Key-Interval A* and Automatic Subspace Decomposition

Anonymous submission

Abstract

We propose two novel techniques for accelerating grid-based path planning: Key-Interval A* and Automatic Subspace Decomposition. Key-Interval A* introduces a higher-level abstraction based on intervals—maximal traversable segments on grid rows or columns—and extracts key points that capture structural changes in interval boundaries through horizontal and vertical scans. By restricting search to these key intervals and navigating via key points, the algorithm reduces the number of node expansions. Building on interval connectivity, Automatic Subspace Decomposition partitions the environment into a hierarchy of connected subspaces without relying on fixed-size clustering, enabling integration with hierarchical planners. Experimental results on standard grid benchmarks demonstrate that our approach significantly improves computational efficiency and produces solutions that are comparable in quality to those of classical A*.

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 and 8-connected grids. Anya (Harabor and Grastien 2013) provides online, optimal any-angle paths by searching over intervals of states (line-of-sight intervals). Hierarchical approaches such as HPA* (Botea, Müller, and Schaeffer 2004) and Block A* (Yap et al. 2011) accelerate grid search by partitioning the map into fixed-size regions—HPA* abstracts entry and exit points on cluster borders, while Block A* further exploits precomputed local distance databases to speed intra-block queries. 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 or mitigating the limitations above. Our methods require only minimal preprocessing and can be extended to support diverse grid connectivity patterns:

- **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 Key-Interval A* achieves substantial speed-ups over classical A*, making it a practical solution for large-scale grid-based navigation tasks.

Preliminaries and Notation

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

Let $V \subseteq G$ denote the set of traversable vertices. Two vertices $(x_1, y_1), (x_2, y_2) \in V$ are 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** can be generalized to support any-angle pathfinding, in this work we focus on the *4-connected* case 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.

Preprocessing

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. 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:

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

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. Cells (x_{start}, y) and (x_{end}, y) are referred to as the start and end vertices of the interval I. As an illustration, Figure 1 shows that $\langle 0, 0, 7 \rangle$ is the sole vertical interval in column 0, whereas column 3 contains exactly two vertical intervals: $\langle 3, 0, 2 \rangle$ and $\langle 3, 5, 7 \rangle$.

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

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

The notations $K_{\text{left}}, K_{\text{right}}, K_{\text{up}}, K_{\text{down}}$ denote the neighboring key intervals, which will be formally introduced in a later section. Although the complete interval representation includes information about these neighbors, we occasionally use a simplified notation, either $I = \langle x, y_s, y_e \rangle$ or $I_k = \langle x_k, y_{s_k}, y_{e_k} \rangle$ with a subscript when the neighbor relationships are not relevant to the discussion.

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:

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

Two vertical intervals are said to be *connected* if one is a direct neighbor of the other. A direct neighbor I_j is classified as a *left direct neighbor* of I_k when $y_j = y_k - 1$, and as a *right direct neighbor* when $y_j = y_k + 1$.

Figure 1 illustrates an example: the interval $\langle 2, 0, 7 \rangle$ has three direct neighbors—$\langle 1, 1, 6 \rangle$, $\langle 3, 0, 2 \rangle$, and $\langle 3, 5, 7 \rangle$. Here, $\langle 1, 1, 6 \rangle$ is the left direct neighbor, while $\langle 3, 0, 2 \rangle$ and $\langle 3, 5, 7 \rangle$ are right direct neighbors.

Key Point Extraction

We extract a set of *key points* to capture structural transitions in the traversable space. A key point marks a location where the free space contracts and then expands across columns or rows, signaling a potential decision point for the search algorithm.

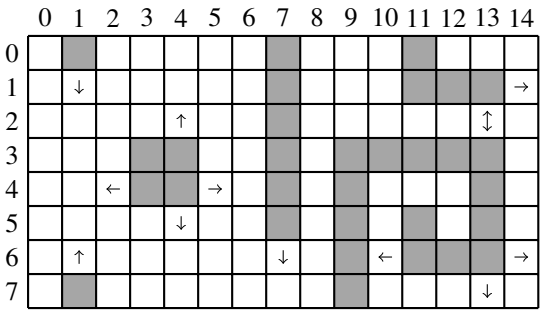

Figure 1: Example of preprocessing. Vertices marked with arrows indicate key points along with their associated directions

Vertical Scan During the vertical scan, we iterate through the columns from left to right. For each vertical interval $I = \langle y, x_{\text{s}}, x_{\text{e}} \rangle$ in column y, let S^l denote the set of left direct neighbors of I. We introduce two temporary trend markers, start trend T_s and end trend T_e, which describe the positional variation of the interval's start and end vertices relative to the previous column. Specifically:

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:

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

In Figure 1, consider the vertical interval $\langle 3, 0, 2 \rangle$. Its I^l_{min} is $\langle 2, 0, 7 \rangle$, where $I^l_{\text{max}}.x_e = 7$. Since x_e of $\langle 3, 0, 2 \rangle$ is 2, which is smaller than $I^l_{\text{max}}.x_e$, the end trend T_e is classified as DECREASING, indicating that the free space is contracting at this position.

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 start and end vertices.

Definition (Key Point) After the scanning phase, we define *key points* as designated vertices that mark local structural transitions in the traversable space. For each vertical

interval $I = \langle y, x_s, x_e \rangle$ in column y, let S^r denote the set of right direct neighbors of I. Define:

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

A vertex (x, y) is classified as a *key point* if it satisfies one of the following conditions:

- $T_s = \text{DECREASING}$ and $I^r_{\text{min}}.T_s = \text{INCREASING}$; in this case, (x_s, y) is marked as a DOWN key point, indicating that the traversable space lies below this vertex.
- $T_e = \text{DECREASING}$ and $I^r_{\text{max}}.T_e = \text{INCREASING}$; in this case, (x_e, y) is marked as an UP key point, indicating that the traversable space lies above this vertex.

All vertical intervals are checked against these criteria to extract vertical key points. A corresponding procedure is applied to horizontal intervals to identify horizontal key points, which are classified as LEFT and RIGHT key points.

Each detected key point (x, y) is associated with the vertical interval $I = \langle y, x_s, x_e \rangle$ containing it, i.e., $x \in [x_s, x_e]$ and $y = I.y$. Figure 1 illustrates all detected key points.

Key Interval

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

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

where each component is defined as follows:

- I is the underlying vertical interval itself. For simplicity, we may occasionally refer to I as the key 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 neighbor key intervals of I defined later.
- T is a set of *transition vertices*, where each $v \in T$ serves as a traversable anchor point linking a pair of neighboring intervals $(I_i, I_j) \in N_{\text{dir}}(I) \times N_{\text{dir}}(I)$. This set may be empty.
- By definition, 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 possesses more than one left direct neighbor or more than one right direct neighbor, it necessarily contains at least one key point and is therefore a key interval.*

Proof. Without loss of generality, we consider the case where a vertical interval has more than one left direct 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 $[x_e^{(k)} + 1, x_s^{(k+1)} - 1]$ in column y must be occupied by obstacles.

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

1. At row $x_e^{(k)}$, there exists a horizontal interval $H_{x_e^{(k)}} = \langle x_e^{(k)}, y_s^{(0)}, y_e^{(0)} \rangle$, where $y_s^{(0)} \leq y$ and $y_e^{(0)} \geq y + 1$. This is due to the existence of I_k and I.
2. At row $x_e^{(k)} + 1$, there exists a horizontal interval $H_{x_e^{(k)}+1} = \langle x_e^{(k)} + 1, y_s^{(1)}, y_e^{(1)} \rangle$, where $y_s^{(1)} = y + 1$ and $y_e^{(1)} \geq y + 1$. According to the scanning rules, the start trend of H_2 is marked as DECREASING.
3. For each subsequent row $x = x_e^{(k)} + i$ (with $i \geq 2$ and up to $x_s^{(k+1)} - x_e^{(k)-1} - 1$), there exists a horizontal interval $H_i = \langle x, y_s^{(i)}, y_e^{(i)} \rangle$ with $y_s^{(i)} = y + 1$. The upper direct neighbor of H_i with the smallest start coordinate y_s must be H_{i-1}, because no other interval spans $y + 1$ and thus has a smaller start coordinate. Therefore, the start trend of H_i inherits the DECREASING trend from H_{i-1}.
4. At row $x_s^{(k+1)}$, let $j = x_s^{(k+1)} - x_e^{(k)}$ there exists a horizontal interval $H_{x_s^{(k+1)}} = \langle x_s^{(k+1)}, y_s^{(j)}, y_e^{(j)} \rangle$, where $y_s^{(j)} \leq y$ and $y_e^{(j)} \geq y + 1$. According to the scanning rules, the start trend of $H_{x_s^{(k+1)}}$ is INCREASING.

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

Key Interval Neighbors Given a key interval I_k, we determine its neighbor set N by expanding through its direct neighbors. For each direct neighbor $I_d \in N_{\text{dir}}(I_k)$, initialize $I_{\text{curr}} \leftarrow I_d$ and create a list L to record intermediate (non-key) intervals visited during the search. The procedure is as follows:

1. If I_{curr} is a key interval, add it to neighbor set N of I_k.
2. Otherwise:
   (a) Append I_{curr} to L.
   (b) For each direct neighbor $I \in N_{\text{dir}}(I_{\text{curr}})$:
   - If $(I.y - I_{\text{curr}}.y)(I_{\text{curr}}.y - I_k.y) > 0$:
     – Set $I_{\text{curr}} \leftarrow I$ and return to Step 1.

   (c) If no such I exists:
   - Terminate the search in the direction of I_d.

If a key interval I_n is discovered through this process, it is designated as the neighbor of I_k in the direction of its originating direct neighbor I_d. Let I_{last} denote the last interval appended to L; we say that I_n is *reached from* I_{last}.

By Theorem 1, for any I_{curr}, either (i) I_{curr} is a key interval, or (ii) there exists at most one $I \in N_{\text{dir}}(I_{\text{curr}})$ satisfying $(I.y - I_{\text{curr}}.y)(I_{\text{curr}}.y - I_k.y) > 0$. This ensures that the procedure produces at most one distinct neighbor in each search direction.

For each non-key interval $I \in L$, its left and right key intervals are assigned as follows:

$$\text{If } I_k.x < I_n.x: \quad I.K_{\text{left}} \leftarrow I_k, \quad I.K_{\text{right}} \leftarrow I_n$$
$$\text{If } I_k.x > I_n.x: \quad I.K_{\text{left}} \leftarrow I_n, \quad I.K_{\text{right}} \leftarrow I_k$$

Note that key intervals themselves do not have left or right key intervals.

Transition Vertex Given a key interval I_k, if I_k has more than one direct neighbor on the left or right side, consider any pair of direct neighbors $I_a, I_b \in N_{\text{dir}}(I_k)$ that are located on the same side of I_k, i.e., $(I_k.y - I_a.y)(I_k.y - I_b.y) > 0$.

A vertex $p = (x, y) \in I_k$ is defined as a *transition vertex* between I_a and I_b via I_k if and only if it satisfies:

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

By Theorem 1, at least one such transition vertex p is guaranteed to exist within I_k and is identified during the horizontal scan phase. Although multiple vertices may satisfy this condition, we store a single representative vertex in the transition mapping $T(I_a, I_b)$.

Key-Interval A*

Key-Interval A* is a pathfinding algorithm that operates on a higher-level abstraction of the environment. Unlike traditional A*, which expands individual cells, Key-Interval A* performs the search over a reduced search space composed of key intervals, leveraging precomputed structural features—namely, 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 represented by the 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.
- **waypoints**: An ordered list of vertices that the final path must traverse.
- **up, down**: Auxiliary key points used to maintain vertical visibility constraints.
- **isTarget**: Boolean flag indicating whether the node corresponds to the target.

Algorithm Description

Algorithm 1 outlines the overall workflow of Key-Interval A*. The algorithm begins by identifying the start and target key intervals, initializing the priority queue, and seeding it with nodes corresponding to all start intervals. The main loop repeatedly extracts the node with the lowest f-value and processes its associated key interval.

Three key subroutines encapsulate the core operations. HANDLETARGETKEYINTERVAL manages the case where the search reaches a target interval and finalizes the waypoint sequence leading to the goal. HANDLENEIGHBOR processes each neighboring key interval of current key interval, generates a corresponding successor node and inserts it into the priority queue. Finally, the algorithm terminates once the goal is reached and the complete waypoint chain is used to construct the final path.

Key-Interval A* maintains a single best g-value for each key interval: whenever multiple paths reach the same interval, only the lowest g-value is stored and used for further expansion.

Further details are provided in the following sections.

Algorithm 1: Key Interval A*

Input: Start vertex s, Target vertex t, Preprocess data
Output: Path from s to t

$S, T \leftarrow$ QueryKeyIntervals(s, t);
if *DirectlyReachable*(S, T, s, t) **then**
  return ConstructPath({s,t});
end
Initialize priority queue Q ordered by f-value;
for $I \in S$ **do**
  Initialize search node with key interval I;
  Insert the node into Q;
end
while Q *is not empty* **do**
  $current \leftarrow Q$.pop();
  if $current$*.isTarget* **then**
    return ConstructPath($current.waypoints$);
  end
  if $current.K \in T$ **then**
    HandleTargetKeyInterval($current, t, Q$);
    continue;
  end
  foreach $K_n \in current.K.N$ **do**
    if $K_n = current$*.parent* **then**
      continue;
    end
    HandleNeighbor($current, K_n, t, Q$);
  end
end
return $\emptyset$;

Key Interval Query

The function QUERYKEYINTERVALS determines the spatial relationship between the start and target vertices. Given a start vertex s and a target vertex t, they are considered *directly reachable* if any of the following conditions hold:

1. s and t lie within the same key interval;
2. The key interval containing s is exactly the left or right key interval of t;
3. The key interval containing t is exactly the left or right key interval of s;
4. Neither s nor t lies within any key interval, and their left key interval and right key interval either (i) both exist and match, (ii) only the left key intervals exist and match, (iii) only the right key intervals exist and match, or (iv) are both absent.

If direct reachability is satisfied, the process terminates immediately and a path is constructed using s and t as the only waypoints. Note that case (iv) in the last condition may also occur when t is actually unreachable from s (e.g., both are located in isolated regions). In such situations, invoking the CONSTRUCTPATH will return an empty path, indicating that the target is unreachable.

If direct reachability does not hold, the function identifies *start intervals* and *target intervals* as follows. If s lies within a key interval, the neighbors of that interval are designated as start intervals; otherwise, the left and right neighbor key intervals of the vertical interval containing s are used. Target intervals are determined based on the position of t: if t is contained within a key interval, that interval becomes the sole target interval; otherwise, the left and right neighbor key intervals of the interval containing t are selected.

For each identified start interval, a corresponding search node is initialized. Each start node records the associated key interval, sets its parent to null, assigns its up key point (if present) as the up vertex, and its down key point (if present) as the $down$ vertex. It then computes the g and h values, initializes the waypoint list with the start vertex and sets $isTarget$ to *false*. All start nodes are subsequently inserted into the open list (priority queue) to begin the search.

Calculation of g and h Values

The g-value of a search node is computed as the cumulative Manhattan distance between all consecutive waypoints along the current path. If the last waypoint lies outside the current key interval I, an auxiliary evaluation vertex v_{eval} is introduced to represent the projected position of the target within I. This vertex is defined as:

- $v_{\text{eval}} = (x_s, y)$ if $t.x \leq x_s$;
- $v_{\text{eval}} = (x_e, y)$ if $t.x \geq x_e$;
- $v_{\text{eval}} = (t.x, y)$ otherwise.

The Manhattan distance from the last waypoint to v_{eval} is then added to the g-value.

The heuristic function estimates the remaining cost h to the target vertex t. If an evaluation vertex v_{eval} is introduced, h is computed from v_{eval} to t using the chosen admissible heuristic (e.g., Manhattan distance). If the last waypoint lies within I, no evaluation vertex is needed, and h is simply computed from the last waypoint to t.

Handling Neighbors

When processing a neighbor key interval K that is not one of the target intervals, the algorithm generates a successor node n from the current node c.

Waypoint Propagation The successor node first inherits the waypoint list from the current node, i.e., n.waypoints $\leftarrow$ c.waypoints.

Let N_{dir1} denote the direct neighbor from which $c.K$ was reached, and N_{dir2} denote the direct neighbor through which $c.K$ expands toward K. If a transition vertex $p \in c.T(N_{\text{dir1}}, N_{\text{dir2}})$ exists, it is appended to n.waypoints.

Otherwise, the algorithm checks whether new waypoints are needed. Let w_l be the last waypoint in c.waypoints, and let v_s and v_e be the start and end vertices of K. The function HANDLEWAYPOINTS(c, n, w_l, v_s, v_e) is invoked to determine if new waypoints should be added (Function 2).

Function 2: HandleWaypoints

Input: Current node c, Neighbor node n, last waypoint (x_w, y_w), vertex (x_1, y_1) , vertex (x_2, y_2)

```
if c.up ≠ null ∧ x_w > c.up.x ∧ x_1 > c.up.x then
    n.waypoints.append(c.up);
    n.waypoints.append((x_1, y_1));
end
else if c.down ≠ null ∧ x_w < c.down.x ∧
   x_2 < c.down.x then
    n.waypoints.append(c.down);
    n.waypoints.append((x_2, y_2));
end
```

If neither a transition vertex nor any new waypoints are required, the waypoint list remains unchanged.

Up/Down Vertex Assignment The values of n.up and n.down are assigned as follows:

- **Case 1:** If any waypoint was appended to the waypoint list in previous waypoint propagation step, both n.up and n.down are set to null.
- **Case 2:** Otherwise, the up and down vertices are updated as follows:
  - **Up vertex:** If both the up vertex of the neighbor interval and c.up exist, set n.up to the vertex with the smaller x-coordinate. If only one exists, assign that vertex. Otherwise, set n.up to null.
  - **Down vertex:** If both the down vertex of the neighbor interval and c.down exist, set n.down to the vertex with the larger x-coordinate. If only one exists, assign that vertex. Otherwise, set n.down to null.

The values of g and h are then computed based on n.waypoints. The parent of node n is set to $c.K$, and its $isTarget$ flag is set to $false$. Finally, n is inserted into the open list.

Target Interval Processing

When the neighbor key interval K belongs to the target intervals T, the algorithm attempts to finalize the waypoint sequence by appending the target vertex t through a final transition. Let c be the current node.

Waypoint Completion A successor node n is created with its waypoint list initialized as n.waypoints $\leftarrow$ c.waypoints. Let dir1 denote the direct neighbor from which $c.K$ was reached, and N_{dir2} denote the direct neighbor through which the target vertex would traverse to K. If a transition vertex $p \in c.T(N_{\text{dir1}}, N_{\text{dir2}})$ exists, both p and t are appended to n.waypoints.

Otherwise, let w_l be the last waypoint in c.waypoints. In this case, additional waypoints are determined by invoking HANDLEWAYPOINTS(c, n, w_l, t, t).

The successor node n is then finalized as follows: its key interval is set to null, and both the up and down vertices are set to null. Its parent is set to $c.K$, and the node is marked as a target node. The g value of n is computed as the accumulated cost along its completed waypoint sequence, while h is set to zero. Finally, the target node n is inserted into the open list for subsequent expanding.

Path Construction

Once a target node is found, or a direct connection between the start and target vertices is identified during the key interval query, the algorithm constructs the final path based on the 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 the shortest path segments between each consecutive pair (w_i, w_{i+1}) in this sequence.

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 A* expansion to only those neighbors that monotonically decrease the Manhattan distance to w_{i+1}. Concretely, the allowed moves are determined by the relative coordinates of the endpoints:

- If $x_2 > x_1$, moves in the $(1, 0)$ direction are allowed.
- If $x_2 < x_1$, moves in the $(-1, 0)$ direction are allowed.
- If $y_2 > y_1$, moves in the $(0, 1)$ direction are allowed.
- If $y_2 < y_1$, moves in the $(0, -1)$ direction are allowed.

Because all waypoints are chosen based on the structural decomposition of the grid (via up and down vertex evaluation), each segment is guaranteed to have a valid Manhattan-optimal path. The directional constraints substantially reduce the number of expanded nodes by preventing exploration of irrelevant directions.

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

Automatic Subspace Decomposition

As demonstrated in the proof of Theorem 1, whenever a vertical interval contains a horizontal key point, there must exist at least two vertical intervals in the adjacent column that are separated by obstacles. This property induces a natural partitioning of the environment into spatially segmented regions. Building on this insight, we introduce an automatic procedure for decomposing the map into subspaces. In this work, we describe the procedure using vertical intervals, while a symmetric variant can be formulated analogously for horizontal intervals.

We begin this stage after completing the scanning phase of preprocessing. We iterate through all vertical intervals, and for each vertical interval I_k, if I_k has not been visited, we perform the following procedure:

Let $N'_{\text{dir}} \leftarrow N_{\text{dir}}(I_k)$ and initialize an empty subspace set $S_{\text{sub}} = \emptyset$. If I_k contains a LEFT key point, remove all right direct neighbors from N'_{dir}; if I_k contains a RIGHT key point, remove all left direct neighbors instead.

For each remaining direct neighbor $I_d \in N'_{\text{dir}}$, initialize $I_{\text{curr}} \leftarrow I_d$ and execute the following steps.

1. **If I_{curr} contains a horizontal key point**
   (a) If I_d is a left direct neighbor of I_k:
      i. If I_{curr} contains a LEFT key point, terminate the process in the direction of I_d.
      ii. If I_{curr} contains a RIGHT key point, add I_{curr} to S_{sub}, mark it as visited, and then terminate the process in the direction of I_d.
   (b) If I_d is a right direct neighbor of I_k:
      i. If I_{curr} contains a LEFT key point, add I_{curr} to S_{sub}, mark it as visited, and then terminate the process in the direction of I_d.
      ii. If I_{curr} contains a RIGHT key point, terminate the process in the direction of I_d.
2. **Otherwise:** Add I_{curr} to S_{sub} and mark it as visited. Then, iterate through its direct neighbors $N_{\text{dir}}(I_{\text{curr}})$. If there exists an interval I such that $(I.y - I_{\text{curr}}.y)(I_{\text{curr}}.y - I_k.y) > 0$, update $I_{\text{curr}} \leftarrow I$ and return to Step 1. If no such I exists, terminate the process in the direction of I_d.

After processing all direct neighbors of interval I_k, all vertices contained in the intervals of S_{sub} collectively define a subgraph of the map. By repeating this procedure for every unvisited vertical interval, we obtain a collection of disjoint interval sets, each corresponding to a distinct subspace within the environment.

This automatic subspace decomposition can serve as a foundation for hierarchical navigation. For example, each derived subspace can be regarded as a cluster in HPA*, but with two notable advantages. First, there is no need to run A* within each cluster to precompute the g-values between exit points, since these values can instead be obtained by evaluating the up and down vertices and identifying the must-pass waypoints between the associated exits. Second, once a high-level path is found on the abstract graph of exits, the low-level search within each subspace can apply the same directional pruning strategy with must-pass waypoints as used in Path Construction. Together, these properties significantly reduce preprocessing time and shrink the search space during online planning.

Experimental Results

We conduct a comprehensive evaluation to assess the performance of Key-Interval A* in terms of runtime, preprocessing time, preprocessing memory consumption, and path quality. We adopt the widely used MovingAI benchmark suite (Stern et al. 2019), which provides grid-based maps along with a large set of predefined start–goal scenarios of varying complexity. Our evaluation covers a diverse set of representative map types, including open spaces, open spaces with random obstacles, mazes, room layouts, warehouse grids and real-game environments.

Two implementations of Key-Interval A* were evaluated, differing only in how they maintain g-scores. The basic version, KIA*, stores a single best g-value for each key interval. A refined variant uses a more detailed composite key consisting of the key interval, the up and down vertices, and the

	Memory (MB)		Prep Time (ms)		Runtime per Instance (ms)				Path Length			
Map type	HPA*	KIA*	HPA*	KIA*	A*	HPA*	KIA*	KIA*(o)	A*	HPA*	KIA*	KIA*(o)
Berlin_1_256	0.54	0.67	24.48	2.17	1.87	0.31	0.08	0.13	204.530	1.011	1.012	1.000
Boston_0_256	0.57	0.79	31.32	3.25	3.31	0.41	0.14	0.82	216.930	1.013	1.017	1.000
Paris_1_256	0.57	0.80	29.95	2.68	4.24	0.45	0.18	3.32	225.410	1.013	1.018	1.000
empty-48-48	0.02	0.02	0.79	0.07	0.02	0.06	0.01	0.01	31.630	1.024	1.000	1.000
den312d	0.04	0.08	3.72	0.27	0.08	0.09	0.03	0.03	56.260	1.036	1.000	1.000
lak303d	0.26	0.38	10.53	1.48	1.54	0.35	0.15	2.18	215.340	1.028	1.010	1.000
orz900d	4.36	4.64	54.44	8.30	25.00	2.84	0.53	8.21	1405.590	1.011	1.007	1.000
w_woundedcoast	1.66	1.84	18.80	3.34	3.38	0.58	0.08	0.16	476.040	1.020	1.001	1.000
maze-32-32-2	0.01	0.05	3.16	0.16	0.06	0.11	0.02	0.03	52.780	1.062	1.000	1.000
maze-128-128-1	0.16	1.13	18.02	4.99	0.89	0.39	0.64	0.62	387.410	1.000	1.000	1.000
maze-128-128-10	0.16	0.22	20.62	0.66	2.32	0.40	0.03	0.03	223.070	1.039	1.000	1.000
random-32-32-20	0.02	0.08	4.42	0.34	0.02	0.10	0.14	0.31	22.700	1.017	1.019	1.000
random-64-64-10	0.07	0.21	11.48	0.88	0.03	0.13	0.35	0.80	44.060	1.015	1.015	1.000
random-64-64-20	0.08	0.29	10.44	1.26	0.05	0.13	0.32	1.50	45.600	1.024	1.022	1.000
room-32-32-4	0.02	0.06	3.60	0.22	0.03	0.10	0.08	0.14	25.860	1.010	1.012	1.000
room-64-64-8	0.05	0.12	8.30	0.40	0.18	0.18	0.06	0.13	61.570	1.040	1.007	1.000
room-64-64-16	0.05	0.07	6.01	0.23	0.31	0.18	0.05	0.39	75.920	1.042	1.005	1.000
warehouse-10-20-10-2-1	0.14	0.65	8.84	2.60	0.03	0.10	13.14	13.27	84.510	1.004	1.000	1.000
warehouse-10-20-10-2-2	0.16	0.67	11.87	2.22	0.04	0.13	7.05	7.01	94.450	1.009	1.000	1.000

Table 1: Experimental results. Reported memory corresponds to preprocessing memory usage. For path length, A* shows the average absolute length, while for the other algorithms the reported values are expressed as a ratio relative to A*.

last waypoint, enabling it to distinguish paths that reach the same interval under different geometric conditions. We denote this variant as KIA*(o), where "(o)" explicitly reflects its design goal of restoring optimality.

We compare both KIA* variants against classical A* and HPA* (cluster size = 10). All algorithms were implemented in C++ and executed single-threaded on an AMD Ryzen 7 5800H processor (3.2 GHz) with 16 GB RAM.

Table 1 summarizes the performance comparison among A*, HPA*, KIA*, and KIA*(o). Overall, KIA* delivers substantial runtime speedups, outperforming both A* and HPA* on almost all map types. On large-scale city maps (e.g., BERLIN_1_256) and complex game maps such as W_WOUNDEDCOAST, KIA* achieves more than an order of magnitude improvement over A* and is consistently 3–7$\times$ faster than HPA*. Similar advantages are observed on structured layouts like mazes, and room-based scenarios (e.g., MAZE-128-128-10), where interval abstraction effectively prunes redundant expansions.

Two scenarios deviate from this trend. First, in dense warehouse maps (e.g., WAREHOUSE-10-20-10-2-1), some key intervals have dozens of neighbors, leading to significantly higher neighbor-handling overhead: KIA* takes 13.14 ms per instance, much slower than HPA*'s 0.10 ms or A*'s 0.03 ms. Second, on maps with random obstacles (e.g., RANDOM-64-64-10), KIA* underperforms both A* and HPA* because the relatively small map size, combined with frequent and irregular contractions and expansions of free space, generates a large number of fragmented key intervals, thereby reducing the effectiveness of interval-based pruning. KIA*(o) exhibits a similar runtime trend across all map types; although its search time is generally higher than HPA* on most maps, it still outperforms A*.

Regarding preprocessing, KIA* relies on lightweight interval scanning and key-point extraction, making it 5–15$\times$ faster than HPA*'s cluster-based abstraction, which requires running A* searches to precompute g-values between exit points (e.g., 2–3 ms on city maps vs. 25–30 ms for HPA*). In terms of memory usage, KIA* is comparable to HPA*.

KIA* occasionally yields paths that are marginally longer than those produced by A* (typically within a 2% deviation), yet it often returns shorter paths than HPA*. The refined variant, KIA*(o), restores path optimality across all experiments, producing solutions that exactly match A* on every benchmark. A formal proof of this optimality property is left for future work.

Conclusion and Future Work

This paper introduced *Key-Interval A** and *Automatic Subspace Decomposition* to accelerate grid-based path planning. KIA* reduces node expansions through interval-based abstraction, achieving substantial speedups over A* and HPA*, while its variant KIA*(o) empirically restores optimality. Automatic Subspace Decomposition partitions the environment into connected subspaces, providing a lightweight foundation for future hierarchical planning.

Several avenues for future research remain: (1) formally investigating the optimality of KIA*(o); (2) improving performance on dense warehouse maps and highly fragmented random maps, where neighbor-handling heuristics may alleviate current inefficiencies; (3) embedding Automatic Subspace Decomposition into hierarchical planners such as HPA*; (4) extending KIA* to any-angle search; and (5) exploring the use of KIA* in multi-agent pathfinding (MAPF), leveraging waypoints to handle conflicts more effectively.

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.

Reproducibility Checklist

Instructions for Authors:

This document outlines key aspects for assessing reproducibility. Please provide your input by editing this `.tex` file directly.

For each question (that applies), replace the "Type your response here" text with your answer.

Example: If a question appears as

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
Type your response here
```

you would change it to:

```
\question{Proofs of all novel claims
are included} {(yes/partial/no)}
yes
```

Please make sure to:

- Replace ONLY the "Type your response here" text and nothing else.
- Use one of the options listed for that question (e.g., **yes**, **no**, **partial**, or **NA**).
- **Not** modify any other part of the `\question` command or any other lines in this document.

You can `\input` this .tex file right before `\end{document}` of your main file or compile it as a stand-alone document. Check the instructions on your conference's website to see if you will be asked to provide this checklist with your paper or separately.

---

1. General Paper Structure

1.1. Includes a conceptual outline and/or pseudocode description of AI methods introduced (yes/partial/no/NA) yes

1.2. Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results (yes/no) yes

1.3. Provides well-marked pedagogical references for less-familiar readers to gain background necessary to replicate the paper (yes/no) yes

2. Theoretical Contributions

2.1. Does this paper make theoretical contributions? (yes/no) yes

If yes, please address the following points:

2.2. All assumptions and restrictions are stated clearly and formally (yes/partial/no) yes

2.3. All novel claims are stated formally (e.g., in theorem statements) (yes/partial/no) yes

2.4. Proofs of all novel claims are included (yes/partial/no) partial

2.5. Proof sketches or intuitions are given for complex and/or novel results (yes/partial/no) yes

2.6. Appropriate citations to theoretical tools used are given (yes/partial/no) yes

2.7. All theoretical claims are demonstrated empirically to hold (yes/partial/no/NA) yes

2.8. All experimental code used to eliminate or disprove claims is included (yes/no/NA) yes

3. Dataset Usage

3.1. Does this paper rely on one or more datasets? (yes/no) yes

If yes, please address the following points:

3.2. A motivation is given for why the experiments are conducted on the selected datasets (yes/partial/no/NA) yes

3.3. All novel datasets introduced in this paper are included in a data appendix (yes/partial/no/NA) yes

3.4. All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no/NA) NA

3.5. All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations (yes/no/NA) yes

3.6. All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available (yes/partial/no/NA) yes

3.7. All datasets that are not publicly available are described in detail, with explanation why publicly available alternatives are not scientifically satisficing (yes/partial/no/NA) NA

4. Computational Experiments

4.1. Does this paper include computational experiments? (yes/no) yes

If yes, please address the following points:

4.2. This paper states the number and range of values tried per (hyper-) parameter during development of the paper, along with the criterion used for selecting the final parameter setting (yes/partial/no/NA) NA

4.3. Any code required for pre-processing data is included in the appendix (yes/partial/no) yes

4.4. All source code required for conducting and analyzing the experiments is included in a code appendix (yes/partial/no) yes

4.5. All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes (yes/partial/no) yes

4.6. All source code implementing new methods have comments detailing the implementation, with references to the paper where each step comes from (yes/partial/no) partial

4.7. If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results (yes/partial/no/NA) NA

4.8. This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks (yes/partial/no) yes

4.9. This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics (yes/partial/no) yes

4.10. This paper states the number of algorithm runs used to compute each reported result (yes/no) no

4.11. Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information (yes/no) no

4.12. The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank) (yes/partial/no) no

4.13. This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments (yes/partial/no/NA) NA