Jump Over Dynamic Block (JODB):an Efficient Line-of-sight Checker for Dynamic Grid/Voxel Maps with Sparse Obstacles

ZHUO YAO, WEI WANG, and YUERI CAI*, Beihang University, China

Background: Line-of-sight (LOS) checking plays a critical role in the computational cost of collision avoidance, particularly in large-scale maps with sparse obstacles, since it requires a cell-by-cell state traversal.
Objectives: To provide a general method for reducing the time cost of LOS checking and thus accelerating path and motion planning, we present an efficient LOS checker, JODB.
Methods: JODB partitions the traversable space into blocks. When our algorithm encounters the surface of a block, its LOS check skips cell-by-cell traversal inside the block and directly jumps to the opposite surface. To handle dynamic maps, we propose a novel data structure, the Space Dyadic Tree (SDT), inspired by quadtree and OctoMap, which recursively partitions the traversable space into blocks. In addition, SDT employs a decision-tree strategy to maximize block size by merging neighboring blocks whenever possible.
Results: Experimental results demonstrate that SDT can be updated in near real-time on large-scale 2D/3D maps with sparse obstacles, and that using SDT significantly reduces the time cost of LOS checking in such maps. Therefore, the time cost of path planning can be significantly reduced by applying SDT-accelerated LOS checking. Specifically, the time cost of LOS checking accelerated by SDT is approximately 40%–60% of that using a quadtree/OctoMap and 22%–70% of that using traditional LOS checking.
Conclusions: Compared with traditional quadtree/OctoMap, SDT is more efficient to initialize and update in dynamic environments, and it makes LOS checking more efficient. To facilitate further research within the community, we have made the source code of the proposed algorithm publicly available. Links to the source code can be found in the Results section.

JAIR Associate Editor: Christine Solnon

JAIR Reference Format:
Zhuo Yao, Wei Wang, and Yueri Cai. 2026. Jump Over Dynamic Block (JODB):an Efficient Line-of-sight Checker for Dynamic Grid/Voxel Maps with Sparse Obstacles. *Journal of Artificial Intelligence Research* 86, Article 22 (July 2026), 21 pages. DOI: 10.1613/jair.1.21467

1 Introduction

In collision avoidance for path planning (e.g., RRT (Jeong et al. 2019; B. H. Meng et al. 2022), Theta* (Daniel et al. 2010; Han et al. 2022)) and motion planning ((Jian et al. 2021); (Nayak et al. 2023); (J. Yao and Zhao 2025); (K. Meng et al. 2020); (Ji et al. 2019)), LOS checking plays a pivotal role. A classic method for performing LOS checks is the Bresenham algorithm (Bresenham 1965), which needs cell-by-cell grid traversal. Thus, its computational demands can be significant, particularly when dealing with large-scale or sparse maps. In practice, more than half of the time in path planning is devoted to LOS checking. Consequently, accelerating the checking process becomes crucial for achieving more efficient path planning and motion planning, enabling faster navigation system responses and reducing computational resource requirements.

*Corresponding Author.

Authors' Contact Information: Zhuo Yao, ORCID: 0000-0001-7532-4200, yaozhuo@buaa.edu.cn; Wei Wang, ORCID: 0000-0002-7215-7627, wangweilab@buaa.edu.cn; Yueri Cai, ORCID: 0000-0003-4494-6300, caiyueri@buaa.edu.cn, Beihang University, Beijing, Beijing, China.

DOI: 10.1613/jair.1.21467

Existing work on reducing the computational cost of LOS checking primarily focuses on decreasing the number of LOS checks (see Section 2) or on avoiding the use of LOS checks, rather than on reducing the cost of each individual check, and is therefore limited to specific cases.

To address this issue, in our previous work (Z. Yao et al. 2023), we introduced a novel algorithm called Jump Over Block (JOB), which aims to accelerate LOS checking. The approach substantially reduces the time cost of LOS checking in maps with sparse obstacles. However, it is limited to static environments, since all blocks must be updated whenever the map changes, which is computationally expensive.

To overcome this limitation, we propose a new algorithm, Jump Over Dynamic Block (JODB), designed for dynamic maps. JODB updates only the blocks that contain cells whose states have changed, making it substantially more efficient for dynamic environments. The major difference between JODB and our previous work is that JODB introduces a new data structure for efficient block detection in dynamic maps, along with several improvements that further enhance its efficiency.

The subsequent sections of this article are structured as follows. Section 2 provides an introduction to relevant studies concerning the reduction of the time cost of LOS checks within the context of path planning; it also reviews related work on quadtree and OctoMap. Section 3 describes the major processes involved in our method. Section 4 presents key results of SDT, including its time cost of initialization and update, and how it accelerates LOS checks in dynamic maps with different obstacle densities. Finally, in Section 5, we discuss the results obtained and potential drawbacks of the proposed method.

This letter contributes the following:

(1) Inspired by quadtrees and OctoMap, we propose the Space Dyadic Tree (SDT), a dimension-independent generalization of these structures. Quadtrees and OctoMap were originally designed to reduce the memory usage of grid maps; to the best of our knowledge, this work is the first to exploit them for accelerating LOS checking.
(2) To further improve the efficiency of SDT for LOS checking, we introduce several practical optimizations, including a minimum block width constraint and map downsampling, which significantly reduce the time cost of SDT initialization and update. In addition, a decision-tree-based block merging strategy is proposed to locally maximize block sizes, thereby reducing the number of blocks traversed during LOS checking.

2 Related Works

2.1 Reducing Time Cost of LOS Check

A strategy to speed up LOS checks is to reduce the total number of LOS checks and only perform LOS checks when necessary. Hauser et al. (Hauser 2015) proposed two asymptotically optimal planners, Lazy-PRM* and Lazy-RRG*, that eliminate the majority of collision checks using a lazy strategy. This implies that edges are not immediately checked for collision; rather, they are checked only when a better path to the goal is found. This approach avoids checking the vast majority of edges that have no chance of being on an optimal path. Zhang et al. (Zhang, Tang, and Liu 2019; Zhang, Tang, Zhou, et al. 2019) introduced Late Line-of-Sight Check A* (LLA*) to reduce the frequency of LOS checks. This involves performing an LOS check of a vertex v and its parent when v is about to be expanded.

Bialkowski et al. (Bialkowski et al. 2013) proposed an alternative approach to decrease the frequency of collision checking within the framework of sampling-based motion planning. During the LOS check, it is determined whether the distances from both ends of a line to the nearest vertex v (a randomly sampled point) are smaller than v's distance to its nearest obstacle. If this condition is met, the line is considered collision-free; otherwise, a standard line collision check is performed.

The aforementioned algorithms achieve lower time costs than the original path planning methods by reducing the frequency of LOS checks. However, it is important to note that these approaches do not alter the fundamental

nature of LOS checks. Consequently, their applicability remains confined to graph-based path planning scenarios. In other scenarios, geometric or other factors prevent these methods from significantly reducing the number of required LOS checks.

Popov et al. (Popov et al. 2013) introduced a generic LOS check caching algorithm that stores all LOS check results in a single global hashtable with efficient storage and retrieval. If the start and end points of a new LOS check are similar to those of a previous check in the hashtable, it returns the previous result to avoid performing a new check. However, its performance is initially comparable to that of the traditional LOS check, and it only becomes effective in avoiding redundant checks once nearly every pair of points in the map has been processed. Moreover, to reduce memory usage, it does not store all LOS check results, which introduces a loss of precision. In contrast, JODB achieves the same accuracy as the traditional LOS check, with no loss of precision.

Another strategy to expedite LOS checks is to limit the range of examination to the local map instead of doing LOS checks between two points. Anya and ENL-SVG are representative approaches based on this strategy.

Anya (Harabor and Grastien 2013; Harabor, Grastien, et al. 2016) introduced the concept of interval search, which requires no preprocessing and expands from the current vertex to identify all visible vertices, stopping upon encountering obstacles. By integrating interval search with A*-based search, Anya guarantees optimal path length on uniform-cost grid maps while allowing movement in arbitrary directions. This method was designed for 2D grid maps. Another similar method, called the "line-of-sight scan," was proposed in ENL-SVG (Oh and Leong 2017), an optimal any-angle pathfinding method. It incorporates a precomputation technique that generates sparse visibility graphs, which requires a large number of visibility checks. Line-of-sight scans exhibit runtime dependence on the number of visible vertices for each vertex, while LOS checks depend on the total number of vertices in the entire map. Line-of-sight scans are considerably faster, especially in larger maps with crowded obstacles.

However, for large-scale maps with sparse obstacles, such as voxel maps in Brewer et al. (Brewer and N. R. Sturtevant 2018), local scan methods often necessitate scanning nearly the entire space. Furthermore, existing works primarily focus on 2D grid maps, and to the best of our knowledge, there is currently no LOS scan method available for 3D or higher-dimensional maps. Moreover, this strategy is limited to visibility-based path planning, whereas our method is applicable to various scenarios that need LOS checks.

2.2 Comparison with Quadtree and OctoMap

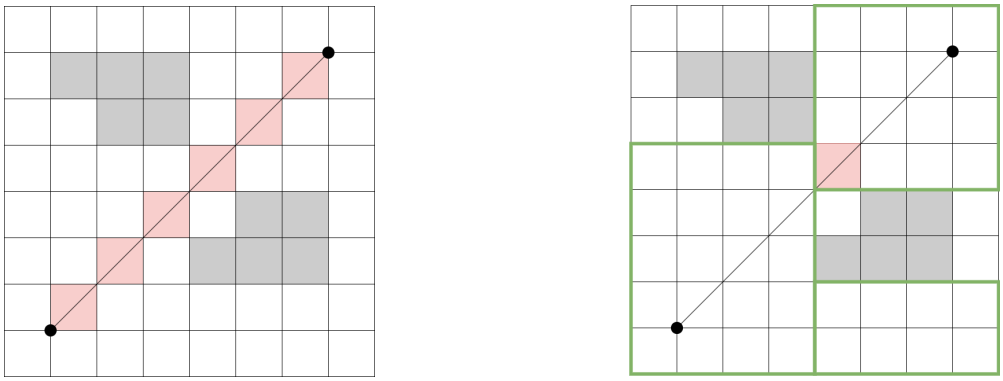
OctoMap (Hornung et al. 2013) and quadtree (Vörös 2001) divide the passable space into passable and unpassable areas for efficiency in storage, query, and update. Each area size is fixed to powers of 2. These passable areas can be used as blocks to accelerate LOS checks, although this was not their primary purpose. Considering that acceleration of LOS checks in dynamic maps requires efficiency in querying and updating passable areas, OctoMap and quadtree could be utilized to split the passable areas for JOB. However, quadtree and OctoMap are limited to 2D and 3D maps, respectively. To maintain the versatility of our algorithm, we generalize quadtree and OctoMap to obtain a data structure that is not limited to a specific dimension: the Space Dyadic Tree (SDT). Compared with quadtrees and OctoMap, we propose several specific improvements to SDT to improve initialization and update efficiency for accelerating LOS checking in dynamic maps. Details are provided in the Methodology section.

3 Methodology

3.1 Definitions

In this section, we present fundamental definitions pertaining to JODB. Considering that our method is dimension-independent, the definitions provided apply to any-dimensional grid space $C_{\mathcal{N}}, \mathcal{N} \geq 2$.

3.1.1 Grid Space. Let $C_{\mathcal{N}}$ ($\mathcal{N} \geq 2$) denote a finite $\mathcal{N}$-dimensional integer Euclidean space, where the size of the space is defined as $\mathcal{D} \subseteq C_{\mathcal{N}}$, with $\mathcal{D} = \{d_1, d_2, ..., d_i, ..., d_{\mathcal{N}}\}$ and $d_i \in \mathbb{N}$. Here, d_i is the range of the i-th dimension.



A: how traditional LOS check works

B:how JODB's LOS check works

Fig. 1. The two subfigures provide an illustrative example of how JOB accelerates LOS checks in a grid map. The left figure shows the process of the conventional LOS check, while the right figure illustrates how JODB works. Obstacle cells are shown in gray, cells evaluated during the LOS check are highlighted in pink, passable blocks are represented in green, and the start and end points of the LOS are indicated by black dots. In the conventional approach, every cell traversed by the line must be checked. In contrast, the JODB method skips the internal cells within each block and examines only those located along the block boundaries, thereby significantly reducing the computational cost of LOS checking.

3.1.2 Cell State. There are only two possible states for a cell/element in $C_{\mathcal{N}}$: passable or unpassable. The set of all passable cells in $C_{\mathcal{N}}$ is denoted as $\mathcal{F} \subseteq C_{\mathcal{N}}$, while the set of all unpassable cells is denoted as $\mathcal{O} \subseteq C_{\mathcal{N}}$. Therefore, $\mathcal{F} \cup \mathcal{O} = C_{\mathcal{N}}$. For convenience, we denote all cells outside $C_{\mathcal{N}}$ as unpassable.

3.1.3 Line Collision Condition. Denote the line connecting two cells g_1 and g_2 as $g_1 \rightarrow g_2$. We define the set of all cells that $g_1 \rightarrow g_2$ crosses as $\omega(g_1 \rightarrow g_2)$. The line $g_1 \rightarrow g_2$ is said to be in collision if there exists $g \in \omega(g_1 \rightarrow g_2)$ such that $g \in \mathcal{O} \subseteq C_{\mathcal{N}}$. If $g_1 \rightarrow g_2$ is in collision, it is denoted as $(g_1 \rightarrow g_2) \in \mathcal{O}$, i.e., the line fails to pass the LOS check; otherwise, it is denoted as $(g_1 \rightarrow g_2) \in \mathcal{F} \subseteq C_{\mathcal{N}}$, i.e., the line passes the LOS check.

3.1.4 Block. A block is a fully passable box-shaped space b defined by two cells: a cell $g_{\min}$ with minimum coordinates and a cell $g_{\max}$ with maximum coordinates. There is no overlap between any two blocks. An example of blocks in C_2 is shown in Fig. 1 B. The set of all blocks is denoted as $\mathcal{B}$, and the set of all blocks in $C_{\mathcal{N}}$ with a minimum required width τ is denoted as $\mathcal{B} \subseteq (C_{\mathcal{N}}, \tau)$.

Our method partitions the passable space into blocks, and the LOS check skips cells within these blocks (as illustrated in Fig. 1), thereby expediting the process. This strategy substantially reduces the time cost of LOS checks in scenarios with sparse obstacles. Importantly, it produces results identical to those of the traditional LOS check, thereby preserving accuracy.

3.1.5 Space Dyadic Tree. A Space Dyadic Tree (SDT) is a tree data structure used for spatial partitioning of $C_{\mathcal{N}}$. Each node in the tree represents an equilateral region and has exactly $2^{\mathcal{N}}$ children, recursively subdividing the volume into $2^{\mathcal{N}}$ smaller areas whose side lengths are half of the parent area. Side lengths of all areas are $2^x, x \in \mathbb{N}$. If a node's area's side length is 1 (i.e., 2^0), we denote it as reaching the end of the tree, as there is no further smaller block.

If a node's area has no unpassable cells, we denote it as a passable area/node. If a node's area has no passable cells, we denote it as an unpassable area/node. If a node's area has both passable and unpassable cells, we denote it as a mixed area/node. **These passable areas are used as blocks to accelerate LOS checking**. All non-leaf

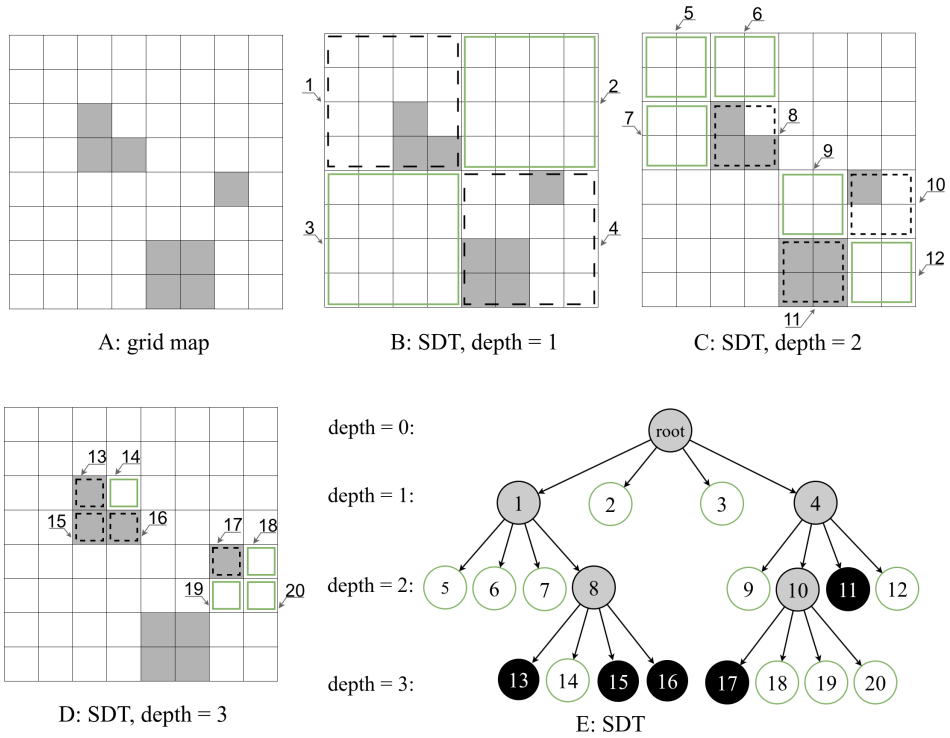


Fig. 2. These figures show a grid map and its SDT. Figure A is the raw grid map; Figure B, C, and D show the SDT's nodes at different depths. Figure E provides an overview of the SDT. Grey nodes in the SDT are mixed nodes, black nodes are unpassable nodes, and green nodes are passable leaf nodes (i.e., blocks).

nodes are mixed nodes and all leaf nodes are passable or unpassable nodes. SDT is denoted as $\mathcal{S}$ in the following text and algorithms. A toy example of SDT is shown in Fig. 2.

Compared to JOB's block detection, which needs to maximize all blocks, SDT's update is limited to the local area when a cell's state changes, avoiding the need to update almost all blocks and making it more efficient in dynamic maps. For example, in the map shown in Fig. 2, if the cell corresponding to node 20 is changed from passable to occupied, only node 20 and its ancestor nodes (nodes 10, 4, and the root) need to be updated. The states of all other nodes remain unchanged, as changes to a node only affect that node and its ancestors in the tree.

Since updates in the SDT are limited to a local area when a cell's state changes, JODB uses the leaf nodes of the SDT as blocks to enable efficient updates in maps with dynamic obstacles. To determine blocks from a map, we first construct the map's SDT and then select its leaf nodes as blocks.

3.1.6 Time Cost Ratio. The time cost ratio is defined as the ratio of the time cost of JODB's LOS check to that of the traditional LOS check. A smaller ratio indicates better acceleration performance.

3.2 Initialization and Updating of SDT

In this section, we describe how to initialize the SDT from $C_{\mathcal{N}}$ and how to update the SDT when a cell's state in $C_{\mathcal{N}}$ changes. Both the initialization and update processes rely on a core function, `update_cell_state`($\mathcal{S}, g$, *is_occupied*),

where $\mathcal{S}$ is the SDT to be updated, g is the cell whose state has changed, and *is_occupied* is a boolean indicating the new state; *is_occupied* = $True$ sets the cell to unpassable, and *is_occupied* = $False$ sets it to passable. After calling *update_cell_state*, the state of cell g in the SDT is updated accordingly.

We first explain how this function is used to implement both the initialization and update of the SDT. Then, we describe the implementation details of `update_cell_state`.

Algorithm 1: Initialization of SDT

Input: $C_{\mathcal{N}}$
Output: $\mathcal{S}$
$\mathcal{S} = \{root\}$;
for *passable grid* g *in* $C_{\mathcal{N}}$ **do**
    `update_cell_state`$(\mathcal{S}, g, False)$;
return $\mathcal{S}$;

Algorithm 2: Update SDT

Input: $G_1, G_2, \mathcal{S}$ // G_1: new passable cells, G_2: new unpassable cells
Output: $\mathcal{S}$
$\mathcal{S} = \{root\}$;
for *grid* g *in* G_1 **do**
    `update_cell_state`$(\mathcal{S}, g, False)$;
for *grid* g *in* G_2 **do**
    `update_cell_state`$(\mathcal{S}, g, True)$;
return $\mathcal{S}$;

During the initialization of the SDT, we account for the fact that the size of $C_{\mathcal{N}}$ may not be a power of two. Therefore, each dimension of $C_{\mathcal{N}}$ is expanded to the nearest power of two. The expanded regions are always considered unpassable. As a result, the SDT is initially constructed with a root node marked as unpassable.

Subsequently, each passable cell in $C_{\mathcal{N}}$ is marked as passable in $\mathcal{S}$ by invoking the `update_cell_state` function. This process completes the initialization of the SDT, as outlined in Algorithm 1.

For updating the SDT, a similar approach is followed: new passable cells are marked as passable in $\mathcal{S}$, and newly blocked (unpassable) cells are marked as unpassable, as illustrated in Algorithm 2.

The implementation of the function `update_cell_state`$(\mathcal{S}, g,$ *is_occupied*$)$ consists of the following steps:

(1) Locate the leaf node n in $\mathcal{S}$ that contains the cell g. If the state of n is already consistent with the desired state of g, no update is necessary, and the function exits early (Lines 1–3 in Algorithm 3).
(2) If n reaches the end of $\mathcal{S}$ (i.e., the smallest unit in the tree), set its state to *is_occupied*. Otherwise, recursively create child nodes until reaching the end of the tree. The node corresponding to g is then set to *is_occupied*, while all other newly created sibling nodes inherit the original state of n. The function then exits (Lines 4–13 in Algorithm 3).
(3) If n is at the leaf level after modification, the algorithm checks whether all of its sibling nodes now share the same state. If so, the parent node is set to the same state as its child nodes, n is set to its parent node, and all child nodes are removed. This merging process is repeated recursively up the tree until the parent's child nodes are no longer the same (Lines 14–17 in Algorithm 3).

Since only the parent nodes of the updated nodes need to be updated, this process is highly efficient in dynamic maps.

Algorithm 3: *update_cell_state*

Input: $\mathcal{S}, g, is\_occupied$
Output: $\mathcal{S}$

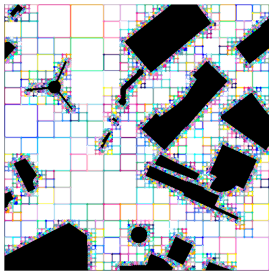
```
n = the leaf node that contains g; // step 1
if n.is_mixed = False and n.is_occupied = is_occupied then
    return;
// step 2
if n reaches the end of S then
    n.is_occupied = is_occupied;
else
    while n is not reaching the end of the tree do
        n.is_mixed = True; // its child nodes have both passable and impassable cells
        create 2^N child nodes and add them to n;
        n = n's child that contains g;
    n.is_occupied = is_occupied;
    return;
// step 3
while n's parent node's child nodes all have the same state do
    n = n's parent node;
    n.is_mixed = False;
    remove all child nodes of n;
    set n's state to its child nodes' state;
return;
```

3.3 Generate Blocks from SDT

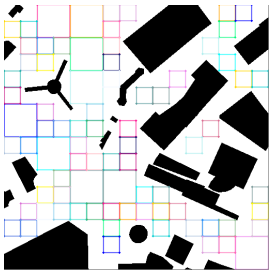
Essentially, blocks correspond to the passable leaf nodes of the SDT whose side lengths are smaller than the threshold of minimum width τ. To accelerate the LOS check, it is necessary to determine whether a given cell belongs to a block and, if so, identify the corresponding block. While it is possible to locate the leaf node containing a cell by traversing the SDT from the root, this approach is computationally expensive. To improve efficiency in JODB, we maintain a mapping from each cell to the reference of the block it belongs to, or to *null* if the cell is not contained in any block. During SDT initialization, this mapping is constructed after all passable cells have been processed with *update_cell_state*, thereby avoiding frequent updates. In contrast, during online updates to the SDT, the mapping is updated immediately after each call to *update_cell_state*.

3.4 Accelerate LOS Check with Blocks

In this section, we describe how blocks are utilized to accelerate LOS checks. The key distinction between the traditional LOS check and our proposed method lies in the introduction of a block-skipping mechanism, referred to as “jump over blocks", which is applicable to spaces of arbitrary dimensionality.



A: $\tau = 2$



B: $\tau = 32$

Fig. 3. Figures A and B illustrate the blocks generated from the SDT on a map with different values of the minimum block width τ. The map, taken from (N. Sturtevant 2012), has a size of 1024×1024. In the figures, black regions indicate unpassable areas, white regions denote passable areas, and colored hollow squares represent the detected blocks.

Specifically, when traversing all grid cells along the line segment from g_1 to g_2 with a given minimum block width τ, the algorithm checks whether the current cell lies on the surface of a block b. If so, it skips the internal cells of b and jumps directly to the opposite surface of the block in the direction of the line.

This optimization avoids redundant cell-state checks within fully passable blocks while preserving the exactness of the LOS result. In other words, the acceleration does not compromise correctness. The detailed procedure for this block-skipping acceleration is outlined in Algorithm 4.

Algorithm 4: Block accelerated LOS check

Input: $g_1, g_2, C_{\mathcal{N}}, \mathcal{B}' \subseteq (C_{\mathcal{N}}, \tau)$ // two ends of the line, grid space $C_{\mathcal{N}}$ and related merged blocks $\mathcal{B}$
Output: Whether the current line collides with obstacles

```
step = 0;
while step < Ω(g1 ⊆ g2) do
    if g ∈ O ⊆ C_N then
        return True; // g1 → g2 is colliding with obstacles
    if g ∈ b ∈ B' ⊆ (C_N, τ) then
        step = step + the length of the line g1 → g2 intersecting b;
    else
        step = step + 1;
return False; // g1 → g2 does not collide with obstacles
```

3.5 Improvements

In this section, we introduce several improvements to our algorithm. These improvements are not indispensable components of the core method, but they can substantially enhance the overall performance. Specifically, a minimum block width is imposed to limit the smallest block size, thereby reducing the time cost of JODB's LOS check. Map downsampling is applied to decrease the initialization and update costs of the SDT. In addition, block merging is employed to combine small blocks into larger ones, further reducing the computational cost of JODB's LOS check. More details are provided in the following subsections.

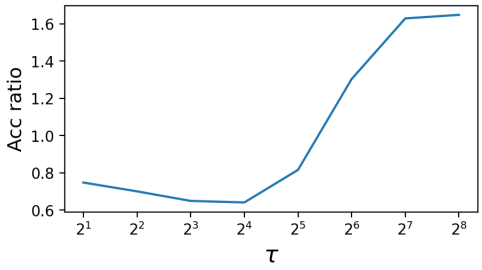


Fig. 4. This figure illustrates the time cost ratio of JODB under various values of τ. As shown in the figure, the time cost ratio initially decreases, reaching a minimum value of 0.62 at $\tau = 2^4$, and then begins to increase. Eventually, the time cost ratio stabilizes at approximately 1.6.

3.5.1 Choosing Minimal Block Width. In this section, we analyze the difference in time cost between the traditional LOS check and the LOS check used in JODB. We then show that jumping over a block in JODB's LOS check is more time-consuming than moving to the next cell in the traditional LOS check. This observation motivates the introduction of a minimum block width to reduce the time cost of JODB's LOS check as much as possible.

In the traditional line-of-sight (LOS) check, the algorithm proceeds in a cell-by-cell manner until the first unpassable cell is encountered.

For JODB, the time cost consists of two parts:

1, for segments that are not covered by blocks, the time cost is the same as that of the traditional LOS check and is proportional to the number of visited cells;

2, for segments covered by blocks, the time cost is proportional to the number of visited blocks.

Since the traditional LOS check also visits the cells in the first part, the difference in time cost between the traditional LOS check and JODB's LOS check is mainly determined by the difference between the cost of visiting a block and that of visiting the cells contained in the block. Therefore, we focus on analyzing the computational components involved in visiting blocks and cells.

We assume that the time cost of checking whether a cell is passable and whether a cell belongs to a block is the same constant, as both operations are implemented via indexed array access. Under this assumption, the computational cost of visiting cells mainly comes from computing the next cell based on the current cell and the line equation.

In two-dimensional or three-dimensional space, a straight line can be represented in parametric form:

$$\mathbf{p}(t) = \mathbf{p}_0 + t\,\mathbf{d}, \quad t \in \mathbb{R}, \tag{1}$$

where $\mathbf{p}_0$ is the entry point and $\mathbf{d}$ is the unit direction vector of the line. Assume the current cell is denoted by c, and the next cell along the line is denoted by c':

$$c' = c + d \tag{2}$$

For block traversal, the main computation is to determine the exit cell (where the line leaves the block) given the entry cell (where the line enters the block), the block boundaries $g_{\min}$ and $g_{\max}$, and the line equation.

To address this problem, we adopt the classical slab method (Foley et al. 1996). Consider a block defined by its minimum and maximum coordinates:

$$g_{\min} = (g^1_{\min}, \ldots, g^{\mathcal{N}}_{\min}), \quad g_{\max} = (g^1_{\max}, \ldots, g^{\mathcal{N}}_{\max}). \tag{3}$$

For convenience, the line is parameterized as

$$\mathbf{p}(t) = p_{entry} + t\,\mathbf{d}, \quad t \geq 0, \tag{4}$$

where p_{entry} is the known entry point of the line into the block and $\mathbf{d} = \{d^1, d^2, ..., d^{\mathcal{N}}\}$ is the direction vector.

For each dimension $i \in \{1, \ldots, \mathcal{N}\}$, the parameter at which the line exits the block along that dimension is computed as

$$t_i = \begin{cases} \dfrac{g^i_{\max} - p^i_{\text{entry}}}{d^i}, & d^i > 0, \\ \dfrac{g^i_{\min} - p^i_{\text{entry}}}{d^i}, & d^i < 0. \end{cases} \tag{5}$$

The global exit parameter is then obtained by

$$t_{\text{exit}} = \min_{i=1,\ldots,n} (t_i)\,. \tag{6}$$

Finally, the exit point of the line from the block is given by

$$p_{exit} = p_{entry} + t_{exit}\,\mathbf{d}. \tag{7}$$

From the above analysis, we observe that computing the exit point of a block is more complex than computing the next point along the line. Moreover, the time cost of finding the exit point is approximately constant and independent of the block size.

Therefore, we hypothesize that using all leaf nodes of the SDT as blocks for accelerating LOS checks may introduce side effects on the overall time cost: a large number of small blocks increases the frequency of exit-point computations, while discarding too many small blocks and retaining only a few large blocks leads to excessive cell-by-cell traversal, which also increases the time cost.

To identify an appropriate minimum block width, we evaluate how the time cost ratio varies with different minimum block widths. Specifically, we test the performance of JODB under varying values of τ using the map shown in Fig. 3. The results, presented in Fig. 4, are consistent with our analysis.

In practical applications of JODB, we recommend testing multiple τ values and selecting the one that yields the lowest time cost ratio to achieve optimal performance.

3.5.2 Downsampling of Map. Essentially, SDT is a dimension-independent generalization of quadtree and OctoMap. As a result, the initialization and update costs of both SDT and quadtree/OctoMap increase rapidly as the map size grows (see Section 4 for examples). To reduce the initialization and update costs of SDT, we introduce a downsampling step on the map before the initialization of SDT. The scale factor is set to $\frac{1}{\tau}$, so that a passable cell in the shrunk map corresponds to a minimal block in the original map. Specifically, one cell in the shrunk map corresponds to $\tau^{\mathcal{N}}$ cells in the original map. If all of these cells are passable, then the corresponding cell in the shrunk map is marked as passable; otherwise, it is marked as impassable. Since τ is a power of 2, the shrunk map produces the same tree structure. After downsampling, blocks smaller than τ disappear. However, this process does not affect the performance of JODB, as such blocks are not considered during LOS checks. When we use blocks to accelerate LOS checks, the size of blocks in the SDT of the downsampled map needs to be multiplied by τ to get the original size of the blocks.

For example, for the map shown in Fig. 2, when τ is set to 2, the SDT of the downsampled map is shown in Fig. 5. The SDT in Fig. 5 has the same structure as that in Fig. 2 after blocks with widths smaller than 2 are discarded.

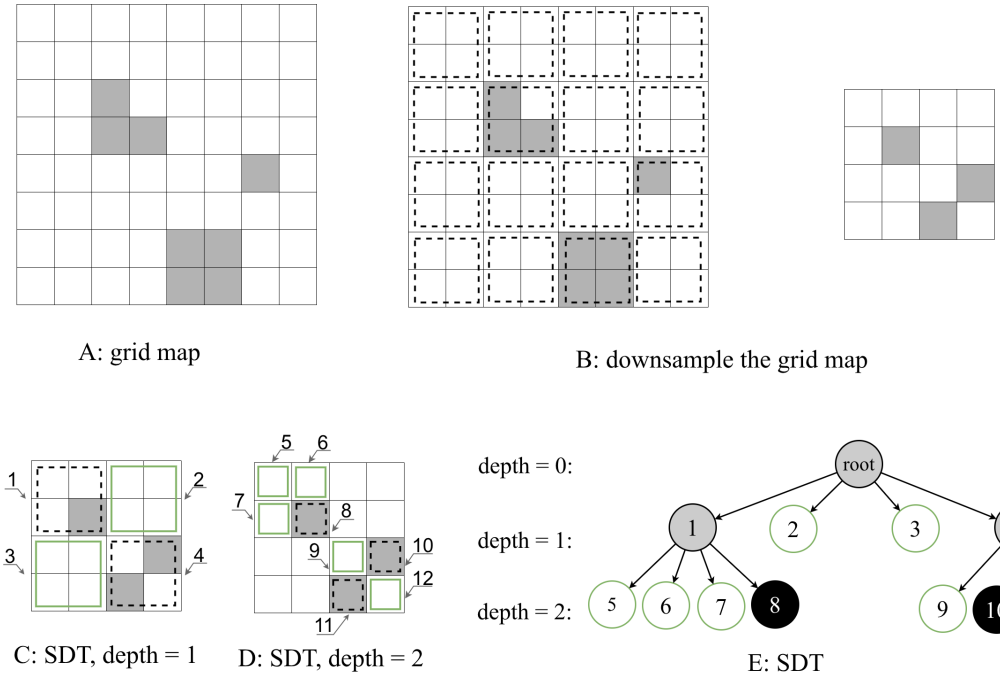


Fig. 5. These figures illustrate the downsampling process of a grid map and the corresponding SDT. Figure A shows the original grid map, while Figure B presents the downsampled map. Figures C and D depict the SDT nodes at different depths. Figure E provides an overall view of the SDT structure. During downsampling of the map, 2^N cells are merged into one cell. The merged cell is passable only if all cells of the 2^N cells are passable; otherwise, the merged cell is impassable.

Fig. 6. These figures demonstrate how a block in an area shown in Figure A expands in four directions. The dashed boxes indicate the four blocks resulting from the expansion in each direction.

3.5.3 Merging Blocks. Larger blocks require fewer visits, thereby reducing the time cost of LOS checks accelerated by blocks. To obtain larger blocks, we perform expansion at the block level; specifically, a block is expanded in a given direction if it can be merged with neighboring blocks in that direction to form a larger block. An illustrative example is provided in Fig. 6.

We observe that the order of expansion determines the final block size, as shown in Fig. 7. To maximize the size of the final block, we introduce a decision tree. At each step, we attempt to expand in all directions and

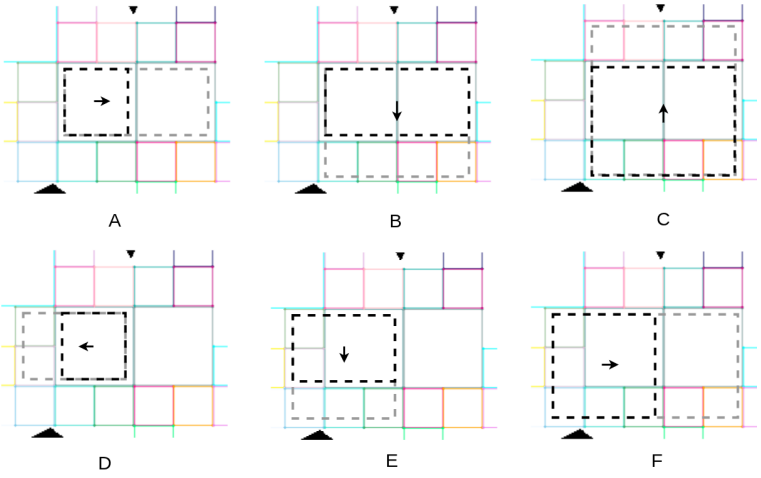
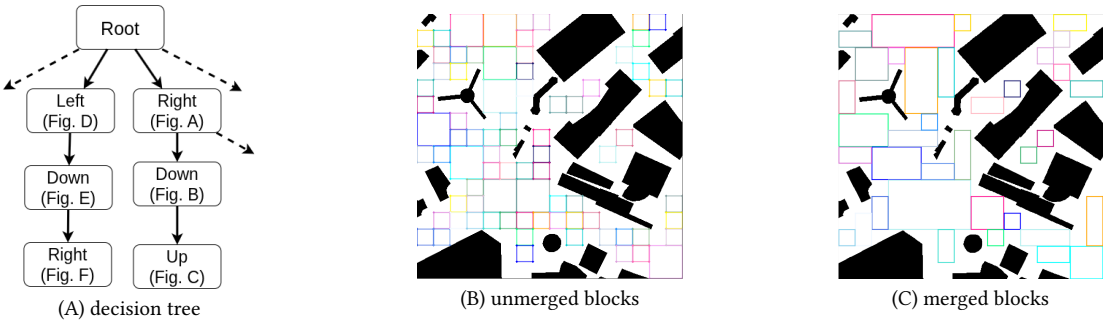


Fig. 7. This figure demonstrates two ways in which the block in Fig. 6B expands in four directions until it can no longer expand. Figs. A, B, C show the block starting to expand to the right, then downward, and finally upward, while Figs. D, E, F show the block starting to expand to the left, then downward, and finally to the right. Black dashed boxes indicate the current blocks, and light gray dashed boxes indicate the blocks after expansion. The final block size for Figs. A, B, C is $3 \times 5 = 15$, and for Figs. D, E, F it is $4 \times 4 = 16$.



(A) decision tree (B) unmerged blocks (C) merged blocks

Fig. 8. Figure A illustrates the decision tree for the two expansion strategies of the block shown in Fig. 7. Black arrows and boxes represent expansions shown in Fig. 7, while dashed arrows indicate expansions not shown in that figure. *Visualizing all possible expansions for the block in Fig. 6A would require excessive space, so only two expansion strategies are shown in Fig. 7 and this figure.* Figure B shows the unmerged blocks for the map in Fig. 3B. Figure C shows the merged blocks for the same map. Compared to Figure B, the blocks in Figure C are larger and fewer in number. As a result, LOS checks accelerated by these blocks require less time than those accelerated by the blocks in Figure B.

store every merged block in the decision tree. The expansion of a block stops if there is no block in the given direction, an obstacle is encountered, or a block already belongs to another merged block. Once no merged block can expand in any direction, we select the merged block with the maximum size as the result of the merging process, as illustrated in Algorithm 5. The decision tree corresponding to Fig. 7 is shown in Fig. 8A.

For the map in Fig. 3B, the resulting merged blocks are shown in Fig. 8B.

Algorithm 5: Merge blocks

Input: $\mathcal{B} \subseteq (C_{\mathcal{N}}, \tau)$ // Raw blocks of SDT
Output: Merged blocks $\mathcal{B}'$

```
B' = {}; // merged blocks
for b in B do
    b' = {b}; // initialize the first merged block
    B_1 = {b'}; // a set of temporary merged blocks
    while B_1 is not empty do
        B_2 = {}; // another set of temporary merged blocks
        for b'' in B_1 do
            for direction d in 2 * N directions do
                if b'' can expand in direction d then
                    // construct a new block n
                    n.parent = b'';
                    add n to b''.children;
                    add n to B_2;
        B_1 = B_2;
    add the largest block in B_1 to B';
return B';
```

4 Results

In general, it is important to compare a proposed method with existing approaches addressing similar problems. However, to the best of our knowledge, no prior work has specifically aimed to accelerate LOS checking by reducing the computational cost of the LOS check itself. Existing methods primarily reduce the time cost of LOS checks by either decreasing the number of LOS checks (e.g., (Bialkowski et al. 2013; Hauser 2015; Popov et al. 2013; Zhang, Tang, and Liu 2019; Zhang, Tang, Zhou, et al. 2019)) or by avoiding LOS checks altogether (e.g., (Brewer and N. R. Sturtevant 2018; Harabor and Grastien 2013)). No existing method focuses on reducing the cost of each LOS check itself. Therefore, in the experimental results, we do not compare with these methods. Instead, we compare SDT with quadtree and OctoMap, as they also partition passable space into blocks, which can be used to accelerate LOS checks, although this is not their primary purpose.

In comparison, we focus on evaluating the following key aspects: (1) the time cost of LOS checks; (2) the time cost of updating the SDT and quadtree/OctoMap, which is crucial in dynamically changing environments; and (3) the initialization time costs of the SDT and the raw quadtree/OctoMap. To assess the adaptability and robustness of our algorithm, we conduct experiments on both 2D and 3D dynamic maps with varying sizes and obstacle densities.

In terms of maps, we generated 500 random 2D maps and 400 random 3D maps. Each map contains a random number of rectangular (2D) or cuboid (3D) obstacles of randomly assigned sizes, resulting in varying obstacle densities, as illustrated in Fig. 9. The 2D maps have side lengths of 200, 400, 600, 800, and 1000, while the 3D maps have side lengths of 200, 400, 600, and 800. All maps are square (2D) or cubic (3D) and contain only dynamic obstacles; no static obstacles are present. To simulate dynamic environments, all obstacles move randomly during the experiments. The width of each obstacle is randomly selected from a range between one-fiftieth and one-tenth of the map side length. Since updating the SDT is more time-consuming when obstacles move longer distances,

(A)2D, Φ = 31.1% (B)2D, Φ = 86.5% (C)3D, Φ = 31.6% (D)3D, Φ = 56.8%

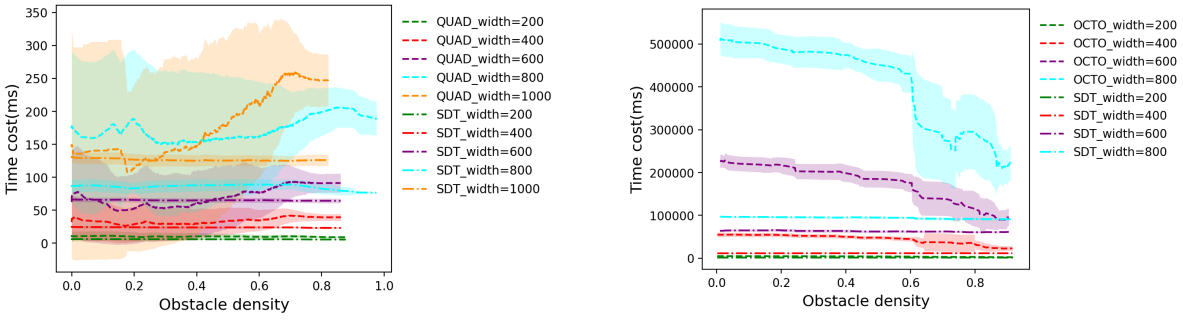
Fig. 9. These figures present a selection of 2D and 3D randomly generated maps along with their corresponding Φ values. As illustrated, the collision ratio Φ increases consistently with the rise in obstacle density, validating its effectiveness as an indicator of spatial obstruction across different map configurations.

we investigate the update time cost for four obstacle movement distances: 1, 2, 4, and 16 cells. This allows us to analyze how the update time varies with obstacle movement distance. Moreover, to evaluate the performance of SDT in real-world applications, we test our algorithm on dynamically changing real-world maps.

To quantify obstacle density across different maps, we introduce a metric called the *collision ratio*, denoted as Φ, as an indicator of obstacle density. This ratio measures the likelihood of collision along the line between two randomly selected passable cells in the grid space C_N. The value of Φ lies in the range $[0, 1]$: it approaches 0 in sparsely obstructed environments and approaches 1 in densely obstructed ones. Importantly, Φ is independent of the scale and dimensionality of the grid space, making it a reliable and consistent indicator of obstacle density.

The experiments were performed on a computer running Ubuntu 20.04, equipped with a Ryzen 7 5800H (3.2GHz) CPU and 128GB of memory. All experiments were implemented in C++. To facilitate further research within the community, we have made the source code of the proposed algorithm publicly available[1].

4.1 Experiment on Massive Simulated Maps



A, time cost comparison (2D) B, time cost comparison (3D)

Fig. 10. Figures A and B show the average initialization time cost of quadtree/OctoMap and SDT on the aforementioned random 2D and 3D maps, respectively. The shaded regions indicate the mean ± one standard deviation computed using a sliding-window method, and the same applies to the following figures.

In this section, we evaluate the initialization and update time costs of the SDT and quadtree/OctoMap, as well as the time cost of LOS checks. The initialization time cost refers to the time required to construct the SDT or a quadtree/OctoMap from the map, while the update time cost refers to the time required to update the SDT or

[1]https://github.com/JoeYao-bit/JumpOverBlock/tree/main

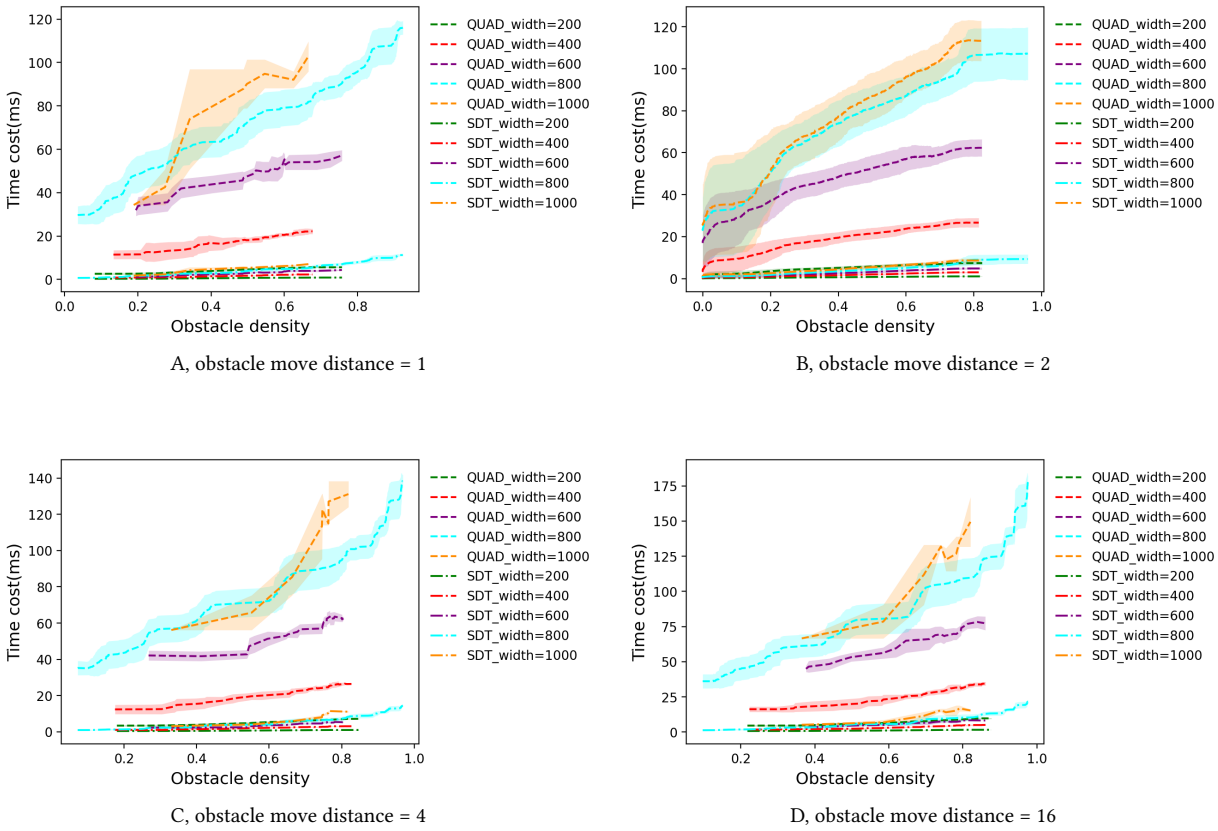
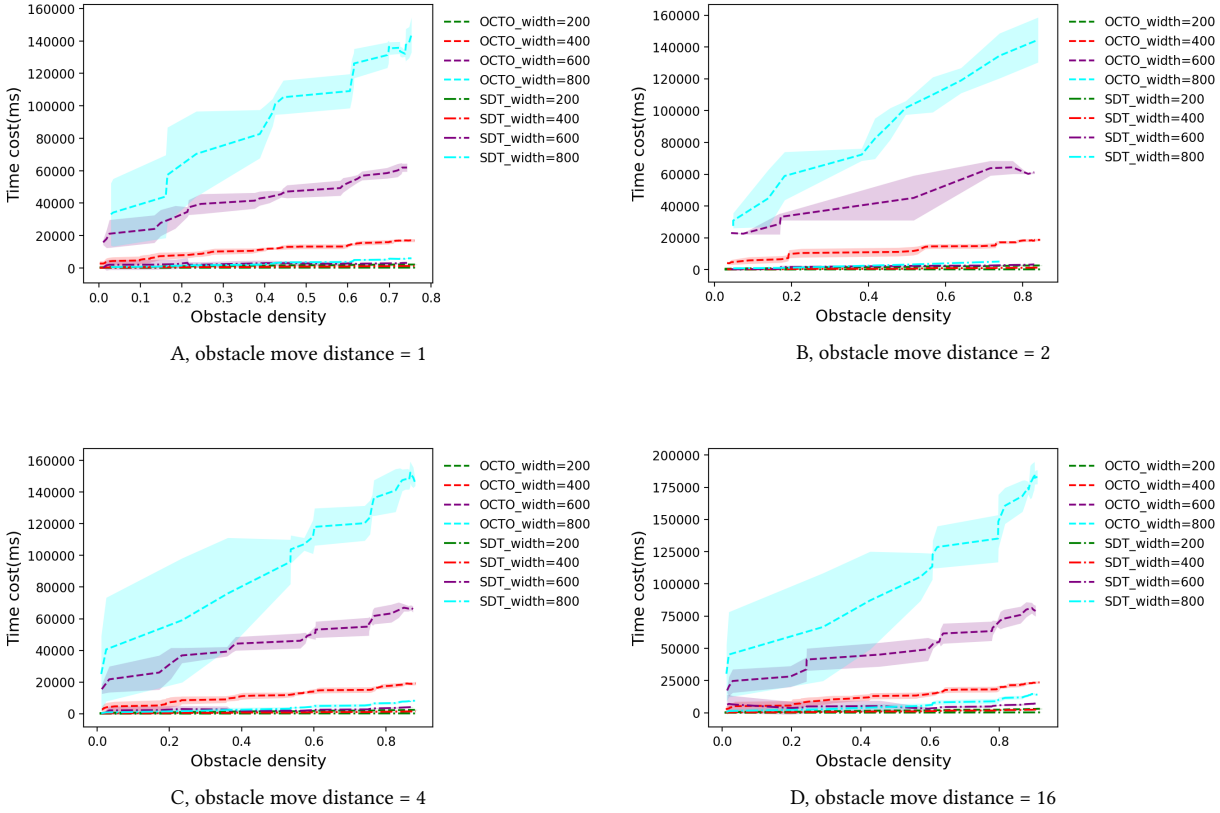


Fig. 11. Figures A, B, C and D show the average time cost of quadtree updates for various obstacle movement distances on the aforementioned random 2D maps.

quadtree/OctoMap when a dynamic obstacle moves. The related summaries are presented in Tables 1, 2, and 3. More details are provided in the Appendix. Specifically, the changes in initialization time cost with varying obstacle density are shown in Fig. 10; the changes in update time cost with varying obstacle density and obstacle movement distance are shown in Figs. 11 and 12; and the changes in LOS check time cost with varying obstacle density are shown in Fig. 13.

As shown in Table 1, in terms of initialization time cost, **SDT has lower time cost (27% ∼ 55% of quadtree/ OctoMap's time cost)**, because SDT downsamples the map, whereas quadtree and OctoMap do not. As illustrated in Fig. 10, both quadtree and OctoMap exhibit higher time costs when the obstacle density is low. Moreover, larger maps require more time to initialize, as a greater number of passable cells need to be set. This is because our algorithm needs to traverse all passable cells in the map and call the `update_cell_state` function, and in sparsely obstructed maps, a large number of passable cells must be updated. It is feasible to default all nodes as passable; however, this will cause a lot of time to initialize when there are lots of unpassable cells in the map. Considering the diversity of maps, some of which contain more passable cells while others contain more unpassable cells, initializing all nodes as either passable or unpassable is equivalent in terms of initialization efficiency.



A, obstacle move distance = 1

B, obstacle move distance = 2

C, obstacle move distance = 4

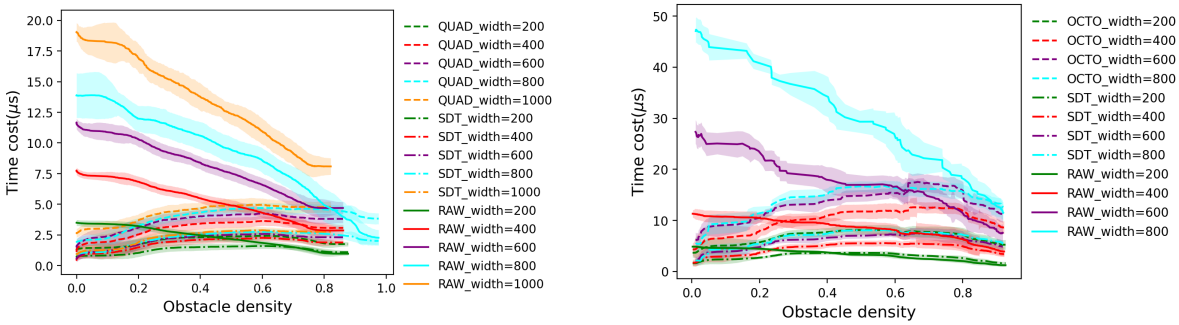
D, obstacle move distance = 16

Fig. 12. Figures A, B, C and D show the average time cost of OctoMap updates for various obstacle movement distances on the aforementioned random 3D maps.

Table 1. Average initialize time cost (s) of quadtree/OctoMap and SDT

Map width	200	400	600	800	1000
Quadtree(2D)	0.011	0. 051	0.133	0.234	0.384
SDT(2D)	0.0056	0.021	0.067	0.094	0.135
OctoMap(3D)	3.73	45.72	180.83	346.84	-
SDT(3D)	1.41	11.52	60.23	93.32	-

As shown in Table 2, similar to the initialization time cost, **SDT also has lower update time cost compared to quadtree/OctoMap (5% ~ 16% of quadtree/OctoMap's time cost)**, due to its downsampled map representation. As illustrated in Figs. 11 and 12, larger maps require more time to update, as they contain more cells that need updating for the same obstacle density (all obstacles move randomly in our experiments). Furthermore, the update



A, time cost comparison (2D)

B, time cost comparison (3D)

Fig. 13. Figures A and B show the average time cost of LOS checks using quadtree/OctoMap and SDT on the aforementioned random 2D and 3D maps, respectively.

Table 2. Average update time cost (ms) of quadtree/OctoMap and SDT

Map width	200	400	600	800	1000
Quadtree(2D)	5.44	13.97	24.77	34.55	36.82
SDT(2D)	0.86	2.08	3.30	4.48	6.14
OctoMap(3D)	$1.52 * 10^3$	$7.8 * 10^3$	$3.55 * 10^4$	$1.19 * 10^5$	-
SDT(3D)	131.12	939.37	2987.15	6084.02	-

Table 3. Average time cost of LOS check (μs)

Map width	200	400	600	800	1000
Traditional LOS(2D)	1.66	4.07	6.72	8.87	11.63
Quadtree(2D)	1.87	2.35	3.01	3.14	3.97
SDT(2D)	1.16	1.35	1.58	1.63	1.84
Traditional LOS(3D)	3.79	8.88	15.89	27.12	-
OctoMap(3D)	4.02	7.20	9.32	15.38	-
SDT(3D)	2.78	3.08	4.05	6.57	-

time cost increases as the obstacle movement distance grows, since larger movements require updating the states of more tree nodes. However, because SDT downsamples the map, it has a smaller tree, resulting in lower update time cost than quadtree/OctoMap. In addition, the update time cost of SDT increases more slowly compared to that of quadtree/OctoMap, also due to its use of a downsampled map representation.

In Table 3, we compare the traditional LOS check, LOS checks accelerated by quadtree/OctoMap blocks, and LOS checks accelerated by SDT blocks. Both SDT and quadtree/OctoMap block-accelerated LOS checks have lower time cost compared to the traditional LOS check. However, because SDT introduces a minimum block

Table 4. Average number of visits of grids/blocks during LOS checks

Map width	200	400	600	800	1000
Quadtree(2D)	14.32/13.12	14.95/13.90	15.82/14.81	16.12/15.12	17.79/16.81
SDT(2D)	20.55/3.74	17.13/5.10	16.69/5.76	16.21/6.20	16.27/6.56
OctoMap(3D)	21.40/21.59	21.78/21.03	25.52/24.86	34.90/33.65	-
SDT(3D)	25.82/7.51	21.22/8.75	21.19/11.99	33.58/15.94	-

Table 5. Average time cost of path plannings (ms)

Method	Theta*	LazyTheta*	RRT	RRT*
Traditional LOS	304.00	77.20	16.56	36.63
Quadtree	169.90	35.25	15.45	32.11
SDT	85.72	8.70	7.63	15.11

width and merges blocks, it has fewer but larger blocks, resulting in fewer blocks visited during LOS checks. **LOS checks accelerated by SDT blocks visit fewer blocks than those with quadtree/OctoMap (28% ∼ 44% of quadtree/OctoMap's)**, as shown in Table 4, even LOS checks accelerated by SDT visit more points, because jumping over a block is more time-consuming than a grid state check. So LOS checks accelerated by SDT blocks incur a lower time cost compared to those accelerated by quadtree/OctoMap blocks (**LOS checks accelerated by SDT have a time cost that is 40% ∼ 60% of that with quadtree/OctoMap, and 22% ∼ 70% of that with the traditional LOS check**).

We also studied the performance of LOS checks accelerated by SDT blocks as map scale and obstacle density change. As shown in Fig. 13, the traditional LOS check is very time-consuming when the map has low obstacle density, as it requires many cell state checks. The time cost of the traditional LOS check decreases as obstacle density increases. In contrast, LOS checks accelerated by SDT blocks and quadtree/OctoMap blocks have lower time cost, as they avoid many cell state checks by jumping over blocks.

When the obstacle density approaches 1, the traditional LOS check, SDT-accelerated LOS check, and quadtree/OctoMapaccelerated LOS check exhibit similar time costs, since there are almost no blocks to jump over and the cost is determined primarily by the number of cells intersected by the line. Moreover, the time cost of all methods decreases as the obstacle density approaches 1, since the likelihood of finding long collision-free lines becomes lower in denser environments.

As shown in Fig. 13, LOS checks accelerated by SDT blocks perform better in maps with low obstacle density (i.e., sparse obstacles) or large-scale maps. This is because, in such maps, SDT is more likely to generate larger blocks, and larger blocks result in fewer block and cell visits during LOS checks. Consequently, LOS checks accelerated by SDT incur lower time cost. On the other hand, SDT is not suitable for small maps with high obstacle density, as there are fewer blocks available for acceleration.

4.2 Experiments on Real World Maps

In this section, we evaluate whether our method can accelerate several classical path planning algorithms, including Theta*, LazyTheta*, RRT, and RRT*, on real dynamic maps. The maps are obtained using the RTAB-SLAM algorithm (Labbé and Michaud 2018) and real-world sensor data. The map is updated using real-time data

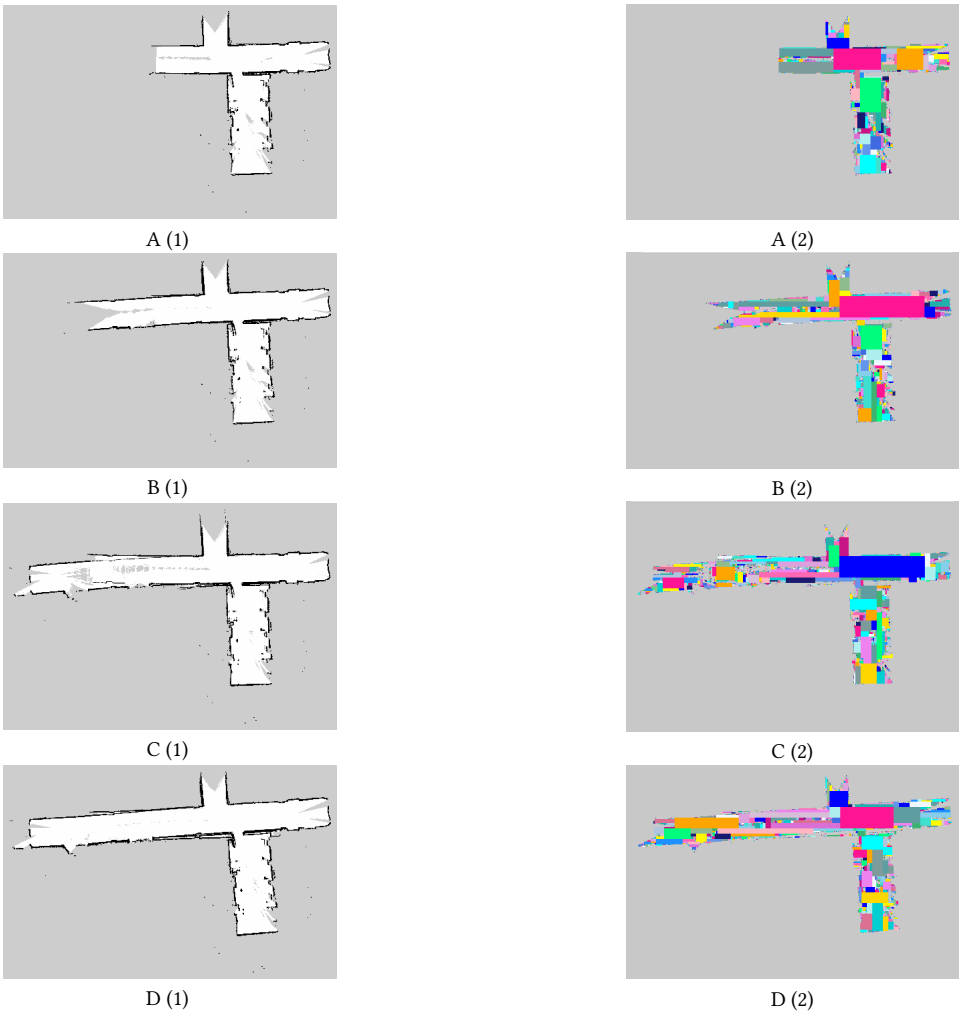


Fig. 14. These figures show several maps in chronological order from our experiment, along with their corresponding merged blocks. The maps are shown on the left, and the corresponding merged blocks are shown on the right.

during the experiment; thus, it is dynamically changing. An example of dynamic maps and their corresponding merged blocks is shown in Fig. 14. During map updates, path planning is performed with randomly selected start and target states (both of which are passable and connected by at least one feasible path) every 5 seconds, using both traditional LOS checking and LOS checking accelerated by SDT or quadtree/OctoMap. The resulting path planning time costs are summarized in Table 5.

As shown in Table 5, both quadtree and SDT blocks can reduce the time cost of path planning compared to using the traditional LOS check, but path planning using our LOS check incurs the lowest time cost. This is because, as demonstrated in the previous section, LOS checks accelerated by SDT blocks are more efficient than those accelerated by quadtree/OctoMap blocks or the traditional LOS check. In future work, we plan to extend our simulations to include non-cubic obstacles and to test our algorithm in scenarios without static obstacles.

5 Discussion and Conclusion

Motivated by the fact that the time cost of LOS checking significantly impacts the overall performance of path and motion planning, Yao et al. previously proposed JOB (Jump Over Block) (Z. Yao et al. 2023), which partitions the passable space into blocks to accelerate LOS checking in large-scale maps with sparse obstacles. By skipping cell state checks within blocks, JOB substantially reduces the time cost of LOS checks, particularly in environments containing many large blocks. However, JOB is limited to static maps and cannot efficiently handle dynamic environments, as all blocks must be updated whenever the map changes.

To overcome this limitation, we introduce the SDT—a dimension-independent data structure inspired by quadtree and OctoMap—that enables efficient updates in dynamic maps while maintaining the block-based acceleration mechanism. By integrating SDT into JOB, we extend it to dynamic environments and propose JODB, an efficient LOS checker designed for dynamic maps.

Compared to quadtree and OctoMap, SDT downsamples the map before initialization to reduce the time cost of initialization and updates, and uses a decision tree to merge blocks until every block is locally maximal. Thus, SDT has lower initialization and update time cost, and LOS checks accelerated by SDT incur lower time cost than those accelerated by quadtree and OctoMap.

In the Results section, we assess the performance of JODB from three perspectives: (1) the time cost of LOS checks accelerated by SDT blocks; (2) the time cost of initializing and updating SDT, which is critical in dynamic environments; and (3) how SDT accelerates path planning in real-world maps. Since existing methods for reducing the time cost of LOS checks focus on reducing the number of LOS checks or avoiding the use of LOS checks, and as far as we know, no method focuses on reducing the time cost of the LOS check itself. Therefore, we compare JODB with LOS checks accelerated by quadtree and OctoMap. The results illustrate that JODB incurs lower time cost compared to LOS checks accelerated by quadtree and OctoMap. Compared with quadtree and OctoMap, SDT requires less initialization and update time. More importantly, its ability to significantly reduce the cost of LOS checking leads to considerable improvements in path planning efficiency.

It is important to acknowledge the limitations of JODB. First, SDT does not maximize all possible blocks, as the size of each block is restricted to sums of powers of two, which leads to lower efficiency compared to JOB, where block sizes are unrestricted. In other words, JODB trades some LOS checking efficiency for adaptability to dynamic environments. Moreover, the acceleration effect of JODB decreases as obstacle density increases or map scale decreases.

In future work, we plan to investigate more memory-efficient data structures and selective update strategies to further improve the scalability and efficiency of JODB in large-scale, high-dimensional dynamic environments.

References

J. Bialkowski, S. Karaman, M. Otte, and E. Frazzoli. 2013. "Efficient collision checking in sampling-based motion planning." In: *Algorithmic Foundations of Robotics X: Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics*. Springer, 365–380.

J. E. Bresenham. 1965. "Algorithm for computer control of a digital plotter." *IBM Systems Journal*, 4, 1, 25–30.

D. Brewer and N. R. Sturtevant. 2018. "Benchmarks for Pathfinding in 3D Voxel Space." *Symposium on Combinatorial Search (SoCS)*.

K. Daniel, A. Nash, S. Koenig, and A. Felner. Sept. 2010. "Theta*: any-angle path planning on grids." *J. Artif. Int. Res.*, 39, 1, (Sept. 2010), 533–579.

J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. 1996. *Computer Graphics: Principles and Practice*. Addison-Wesley.

S. Han, L. Wang, Y. Wang, and H. He. 2022. "A dynamically hybrid path planning for unmanned surface vehicles based on non-uniform Theta* and improved dynamic windows approach." *Ocean Engineering*, 257, 111655.

D. Harabor and A. Grastien. 2013. "An optimal any-angle pathfinding algorithm." In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 23, 308–311.

D. Harabor, A. Grastien, D. Öz, and V. Aksakalli. May 2016. "Optimal any-angle pathfinding in practice." *J. Artif. Int. Res.*, 56, 1, (May 2016), 89–118.

K. Hauser. 2015. "Lazy collision checking in asymptotically-optimal motion planning." In: *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2951–2957.

A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. 2013. "OctoMap: An efficient probabilistic 3D mapping framework based on octrees." *Autonomous robots*, 34, 189–206.

I.-B. Jeong, S.-J. Lee, and J.-H. Kim. 2019. "Quick-RRT*: Triangular inequality-based implementation of RRT* with improved initial solution and convergence rate." *Expert Systems with Applications*, 123, 82–90.

Y. Ji, Y. Tanaka, Y. Tamura, M. Kimura, A. Umemura, Y. Kaneshima, H. Murakami, A. Yamashita, and H. Asama. 2019. "Adaptive Motion Planning Based on Vehicle Characteristics and Regulations for Off-Road UGVs." *IEEE Transactions on Industrial Informatics*, 15, 1, 599–611. doi:10.1109/TII.2018.2870662.

Z. Jian, S. Zhang, S. Chen, Z. Nan, and N. Zheng. 2021. "A global-local coupling two-stage path planning method for mobile robots." *IEEE Robotics and Automation Letters*, 6, 3, 5349–5356.

M. Labbé and F. Michaud. Oct. 2018. "RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation." *Journal of Field Robotics*, 36, (Oct. 2018), 416–446. doi:10.1002/rob.21831.

B. H. Meng, I. S. Godage, and I. Kanj. 2022. "RRT*-Based Path Planning for Continuum Arms." *IEEE Robotics and Automation Letters*, 7, 3, 6830–6837.

K. Meng, Y. Jia, H. Yang, F. Niu, Y. Wang, and D. Sun. 2020. "Motion Planning and Robust Control for the Endovascular Navigation of a Microrobot." *IEEE Transactions on Industrial Informatics*, 16, 7, 4557–4566. doi:10.1109/TII.2019.2950052.

S. Nayak, M. Paton, and M. W. Otte. 2023. "A Heuristic-Guided Dynamical Multi-Rover Motion Planning Framework for Planetary Surface Missions." *IEEE Robotics and Automation Letters*, 8, 5, 2542–2549.

S. Oh and H. W. Leong. 2017. "Edge n-level sparse visibility graphs: Fast optimal any-angle pathfinding using hierarchical taut paths." In: *Tenth Annual Symposium on Combinatorial Search*.

S. Popov, I. Georgiev, P. Slusallek, and C. Dachsbacher. 2013. "Adaptive quantization visibility caching." In: *Computer Graphics Forum* 2pt4. Vol. 32. Wiley Online Library, 399–408.

N. Sturtevant. 2012. "Benchmarks for Grid-Based Pathfinding." *Transactions on Computational Intelligence and AI in Games*, 4, 2, 144–148. http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.

J. Vörös. 2001. "Low-cost implementation of distance maps for path planning using matrix quadtrees and octrees." *Robotics and Computer-Integrated Manufacturing*, 17, 6, 447–459.

J. Yao and J. Zhao. 2025. "Decoupling Sampling and Planning Frameworks for Redundant Manipulators Motion Planning Under End-Effector and Performance Constraints." *IEEE Transactions on Industrial Informatics*, 21, 7, 5492–5503. doi:10.1109/TII.2025.3556026.

Z. Yao, W. Wang, J. Zhang, Y. Wang, and J. Li. 2023. "Jump Over Block (JOB): An Efficient Line-of-Sight Checker for Grid/Voxel Maps With Sparse Obstacles." *IEEE Robotics and Automation Letters*, 8, 11, 7575–7582.

C. Zhang, Y. Tang, and H. Liu. 2019. "Late line-of-sight check and partially updating for faster any-angle path planning on grid maps." *Electronics Letters*, 55, 12, 690–692.

C. Zhang, Y. Tang, L. Zhou, and H. Liu. 2019. "Late line-of-sight check and prioritized trees for path planning." *Applied Sciences*, 9, 17, 3448.

Received 26 December 2025; accepted 22 March 2026