set S is assigned to be empty, and a trivial path from the start to the target is immediately constructed (Lines 1–5). Otherwise, for each identified start key interval, a corresponding search node is initialized. Each node stores the current key interval, the cost-to-come g, the estimated total cost $f = g+h$, and uses the start vertex as its initial waypoint. All initial nodes are inserted into a priority queue ordered by increasing f-value (Lines 7–9).

In each iteration, the node with the lowest f-value is dequeued (Line 12). If it is marked as a target node, the algorithm terminates and returns the reconstructed path (Lines 13–15). If the current node corresponds to a key interval that intersects with the target intervals, a special transition step is applied to connect the interval with the exact target vertex (Lines 16–19). Otherwise, the algorithm expands all neighbor key intervals of the current interval (Lines 20–25), generating successor nodes with updated cost estimates, waypoints, and visibility constraints.

This process repeats until either a target node is reached or the open list becomes empty (Line 27).

Further details on neighbor expansion, cost evaluation, and waypoint propagation are described later.

Algorithm 1: Key Interval A*

```
Input: Start vertex $s$, Target vertex $t$, Preprocess data
Output: Path from $s$ to $t$
1  $S \leftarrow$ FindNearestKeyIntervals($s$);
2  $T \leftarrow$ FindNearestKeyIntervals($t$);
3  if $S = \emptyset$ then
4      return ConstructPath({s,g});
5  end
6  Initialize priority queue $Q$ ordered by $f$-value;
7  for each $I \in S$ do
8      Initialize search node with key interval $I$;
9      Insert the node into $Q$;
10 end
11 while $Q$ is not empty do
12     $current \leftarrow Q$.pop();
13     if $current.isTarget$ then
14         return ConstructPath($current.waypoints$);
15     end
16     if $current.K \in T$ then
17         HandleTargetKeyInterval($current, t, Q$);
18         continue;
19     end
20     foreach $K_n \in current.K.N$ do
21         if $K_n = current.parent$ then
22             continue;
23         end
24         ExpandNeighbor($current, K_n, t, Q$);
25     end
26 end
27 return $\emptyset$;
```

Calculation of g and h Values

For each search node, The g-value is defined as the sum of:

- The Manhattan distances between all consecutive waypoints, and
- The Manhattan distance from the final waypoint to a temporary evaluation vertex v_{eval}.

The h-value is defined as the Manhattan distance from v_{eval} to the target vertex t.

The evaluation vertex v_{eval} is selected based on the position of the target relative to the current interval $K.I = \langle y, x_s, x_e \rangle$:

- If $t.x \geq x_e$, then $v_{\text{eval}} = (x_e, y)$.
- If $t.x \leq x_s$, then $v_{\text{eval}} = (x_s, y)$.
- Otherwise, $v_{\text{eval}} = (t.x, y)$.

Expanding Neighbors

When expanding a neighboring key interval K which is not a target interval, the algorithm creates a successor node n from the current node c in three main steps:

Waypoint Propagation The successor node inherits the waypoint list from the current node, i.e., n.waypoints $\leftarrow$ c.waypoints. Let (x_l, y_l) denote the last waypoint in c.waypoints.

If a transition point $p \in T(K_c, K)$ exists between the current interval K_c and neighbor K, it is appended to n.waypoints. Otherwise, additional waypoints are inserted based on the visibility constraints derived from key points:

- If $K.P_v$.up exists and both $K.P_v$.up.x and x_l are $\leq$ c.down.x, then c.down is appended.
- If $K.P_v$.down exists and both $K.P_v$.down.x and x_l are $\geq$ c.up.x, then $K.P_v$.down is appended.

Up/Down Assignment If any waypoint was added in the previous step, the successor node directly inherits:

$$n.\text{up} \leftarrow K.P_v.\text{up}, \quad n.\text{down} \leftarrow K.P_v.\text{down}.$$

Otherwise, up and down vertices are selectively assigned as follows:

- If $K.P_v$.up exists and $K.P_v$.up.x $\leq$ c.up.x, then set n.up $\leftarrow$ $K.P_v$.up.
- If $K.P_v$.down exists and $K.P_v$.down.x $\geq$ c.down.x, then set n.down $\leftarrow$ $K.P_v$.down.

The g- and h-values for the new node are computed based on the updated waypoint list and $up, down$ vertex, as previously described. The $parent$ of the new node is $c.K$ and the $isTarget$ is *false*.

Target Interval Processing

When the neighbor key interval K being expanded is one of the target intervals, the algorithm attempts to directly connect the current path to the target vertex $t = (x_t, y_t)$ by appending a final transition. Let c be the current node, and (x_l, y_l) denote the last waypoint in c.waypoints. A new node n is created with its waypoint list initialized as n.waypoints $\leftarrow$ c.waypoints.

Depending on the vertical relation between the last waypoint and the target, additional intermediate points are inserted as follows:

- If $x_l \geq c$.up.x and $x_t \geq c$.up.x, append c.up and t to n.waypoints.
- Else if $x_l \leq c$.down.x and $x_t \leq c$.down.x, append c.down and t to n.waypoints.

After updating the waypoint list, the successor node n is finalized:

- Compute its g- and f-values using the full waypoint sequence.
- Set n.isTarget $\leftarrow$ *true*.

This node is then inserted into the open list for potential expansion.

Path Construction

Once a target node is found, or a direct connection between the start and target vertices is identified during the nearest key interval detection, the algorithm proceeds to construct the final path based on the recorded list of waypoints.

Let $\pi = \langle w_0, w_1, \ldots, w_k \rangle$ denote the ordered sequence of waypoints extracted from the terminal search node. The complete path is constructed by concatenating path segments between each consecutive pair (w_i, w_{i+1}) in the list.

Each segment is computed using a direction-restricted A* search. Given two waypoints $w_i = (x_1, y_1)$ and $w_{i+1} = (x_2, y_2)$, the algorithm restricts the A* expansion to grid neighbors that reduce the Manhattan distance to the goal. Specifically, the allowed move directions are chosen based on the relative positions of the endpoints:

- If $x_2 > x_1$, the algorithm allows moves in the $(1, 0)$ direction.
- If $x_2 < x_1$, it allows $(\text{-}1, 0)$.
- If $y_2 > y_1$, it allows $(0, 1)$.
- If $y_2 < y_1$, it allows $(0, \text{-}1)$.

Since the search is performed between waypoints selected based on key interval connectivity, the existence of a valid path with Manhattan cost $\|w_i - w_{i+1}\|_1$ is guaranteed. The use of directional constraints significantly reduces the number of expanded nodes, as the search avoids exploring irrelevant directions.

The final path is obtained by concatenating all segment paths in order, resulting in a complete, obstacle-free, and cost-optimal trajectory from the start to the target.

Automatic Subspace Decomposition

As demonstrated in the proof of Theorem 1 in the preprocessing stage, when a vertical interval contains a horizontal key point—for example, a LEFT key point—there must exist at least two vertical intervals in the adjacent right column that are separated by obstacles. This observation reveals a natural partitioning of the environment into spatially segmented regions. Based on this property, we propose a method for automatically segmenting the map into subspaces.

In this work, we present the segmentation method using vertical intervals; a symmetric method based on horizontal intervals can be defined analogously. The procedure consists of three main steps:

1. As in the preprocessing phase, we first identify and store all vertical intervals in the map.
2. We perform a horizontal scan to locate all horizontal key points.
3. We execute the subspace segmentation process by iterating through each vertical interval.

During the final step, we skip any vertical interval that has already been visited. For each unvisited vertical interval I, we initialize a direction set S_{dir} and a subspace set S_{sub}. If I has only one left neighbor, we add it to S_{dir}; the same is done for the right neighbor. We then initialize S_{sub} with $\{I\}$.

For each directional neighbor $I_d \in S_{\text{dir}}$, we initialize $I_{\text{curr}} \leftarrow I_d$ and proceed as follows:

1. If I_{curr} is a key interval, we terminate the expansion in this direction.
2. Otherwise, we add I_{curr} to S_{sub} and mark it as visited. Then, we iterate through its directional neighbors $\mathcal{N}_{\text{dir}}(I_{\text{curr}})$. If there exists an interval I such that $I.y - I_{\text{curr}}.y = I_{\text{curr}}.y - I_k.y$, we update $I_{\text{curr}} \leftarrow I$ and return to Step 1. If no such I exists, the process in this direction terminates.

By repeating this procedure for all vertical intervals, we obtain a collection of disjoint sets of intervals. Each set corresponds to a subspace of the environment.

Experimental Evaluation

We conduct a comprehensive evaluation to assess the performance of Key-Interval A* in terms of runtime efficiency, search space reduction, and path optimality. To ensure fair and standardized comparison, we adopt the widely-used benchmark suite provided by the MovingAI lab (Stern et al. 2019), which includes grid-based maps and a large set of predefined pathfinding scenarios (start-goal pairs) of varying complexity.

All algorithms are tested on a variety of representative map types from the benchmark, including open spaces, mazes, room-and-corridor structures, and real-game environments. Each map is paired with a set of 500 queries randomly selected from the corresponding scenario file. For each algorithm, we report average runtime, number of expanded nodes, and final path cost.

We compare our approach against classical A*, Jump Point Search (JPS), and Block A* as baselines. All algorithms are implemented in C++ and executed in a single-threaded environment to ensure consistency.

References

Botea, A. 2011. Ultra-Fast Optimal Pathfinding without Runtime Search. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 7(1): 122–127.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near Optimal Hierarchical Path-Finding. *J. Game Dev.*, 1: 1–30.

Harabor, D.; and Grastien, A. 2011. Online Graph Pruning for Pathfinding On Grid Maps. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1): 1114–1119.

Harabor, D.; and Grastien, A. 2013. An Optimal Any-Angle Pathfinding Algorithm. *Proceedings of the International Conference on Automated Planning and Scheduling*, 23(1): 308–311.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Barták, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Symposium on Combinatorial Search*, 151–159.

Uras, T.; Koenig, S.; and Hernandez, C. 2013. Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids. *Proceedings of the International Conference on Automated Planning and Scheduling*, 23(1): 224–232.

Yap, P.; Burch, N.; Holte, R.; and Schaeffer, J. 2011. Block A*: Database-Driven Search with Applications in Any-Angle Path-Planning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1): 120–125.