IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 June 2023, accepted 4 July 2023, date of publication 7 July 2023, date of current version 25 July 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3293125

RESEARCH ARTICLE

Grid Graph Reduction for Efficient Shortest Pathfinding

CHAN-YOUNG KIM AND SANGHOON SULL, (Senior Member, IEEE)
School of Electrical Engineering, Korea University, Seoul 02841, South Korea

Corresponding author: Sanghoon Sull (sull@korea.ac.kr)

This work was supported by Samsung Electronics Company Ltd., under Grant IO201209-07873-01.

ABSTRACT Single-pair shortest pathfinding (SP) algorithms are used to identify the path with the minimum cost between two vertices in a given graph. However, their time complexity can rapidly increase as the graph size grows. In this paper, we propose a pattern-based blocking algorithm in a grid graph (PBGG) that iteratively blocks or reduces free space vertices that do not require exploration. The blocking process is based on the neighbors of each vertex and utilizes 3×3 binary pattern matching. The time complexity of blocking is $O(I \cdot \lceil |V|/C \rceil)$, where $|V|$ is the number of vertices, I is the maximum number of iterations, and C is the number of parallelized cores. PBGG significantly reduces the total computation time when utilized to preprocess an input grid graph before applying existing SP algorithms. It also guarantees that if a minimum-cost path exists in the original graph, then the SP algorithms can find at least one path with the same minimum cost in the reduced graph. The proposed method is formulated by convolutions that can be easily implemented using machine learning platforms, such as PyTorch. Experimental results show that when PBGG can significantly reduce the total computation time when employed in conjunction with SP algorithms such as A$*$ and Jump Point Search. On average, PBGG reduces the total computation times by 71% for A and 41% for Jump Point Search, compared to the times taken by the SP algorithms alone.

INDEX TERMS Blocking, convolution, dead-end/avoidable vertices, nonblockable vertices, pattern matching, shortest path problem.

I. INTRODUCTION

Pathfinding in two-dimensional (2D) grid graphs plays a crucial role in various applications, such as car navigation [13], [17], [22], digital entertainment [12], [15], [18], and robotics [9]. The most basic pathfinding problem is the single-pair shortest pathfinding (SP) problem, which determines the optimal path with the minimum cost between two vertices in a given graph.

The SP problem with start and end vertices is generally solved by search algorithms such as Dijkstra's algorithm [23] or A$*$ algorithm [16]. However, these approaches necessitate recursive visits to the free-space vertices to calculate the precise distances from the start vertex. Their execution times rapidly increase when the graph size increases.

To expedite those algorithms, we propose a preprocessing method called pattern-based blocking on grid graphs (PBGG). It aims to reduce the number of free-space vertices in a graph before the pathfinding and ensure the optimality of a path. Fig. 1 illustrates the result of reducing a 32×32 input grid graph using the proposed method. The figure reveals that the free- space vertices considered during the search are significantly reduced, whereas the optimal path can still be found.

The proposed method can also be applied even when separate vertex costs are given, for example, to prevent collision with obstacles [20], [21]. In particular, the proposed method can quickly determine whether a path exists.

The proposed approach examines the local 3×3 vertices of each vertex to determine their ability to be blocked, specifically, whether an obstruction interferes with the connection between the start and end vertices. This analysis is performed concurrently across all vertices. The blocking of adjacent vertices can depend on each other, which is difficult to consider in parallel computation. We identify and omit the interconnected vertices from blocking to mitigate

The associate editor coordinating the review of this manuscript and approving it for publication was Walter Didimo.

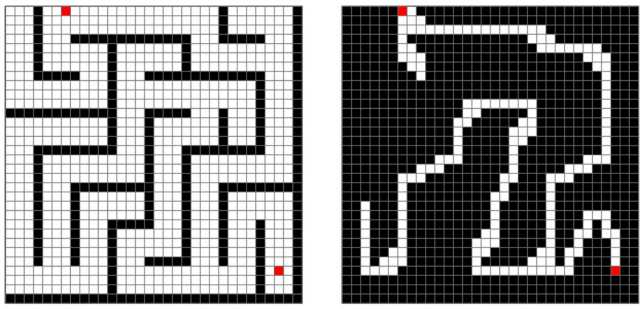

FIGURE 1. Proposed blocking method for a single-pair shortest pathfinding (SP) problem. (a) Input 32 × 32 grid graph consisting of obstacles (black), free space (white), and start/end (red) vertices. (b) Reduced graph after detecting free-space vertices that do not require exploring after 30 blocking iterations, treated as obstacle vertices. The significantly reduced graph is input to SP algorithms.

this problem. Thus, this approach effectively reduces the number of free-space vertices while ensuring that the connection between the start and end vertices remains unimpeded. All operations can be readily and efficiently implemented through machine learning platforms, such as PyTorch [2].

By employing PBGG to reduce the grid graphs and utilizing the SP algorithm on the reduced graph, the total execution time can be significantly reduced. When we experimented with three types of graphs (maze, room, and random) with various sizes, we observed the average computation times of Dijkstra's, A*, and JPS algorithms decreased by 67%, 71%, and 41%, respectively. The explored path may change, but the path cost remains optimal. Moreover, as the size of the graph becomes larger, the blocking effect becomes more prominent. Blocking applies to various problems that require SP and significantly reduces search time in most cases.

In summary, we introduce an algorithm to block a subset of free-space vertices with the following contributions:

1) The proposed method aims to reduce the number of candidate vertices to be explored, reducing SP computation time while guaranteeing an optimal path between the start and end vertices.
2) For various grid graphs with representative search algorithms, such as Dijkstra's, A*, and JPS, we observed an average time reduction of 67%, 71%, and 41%, respectively.
3) The algorithm can also be applied to cases where vertex costs are given to avoid collision with obstacles.

The remainder of the paper is organized as follows. Section II introduces the preliminaries. Next, Section III presents the proposed algorithm, and Section IV explains the detailed implementation. Then, Section V details the experimental results. Finally, Section VI concludes the paper.

II. PRELIMINARIES

A. SHORTEST PATHFINDING IN THE GRID GRAPH

A 2D grid graph is defined by a set of vertices corresponding to a 2D lattice and the edges between adjacent vertices. In pathfinding problems, vertices belong to obstacles or free space, and obstacle vertices are not connected to other vertices by edges. A graph that satisfies these conditions is called a *grid graph for a SP problem*, and we refer to it as a *grid graph* in this paper. Depending on the presence of diagonal edges, a vertex can have up to four neighbors (4N) or up to eight neighbors (8N).

A path in a graph between two vertices is a sequence of edges connecting them. The cost $C(p)$ of a path p is defined as the sum of costs assigned to all vertices and edges in the path. The single-pair SP methods must determine the optimal path if it exists or determine its nonexistence (i.e., no path exists between two vertices). The assigned edge cost is often determined using the Euclidean distance between two vertices. The cost associated with each vertex may or may not be given and is typically used to minimize the possibility of collision with obstacles [20], [21].

B. ALGORITHMS FOR SHORTEST PATHFINDING

Many search-based algorithms have been used to solve single pair shortest path (SP) problems in grid graphs. In this paper, we introduce Dijkstra's algorithm [23] and then explore the differences between different algorithms based on it.

The algorithm comprises several key components: an explored vertex set, an open set containing vertices that have not been explored yet but are reachable, a cost list, and a parent list. The input for the algorithm can take the form of either an adjacency matrix or a binary image, both representing the adjacency relationships among the vertices of a grid graph. In each iteration, the algorithm extracts a vertex from the open set that is closest to the start vertex. This vertex is then added to the explored vertex set, and its adjacent vertices that have not been explored are added to the open set. The value of the cost list associated with the vertex is updated to the length of the shortest path from the start vertex to the vertex. The value of the parent list corresponding to the vertex is also updated to the vertex that precedes it in the shortest path. If the vertex extracted from the open set is the end vertex, the intermediate parents are traced to obtain the final path between the start/end vertices. If the open set becomes empty before the end vertex is encountered, it signifies that there is no feasible path between the start and end vertices in the grid graph.

The computation time of a search algorithm is mainly determined by the time taken to select a vertex to explore from the open set and the number of vertices that need to be explored. As the size of the grid graph increases, both the size of the open set and the number of vertices to explore generally increase, resulting in a significant increase in execution time.

Search-based algorithms share these common structures, but they can be distinguished by utilized data structure, criteria for selecting vertices to explore, criteria for adding adjacent vertices to the open set, and other factors. We can broadly categorize them into three groups: those that accelerate individual steps, those that change the order of exploration, and those that reduce the number of vertices to explore.

The first group encompasses various methods, such as using a 2D array to quickly recognize visited vertices and determine the vertex to search [9]. Another method is to adopt a hybrid approach that combines a 2D array and a heap structure [3]. Finally, multiple vertices can be traversed concurrently [1]. These enhancements improve the efficiency of each step within the search algorithm, while the overall behavior of the algorithm remains unchanged.

The second group of pathfinding algorithms changes the order of exploration by incorporating handcrafted or learned heuristic cost. A* algorithm is one such algorithm that uses a heuristic cost to estimate the cost of the cheapest path from each vertex to the end vertex. This allows A* algorithm to incorporate the end vertex's location into the vertex selection process, resulting in faster pathfinding in most cases. Unlike Dijkstra's algorithm, which determines the order of vertex exploration independently of the end vertex, the A algorithm takes the end vertex into account. This results in faster pathfinding in most cases, as A* algorithm can focus on exploring vertices that are more likely to be on the shortest path to the end vertex. Due to the similarity in structure with Dijkstra's algorithm, the A algorithm can be utilized in conjunction with other groups of algorithms. Learning-based approaches [8], [14] aim to utilize artificial neural networks to acquire improved heuristics. However, the learning-based approaches do not guarantee the optimality of paths.

The last group, where PBGG belongs, reduces the number of vertices to traverse, such as Jump Point Search (JPS) [5], [6], Iterative monotonically bounded A* (IMBA*) [25], and Index-based A* Search (IBAS) [24]. The JPS algorithm, which is considered as one of state-of-the-art does not search neighboring vertices at each iteration. Instead, it performs straight-line jumps in each direction until it encounters a vertex immediately before the obstacle or a vertex adjacent to the obscured area. JPS explore those vertices, referring them as jump points. Although JPS can only consider distance-based edge costs without vertex costs, it is considered one of the state-of-the-art algorithms in such situations. IMBA* [25] employs an initial pathfinding process within a reduced search space and gradually expands the search area based on the results. This approach is effective when the shortest path between start/end vertices contains high-cost vertices and is primarily concentrated around the vicinity of these vertices. In such cases, a small number of iterations can significantly reduce the number of explored vertices. IBAS [24] estimates the upper and lower bounds of the distance between vertices to reduce the search space. This algorithm is specifically designed for directed acyclic graphs, thereby is not suitable for grid graphs.

C. NOTATIONS

In this paper, we propose a blocking method to identify and remove the unnecessary free-space vertices during the tree search. We notate the initial grid graph and the reduced grid graph after the k^{th} iteration of blocking as $\mathrm{G}^0 = \left(\mathrm{V}^0, \mathrm{E}^0\right)$ and $\mathrm{G}^k = \left(\mathrm{V}^k, \mathrm{E}^k\right)$, where $\mathrm{V}^0, \mathrm{E}^0, \mathrm{V}^k$ and E^k represent vertex and edge sets of the initial and reduced graphs, respectively. The set of neighbors in G^k at a free-space vertex v is denoted as $\mathrm{N}^k(v)$.

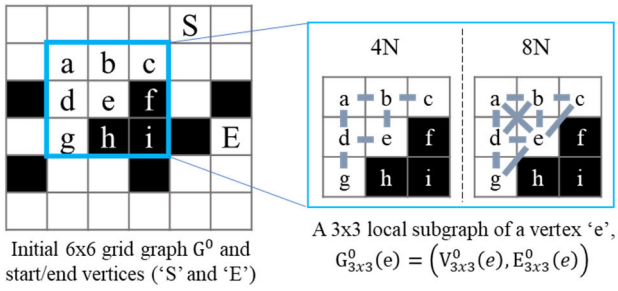

FIGURE 2. Grid graph for the single-pair shortest pathfinding (SP) problem and 3 × 3 local subgraph of vertex *e*. Free-space vertices are white, obstacle vertices are black, and start/end vertices are marked as S and E, respectively. For simplicity, the edges are not shown in the figures of this paper.

In addition, for a free-space vertex v in the grid graph, we let $\mathrm{V}^k_{3\times3}(v)$ denote the set of vertices corresponding to the 3×3 neighborhood of v in the 2D lattice. We let $\mathrm{E}^k_{3\times3}(v)$ denote the set of edges that connect the vertices in $\mathrm{V}^k_{3\times3}(v)$ in the grid graph. We refer to the subgraph defined by $\mathrm{V}^k_{3\times3}(v)$ and $\mathrm{E}^k_{3\times3}(v)$ as a "3×3 local subgraph" of v and define it as $\mathrm{G}^k_{3\times3}(v) = (\mathrm{V}^k_{3\times3}(v), \mathrm{E}^k_{3\times3}(v))$. A grid graph and 3×3 local subgraph are illustrated in Fig. 2.

III. ALGORITHM

The overview of the proposed algorithm is illustrated in Fig. 3. PBGG detects vertices that do not require exploration and blocks them iteratively (i.e., replacing a portion of the free-space vertices with obstacle vertices). Consequently, SP algorithms can be utilized to the reduced graph. The initial grid graph G^0 and the start/end vertices are taken as inputs, and G^1 is generated by blocking in the first iteration. Similarly, after the k^{th} iteration, G^{k+1} is generated from G^k. The exploration for pathfinding is performed by the conventional SP algorithm.

We define two types of vertices that can be blocked while keeping at least one optimal path between the start and end vertices based on the local subgraph of each vertex. The identification of these categories is achieved by matching the local subgraphs with predefined patterns using the parallel convolution operation. Since the time complexity of the parallel convolution operation is determined by the size and number of patterns, we focus on examining the local 3×3 subgraph, which encompasses the neighboring vertices of each vertex and represents the minimum subgraph size. This approach allows us to identify blockable vertices efficiently while ensuring the preservation of optimal paths.

Blocking a vertex changes local 3×3 subgraphs for adjacent vertices; thus, the blocking process depends on those vertices, making the parallelization challenging. We detect and do not block all the related vertices to overcome this.

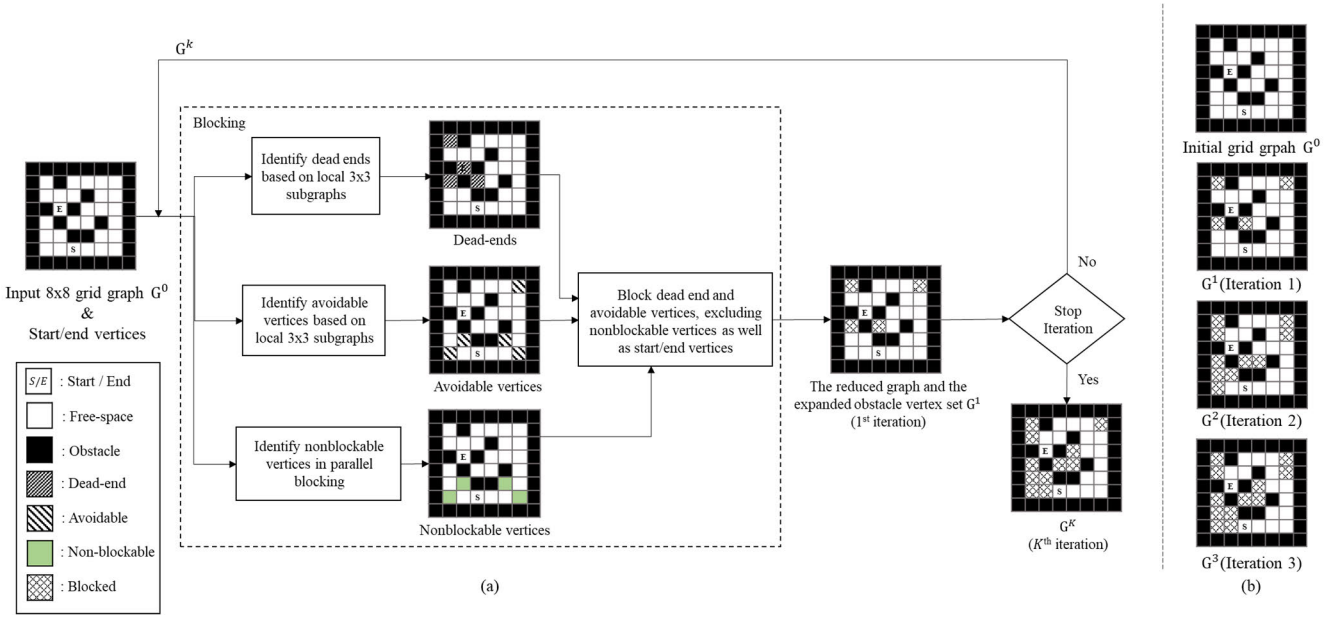

FIGURE 3. Blocking illustration. (a) Algorithm overview. (b) Intermediate graphs during blocking iterations. The input comprises an initial grid graph and start/end vertices. The algorithm recognizes dead ends and avoidable vertices, which can be individually excluded from pathfinding. Nonblockable vertices can cause a disconnection between the start/end vertices due to the limitation of parallel processing. Nonblockable vertices and start/end vertices are recognized and excluded from the blocking targets. Thus, part of the free-space vertices changes to obstacle vertices, and at least one optimal path between the start and end vertices is guaranteed. This process is repeated until the stop iteration is reached.

This detection also be processed on the local 3 × 3 subgraphs of each vertex, making the entire blocking locally processed. Blocking is repeated until the stopping iteration. Then, an existing SP method is applied to the reduced grid graph obtained through adequate iterations for fast SP.

A. DETECTION OF BLOCKABLE VERTICES

In this paper, a blockable vertex is a free-space vertex that can be obstructed from a grid graph without disconnecting the start and end vertices. We introduce two types of blockable vertices: dead end and avoidable, then introduce another type of blockable vertex with two or more neighbors.

Definition 1) A free-space vertex v is a *dead end* in G^k if the degree of v in G^k is 0 or 1.

The vertex b in Fig. 4 is a typical example of a dead end. They do not belong to the optimal path if they are not start/end vertices.

Definition 2) A free-space vertex v is *avoidable* in the grid graph G^k if, for any pair of vertices (u, w) in $N^k(v)$, a path p between them exists in the local 3 × 3 subgraph $G^k_{3\times3}(v)$ and w such that $C(p) \leq C(u\text{-}v\text{-}w)$, where '$u$-$v$-$w$' denotes the path consisting of u, v and w and $C(\cdot)$ denote the assigned path cost.

The vertex d in Fig. 4 is an avoidable vertex which can be replaced with e (4N) or excluded from the path (8N). Certainly, avoidability can be defined not only in 3 × 3 subgraphs but also in arbitrary subgraphs in a similar manner. However, when dealing with larger subgraphs, a significant increase in the number and size of required kernels for implementation poses a problem of decreased computational speed. Therefore, we focus on path comparison within the 3 × 3 subgraph.

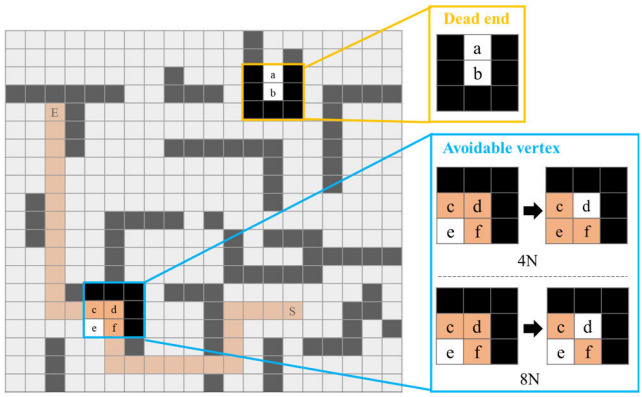

FIGURE 4. Examples of a dead end *b*, avoidable vertex *d*, and path between the start/end vertices (orange). A dead end has only one neighbor. An avoidable vertex can be replaced or excluded from the path.

B. FORMULATION OF BLOCKABLITY

Excessive blocking can hinder or disrupt the connection between start and end vertices. Therefore, we show that blocking dead ends and avoidable vertices preserves the optimality of pathfinding.

Theorem 1) If an optimal path between two vertices (s, e) exists in G^k, the same path exists in G^{k+1}, where G^{k+1} is the grid graph obtained by blocking a dead end v ($\neq s$ or e) from G^k.

Proof) Let the sole neighbor of v denoted by u. If v is included in the optimal path p^*, u will be visited twice. This contradicts the definition of a path, implying that v cannot be part of p^*. Therefore, p^* exists in G^{k+1}.

Theorem 2) If an optimal path exists between two vertices (s, e) in G^k, a path with the same cost exists in G^{k+1}, where G^{k+1} is the grid graph obtained by blocking an avoidable vertex v ($\neq s$ or e) from G^k.

Proof) If a vertex v is a part of the optimal path p^* between vertices s and e in the graph G^k, then there exist two adjacent vertices of v on p^*. We denote the one closer to s as u and the other one closer to e as w. The optimal path p^* can be represented as a sequence $[p_1, p_2, p_3]$, where p_1, p_2, and p_3 are optimal paths between (s, u), (u, w), and (w, e) respectively. From definition 2, another optimal path q ($\neq p_2$) between (u, w) where $C(q) \leq C(p_2)$ exists. Consequently, an alternative path $\dot{p}^*(\neq p^*)$ between (s, e) where $C(\dot{p}^*) = C(p^*)$ is obtained as $[p_1, q, p_3]$ and exists in both G^k and G^{k+1}. On the other hand, if v is not included in the optimal path in G^k, an optimal path in G^k still exist in G^{k+1}. Therefore, in both cases, there is an optimal path with the same cost in both G^k and G^{k+1}.

C. DETECTION OF NONBLOCKABLE VERTICES

Blocking each vertex sequentially takes time proportional to the graph size; thus, we aim to block all such vertices in parallel. However, as depicted in Fig. 5, if two adjacent vertices d and e are independently blocked due to the parallel local 3×3 operation, disconnection occurs between two other vertices c and f, which are not blockable. The disconnection can affect the optimal path between start/end vertices. Thus, we checked all possible configurations in Fig. 6. The green vertices represent avoidable vertices that cause indirection or disconnection between nearby free-space vertices due to parallel blocking based on 3×3 windows.

If such graphs are present as subgraphs within an input graph, these green vertices can potentially disrupt the optimal path between the start/end vertices by forcing detours or causing disconnections. Thus, green vertices should not be blocked and are called nonblockable vertices in this paper.

However, discerning nonblockable vertices based on 4×4 or 5×4 subgraphs in Fig. 6 is inefficient. The time complexity of parallelized convolution is proportional to the number and size of the kernels. For example, unlike the limited 512 possibilities in a 3×3 subgraph, a 4×4 subgraph consists of 65536 different configurations, requiring even more kernels. Furthermore, the size of each kernel also varies almost twice as much, resulting in significantly slower processing compared to the preceding steps based on the 3×3 subgraph.

To handle this, we defined α-type vertices based on local 3×3 subgraphs, as illustrated in Fig. 7(a) and (b) for 4N and 8N, respectively. In Fig. 7(c) and (d), all configurations from Fig. 6 include pairs of $\alpha-$type/start/end vertices connected through one nonblockable vertex. Therefore, we first detect α-type vertices based on the local 3×3 subgraphs and identify the vertices adjacent to two or more α-type/start/end vertices as nonblockable.

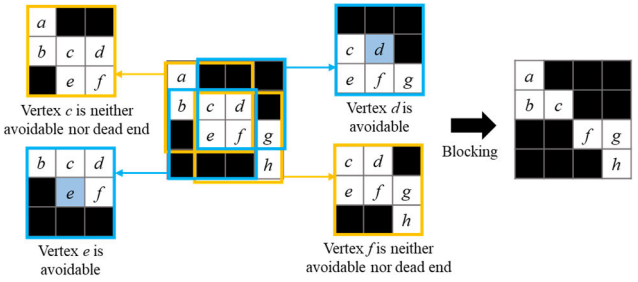

FIGURE 5. Example illustrating parallel blocking of multiple avoidable vertices *d* and *e* in a 4 × 4 grid graph disconnecting free-space vertices *c* and *f* with a four-neighbor (4N) configuration.

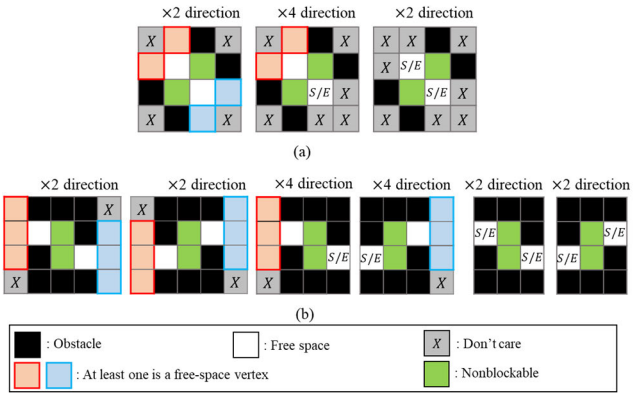

FIGURE 6. All configurations where two vertices blocked by parallel blocking disconnect two adjacent vertices: (a) four- and (b) eight-neighbor configurations. In parallel blocking, the green vertices are avoidable and blocked, disconnecting other free-space vertices or the start/end (S/E) vertices. Graphs with x2 direction can be rotated 0° and 90°. Graphs with x4 direction can be rotated by 0°, 90°, 180°, and 270°.

Some vertices that meet this criterion may not be nonblockable, making some normally avoidable vertices not blocked. Nevertheless, this method enables us to efficiently filter out all nonblockable vertices using a parallel operation based on 3×3 windows. For convenience, we refer to all vertices filtered out using this method as nonblockable vertices throughout the paper.

The resulting detection of α-type vertices and nonblockable vertices for 4N and 8N are presented in Fig. 8. The results are different for the same input graph. Additionally, as the blocking process is iterative, the previously identified nonblockable vertices could be blockable in the next iteration.

IV. IMPLEMENTATION

A. VERTEX DETECTION USING PATTERN MATCHING

A grid graph can be represented as a binary image, where 0 and 1 indicate obstacles and free spaces, respectively. PBGG takes the modified image as an input by assigning a value of -1 or 1 to each vertex, indicating whether it is an obstacle or a free-space vertex, respectively. The resulting

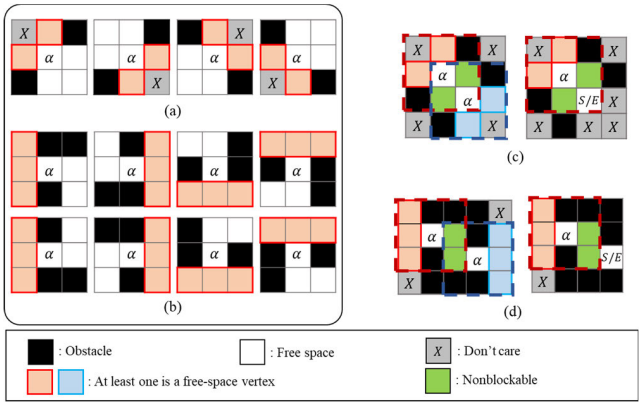

FIGURE 7. All local 3 × 3 subgraphs defining α-type vertices and examples of the combination of these local 3 × 3 subgraphs without collision between free spaces and obstacles: (a) four-neighbor (4N) configuration, (b) eight-neighbor (8N) configuration, (c) α -type vertices detected in the graphs of Fig. 6(a) for 4N, and (d) α -type vertices detected in the graphs of Fig. 6(b) for 8N.

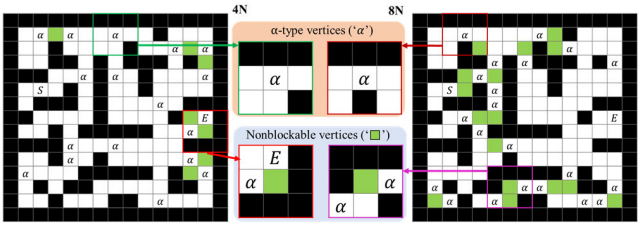

FIGURE 8. Results of detecting α-type vertices and nonblockable vertices (green) with their respective local 3 × 3 subgraphs in four-neighbor (4N, left) and eight-neighbor (8N, right) configurations. First, α−type vertices are identified by their local 3 × 3 subgraphs. Vertices adjacent to two or more α -type/start/end vertices are nonblockable.

binary image is referred to as a grid map in this paper. Similarly, a local 3×3 subgraph of a vertex can be represented as a 3 × 3 binary pattern. A 3 × 3 binary pattern centered at a dead-end vertex is called a dead-end pattern. Avoidable and α-type patterns are also defined similarly. Note that those patterns are mutually disjointed.

We describe the step-by-step process of pattern matching. The inputs are a grid map X and the convolutional kernels consisting of several 3 × 3 weights $\{W_c\}$ $(c = 1, \ldots, C)$ and bias terms $\{b_c\}$ $(c = 1, \ldots, C)$. The convolutional kernels are designed to satisfy the following condition for each 3 × 3 pattern set Z (i.e., dead-end patterns, avoidable patterns, or α-type patterns):

$$\max_{z' \in Z^c} \left(\max_{c \in \{1,\ldots,C\}} \left(W_c \circledast z' + b_c \right) \right) \leq 0 < \min_{z \in Z} \left(\max_{c \in \{1,\ldots,C\}} \left((W_c \circledast z) + b_c \right) \right) \quad (1)$$

where Z^C is the complement set of Z, and is $\circledast$ the convolutional operation.

Initially, the grid map X is expanded by adding a border of -1 values in all four directions. For the grid map with padding, denoted as X', the convolution operation yields a correlation map Y. Each channel of Y is calculated as follows:

$$Y[c, :, :] = W_c \circledast X' + b_c \quad (2)$$

$Y[c, h, w]$ is the correlation between W_c and $X'[h\text{-}1{:}h + 2, w\text{-}1{:}w + 2]$ with the addition of b_c to the computed value. From (1), if any value of tensor Y at location (h, w) is positive, it indicates that the 3 × 3 binary pattern of the grid graph at location (h, w) matches one of the patterns in the set Z.

Therefore, for every location in Y, we verify whether the maximum value exceeds 0. These processes are executed on parallelized computing systems, such as graphics processing units (GPUs). The corresponding pseudocode for this is provided in Appendix. VI-A.

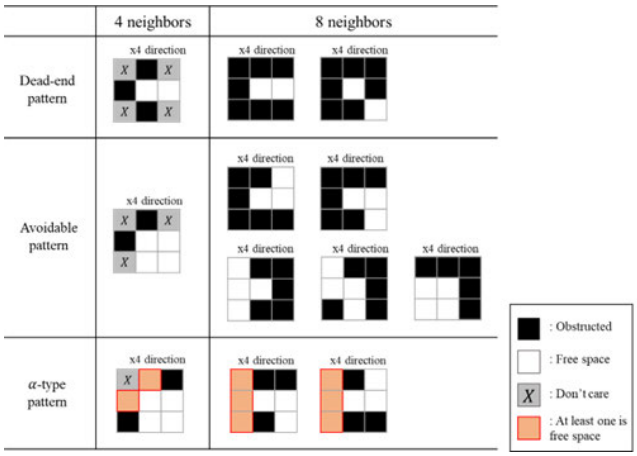

FIGURE 9. All 3 × 3 patterns identified through pattern matching.

We list the 3 × 3 pattern sets to be identified, as shown in Fig. 9. The patterns simply and completely express each binary pattern set, namely, dead-end, avoidable, and α-type patterns. We can determine the suitable convolution kernels satisfying (1) based on the patterns. Each pattern is the combinations of 'obstacle', 'free space', 'don't care' and 'at least ∼'. Convolution weights and biases corresponding to each pattern can be created according to the following rules.

1) 'Obstacle' corresponds to -1, 'free space' corresponds to 1, and 'don't care' corresponds to 0 in W_c.
2) For m vertices satisfying 'at least n is free space (or obstacle)', the weights are assigned as 1 (or -1). Other weights are multiplied by $(m + 1)$.
3) The bias b_c is calculated as $-|W_c|_1 + 2(m\text{-}n) + 1$ where $|\cdot|_1$ is a sum of absolute values.

The rules are designed to monotonically increase the convolution output according to the similarity between w_c and the patterns, while the b_c serves to threshold the similarity.

Furthermore, the time complexity of parallel convolution is related to the number of kernels. To reduce the number of kernels required, three modifications are made, as shown in Fig. 10. First, the dead-end patterns in the 4N configuration are represented using the 'at least∼' condition. Second, in the 8N configuration, both the dead-end patterns and certain avoidable patterns (patterns in the first row in Fig. 9) can

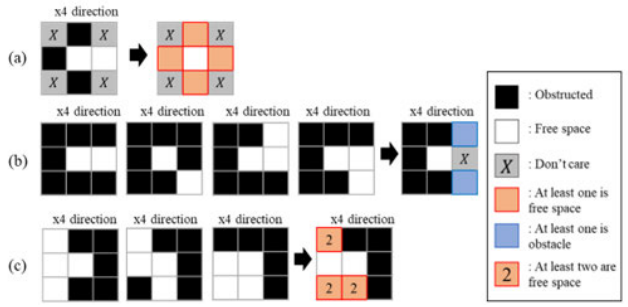

FIGURE 10. 3 × 3 patterns represented with less kernels.

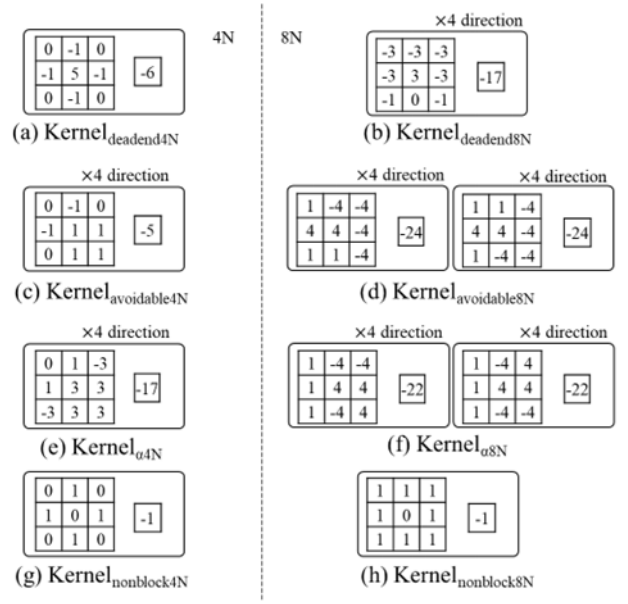

FIGURE 11. Convolutional weights and bias terms for identifying specific types of vertices with four or eight-neighbor (4N or 8N) configurations. (a, b) Dead-end patterns. (c, d) Avoidable patterns. (e, f) α -type patterns. (g, h) Nonblockable vertices.

be recognized together through the common kernels. Third, the remaining avoidable patterns in the 8N configuration are expressed using the 'at least∼' condition.

Nonblockable vertices can also be expressed by patterns that have 'at least two' α-type or start/end vertices in their vicinity. Each convolution kernels tailored for each type of pattern are illustrated in Fig. 11. The pseudocode of the blocking process is described in Appendix. VI-B.

B. INCORPORATING COSTS FOR VERTICES

For various objectives, additional positive costs can be assigned to vertices. Costs for vertices can be represented as an image of the same size as the grid map, called a *cost map*. When a cost map is provided, the process of verifying avoidable vertices becomes more intricate.

To determine whether a vertex v is avoidable, it is necessary to examine the paths connecting each neighboring pair of v and ascertain if a path passing through v has the lowest cost among them. However, cost comparisons for all possible paths through pattern matching become computationally expensive, as the number of required kernels is excessively high.

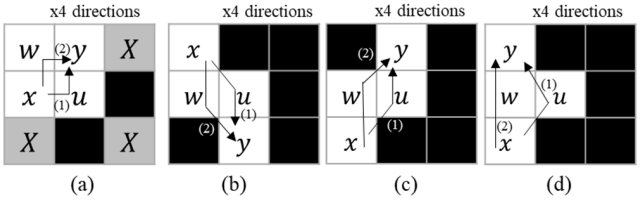

FIGURE 12. Cases requiring a cost comparison between two paths to classify the vertex u as avoidable: (a) four-neighbor (4N) and (b, c, d) eight-neighbor (8N) configurations. They can be rotated by 90°, 180°, and 270°.

Therefore, we propose a conservative but efficient criterion for determining avoidable vertices. It consists of two conditions. Firstly, whether the 3 × 3 pattern in the grid map belongs to the pattern set shown in Fig. 12, which defines avoidable vertices in the absence of vertex costs as discussed in the previous subsection. It enables the determination of the avoidability of the center vertex u by comparing only two paths between x and y indicated in Fig. 12. Therefore, the second condition to determine the vertex u's avoidability is that the path passing through the center vertex u (Path 1) should not be cheaper than the alternative path (Path 2). This criterion may overlook some of avoidable vertices. However, less blocking is solely an efficiency-related concern and does not compromise the shortest path. We address the process for each 4-neighbor and 8-neighbor configuration separately.

We begin by describing the case of 4-neighbor configuration in Fig. 12(a). We can determine u as avoidable based on two conditions, 'the local 3 × 3 pattern in the grid map corresponds to Fig. 12(a)' and '$C(x\text{-}w\text{-}y) \leq C(x\text{-}u\text{-}y)$'. The verification of the first condition follows the same methodology as described in the previous subsection for detecting avoidable vertices. Checking '$C(x\text{-}w\text{-}y) \leq C(x\text{-}u\text{-}y)$' is equivalent to comparing two vertex costs, w and u: '$C(w)$ - $C(u)$ ≤ 0'. It can be implemented by applying the convolution kernels, '$\text{Kernel}_{\text{cost4N}}$', to the cost map. The parameters in kernels are shown in Fig. 13(a). The detailed implementation is described in Appendix. VI-C.

Similarly, in the 8-neighbor configuration, we determine the avoidability of u based on two conditions, 'the 3 × 3 pattern surrounding vertex u belongs to Fig. 12(b-d)' and '$C(x\text{-}w\text{-}y) \leq C(x\text{-}u\text{-}y)$'. However, in Fig. 12(b) and (c), only the vertex costs of w and u need to be compared when comparing the costs of the paths, whereas in Fig. 12(d), the difference in edge costs should also be considered. To distinguish between Fig. 12(b-c), and (d), we apply '$\text{Kernel}_{\text{grid}-\text{case}}$' to the grid map where the parameters of kernels are depicted in Fig. 13(b). For the pattern corresponding to Fig. 12(d), twice the cost difference between the diagonal and directional edges are reflected to the cost comparison. In all cases in Fig. 12(b-d), the cost comparison between w and u is conducted by '$\text{Kernel}_{\text{cost8N}}$' in Fig. 13(b). The detailed

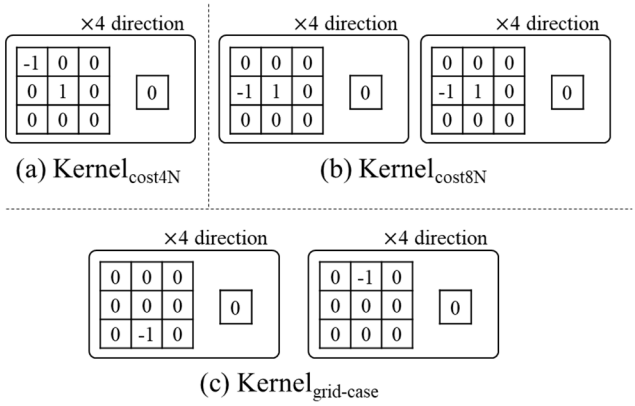

FIGURE 13. Convolutional Kernels consisting of weights and bias when costs are assigned to vertices. (a) For the cost comparison between two vertices with four neighbors (4N). (b) For the cost comparison between two vertices with eight neighbors 8N. (c) For differentiating the cases of Fig. 12(b-c) and (d) for 8N.

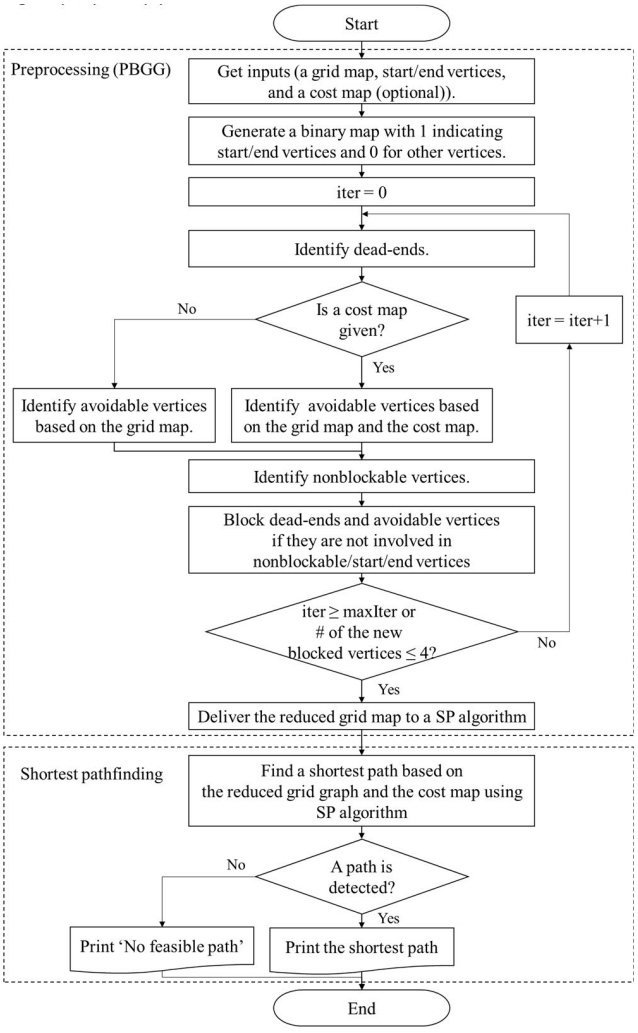

FIGURE 14. Flowchart for shortest pathfinding using PBGG.

implementation is provided in Appendix. VI-D. By replacing the avoidable pattern identification, PBGG is modified as Appendix. VI-E.

C. END CONDITIONS

Continuing blocking until no more blockable vertices is possible but can be computationally expensive and inefficient. Therefore, the maximum number of iterations for blocking is set to $(H + W)//8$, where (H, W) represents the size of the grid map. In addition, if the number of blocked vertices in a single iteration is less than four, the blocking is terminated to prevent unnecessary computation.

D. COMBINATION WITH SP ALGORITHM

Finally, the reduced grid map become an input for SP algorithms. The SP process with PBGG can be summarized as shown in Fig. 14. The interface of SP algorithm and combination of PBGG and SP algorithm are outlined in Appendix VI-F and VI-G, respectively.

E. TIME COMPLEXITY OF BLOCKING

Entire blocking is composed with the convolution operations and the logical operations, yielding the time complexity of blocking to be $O\left(IKhw \cdot \lceil |V|/C \rceil\right)$, where I is the the maximum number of iterations, K is the number of utilized convolution kernels, $(h,\ w)$ are the size of a convolution kernel, C is the number of parallelized cores, and $|V|$ is the number of vertices in the grid graph. For the configurations without vertex costs, K is set to 10 for 4N and 21 for 8N. With vertex costs, K becomes 14 for 4N and 37 for 8N. As PBGG uses 3×3 patterns, both h, w are 3. By excluding the constant components, the time complexity is rewritten as $O\left(I \cdot \lceil |V|/C \rceil\right)$. Since I would be less than C, PBGG reduces the number of free-space vertices efficiently.

V. EXPERIMENTS

The experiments were conducted using an Intel Core i7-7700 with a 3.60 GHz CPU and an NVIDIA GeForce GTX 1080Ti graphics processing unit (GPU), with Python v.3.10 and PyTorch v.1.12. The existing search methods were run on the CPU while the blocking process was run on the GPU.

We considered three types of grid graphs to evaluate the algorithm's performance: maze, room, and random. For the random graph, obstacles were randomly inserted in the remaining area after the path of the maze graph was fixed as a free space. Examples of each type are presented in Fig. 15. The problems were categorized into three sizes, 128×128, 256×256, and 512×512, according to the grid graph size. We generated 100 grid graphs for each type.

A. COMPUTATION TIMES FOR PATHFINDING

To demonstrate the feasibility of PBGG, we selected three representative SP algorithms: Dijkstra's, A*, and JPS, which serve as foundational algorithms for many SP problems. Both A* and Dijkstra's algorithms were also implemented to accept binary grid maps as inputs, like JPS.

We compared the total computation times of applying SP algorithm directly to the input graph, and PBGG and SP algorithms sequentially. When vertex costs were not pro-

TABLE 1. Average total computation time change due to blocking with four-neighbor configuration (SP Only →PBGG+SP, time (s)).

Time (s)		Maze			Room			Random		
		128×128	*256×256*	*512×512*	*128×128*	*256×256*	*512×512*	*128×128*	*256×256*	*512×512*
No cost map	*Dijkstra*	0.102→0.046	0.384→0.090	1.672→0.269	0.149→0.081	1.027→0.327	5.512→1.473	0.040→0.053	0.162→0.115	0.727→0.345
	*A**	0.112→0.046	0.428→0.091	1.975→0.276	0.064→0.058	0.426→0.170	2.064→0.609	0.025→0.047	0.126→0.098	0.550→0.278
	JPS	0.122→0.058	0.448→0.113	1.999→0.344	0.088→0.103	0.402→0.369	1.338→1.182	0.041→0.057	0.165→0.133	0.625→0.392
Cost map	*Dijkstra*	0.120→0.074	0.524→0.165	2.064→0.579	0.197→0.126	1.547→0.624	8.278→2.907	0.041→0.061	0.159→0.124	0.770→0.430
	*A**	0.142→0.074	0.566→0.168	2.320→0.593	0.104→0.094	0.742→0.357	3.600→1.406	0.039→0.061	0.154→0.122	0.762→0.430

TABLE 2. Average total computation time change due to blocking with eight-neighbor configuration (SP only→PBGG+SP, time (s)).

Time (s)		Maze			Room			Random		
		128×128	*256×256*	*512×512*	*128×128*	*256×256*	*512×512*	*128×128*	*256×256*	*512×512*
No cost map	*Dijkstra*	0.205→0.055	0.828→0.135	3.626→0.301	0.351→0.125	2.687→0.650	15.737→3.313	0.154→0.138	0.944→0.775	5.802→4.655
	*A**	0.326→0.056	1.397→0.136	6.369→0.308	0.145→0.076	0.948→0.289	5.682→1.227	0.024→0.030	0.192→0.169	1.077→0.869
	JPS	0.047→0.050	0.169→0.127	0.726→0.280	0.027→0.048	0.100→0.111	0.321→0.280	0.017→0.026	0.100→0.111	0.450→0.450
Cost map	*Dijkstra*	0.234→0.066	0.975→0.145	4.649→0.501	0.418→0.156	3.214→0.836	19.133→4.324	0.163→0.149	0.965→0.824	5.866→4.840
	*A**	0.377→0.068	1.514→0.149	7.507→0.520	0.257→0.117	1.821→0.527	10.009→2.262	0.155→0.128	0.815→0.689	4.783→3.926

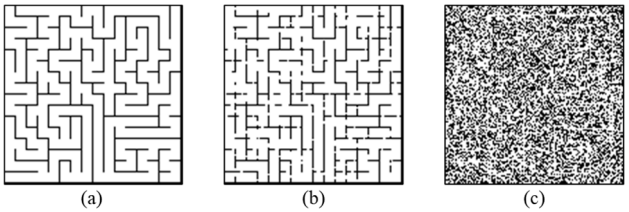

FIGURE 15. Three grid graph types: (a) maze, (b) room, and (c) random.

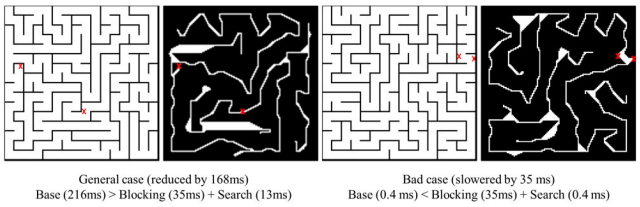

FIGURE 16. Example cases where blocking results in faster or slower pathfinding times. Blocking is redundant if the start/end vertices (red X-marked) are enclosed with each other.

vided (no cost map), we observed that the average runtime decreased by 68%, 78%, and 41% for Dijkstra's, A*, and JPS, respectively. When vertex costs were provided with a cost map, we observed a decrease in average runtime of 66% and 67% for Dijkstra's and A*, respectively. In all cases, the cost of the resulting optimal path remained unchanged depending on the blocking application.

To provide a more detailed analysis, we present the results for various graph sizes, types, and numbers of neighbors in Tables 1 and 2. While the average time decreased overall, this is not always the case. Blocking may be unnecessary in problems that can be solved quickly with only the SP algorithm. For example, when the start and end vertices are very close, as depicted in Fig. 16, A* and JPS algorithms are fast enough, especially when the path between the two vertices is nearly straight (e.g., the random type graph). Furthermore, using the 8N approach, the JPS algorithm operates

TABLE 3. Average total computation time change to recognize nonexistance of a path (SP only→PBGG+SP, time (s)).

		Dijkstra	A*	JPS
4N	128×128	0.088→0.042	0.115→0.043	0.121→0.048
	256×256	0.435→0.100	0.539→0.104	0.601→0.125
	512×512	1.861→0.364	2.340→0.277	2.508→0.261
8N	128×128	0.177→0.046	0.318→0.049	0.046→0.043
	256×256	0.964→0.100	1.776→0.106	0.223→0.093
	512×512	4.037→0.328	7.483→0.344	0.898→0.306

faster overall, particularly in these scenarios. In such cases, blocking increased the computation time of the SP.

Despite these limitations, the blocking method offers significant advantages. First, the blocking is a fast operation that takes around 1 ms per iteration. For cases where blocking resulted in slower performance, the difference was only 0.1 s, while the difference in the most accelerated case was almost 15 s.

Second, when no connectivity exists between the start and end vertices, the time required for recognition can be greatly reduced. Table 3 reveals that the required time was reduced by an average of 87%, 93%, and 80% for Dijkstra's, A*, and JPS, respectively. The computation times in this scenario are proportional to the number of free-space vertices connected to the start vertex. PBGG effectively reduces the number of free-space vertices and consequently achieves significant time savings compared to directly applying the SP algorithms that require tracking connections with the start vertex.

B. VARIATIONS ACCORDING TO ITERATIONS

To conduct a comprehensive analysis, we conducted various observations by employing PBGG in conjunction with A* while varying the blocking iterations, in the absence of a cost map. We address various aspects, including the computation time, the number of visited vertices, and the number of free-space vertices within the grid graph.

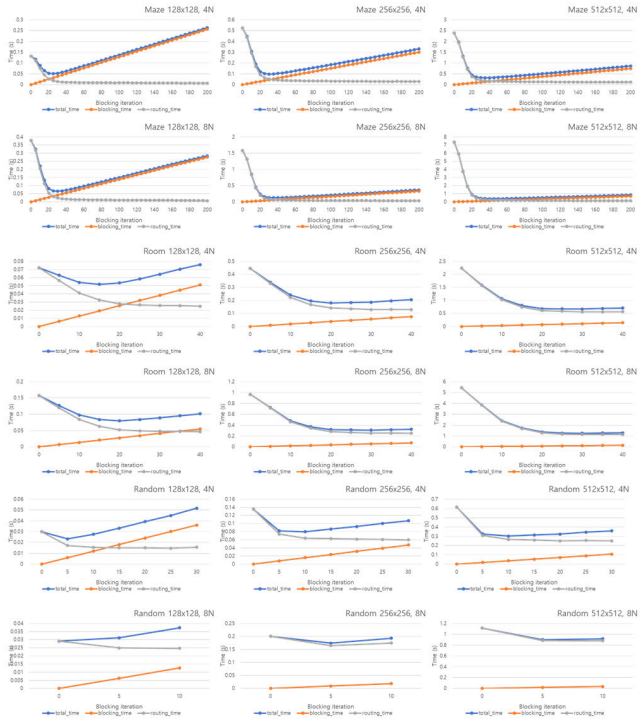

FIGURE 17. Average times for blocking (orange), pathfinding (grey) and total computation (blue) according to blocking iterations. The number of iterations affects the average time for pathfinding, and its sensitivity decreases as the graph size increases. The iterations required for blocking are less than 50 and sufficiently fast compared to the total reduction in time. 4N: four-neighbor configuration, 8N: eight-neighbor configuration.

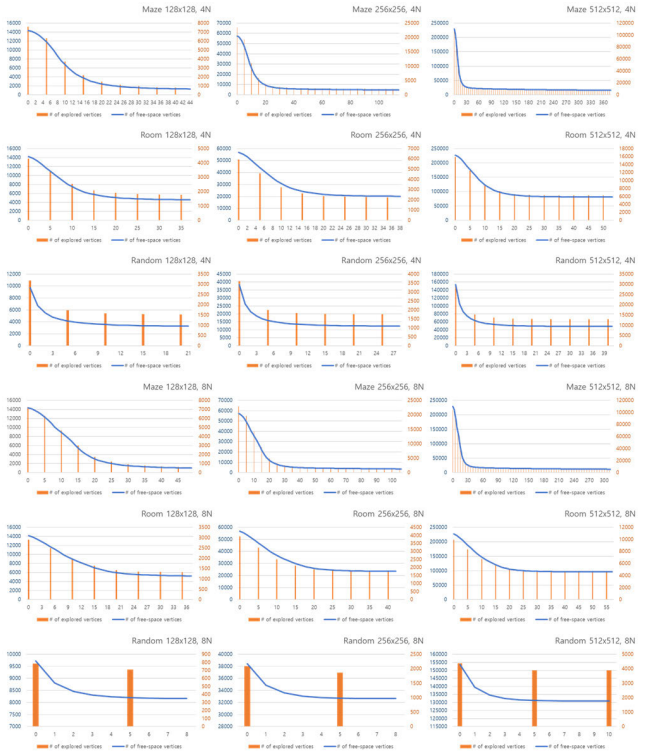

FIGURE 18. Average numbers of free-space vertices (blue) and explored vertices by A⁎ algorithm (orange) according to blocking iterations.

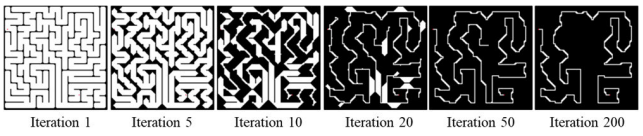

FIGURE 19. Grid map change by blocking iteration.

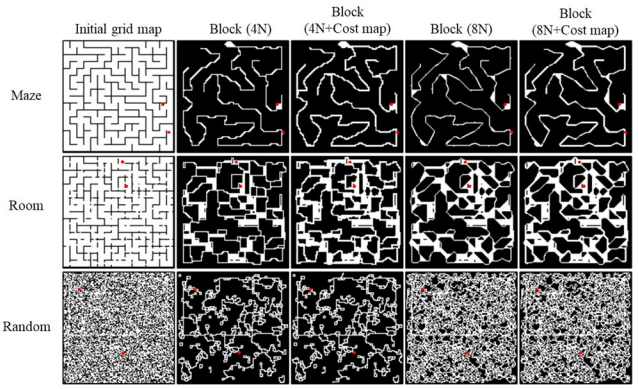

FIGURE 20. Reduced 128 × 128 grid maps for each problem type with two points to connect marked as red.

Fig. 17 illustrates the average computation times for pathfinding according to the iteration. The total computation time, time for blocking, and time for pathfinding are denoted by the blue, orange, and gray lines, respectively. Although the maximum iteration was set to 200, different termination iterations were observed for each type due to another end condition: 'the number of blocked vertices is 4 or fewer.' On average, for graph sizes of 128 × 128, 256 × 256, and 512 × 512, the respective execution times for a single iteration of blocking were approximately 1.3ms, 1.5ms, and 3.3ms in 4N configuration. In the case of 8N, the execution times were 1.3ms, 2ms, and 4.4ms, respectively. It is evident that even after performing 20 iterations, the execution time remains less than 0.1 seconds.

In maze and room graphs, the shortest time is recorded at approximately 20-35 iterations. The V-shaped trend stabilizes as the graph size increases. On the other hand, for the random graphs, the benefits from blocking are relatively small, and in some cases, a little increase in the overall computation time is observed (20ms and 5ms for 128 × 128.4N and 8N). Based on these findings, it can be inferred that the selection of iterations is critical for maze/room graphs, and random graphs require fewer iterations to achieve satisfactory results.

Fig. 18 shows the average number of free-space vertices (blue) and the number of vertices visited by A* algorithm during pathfinding (orange) in the graph according to the iteration. We measured the average number of explored vertices every five iteration and observed a similar decreasing trend with the iteration. The number of vertices decreases with the iteration and tends to converge to a specific value. When considering 20 iterations as a benchmark, it can be

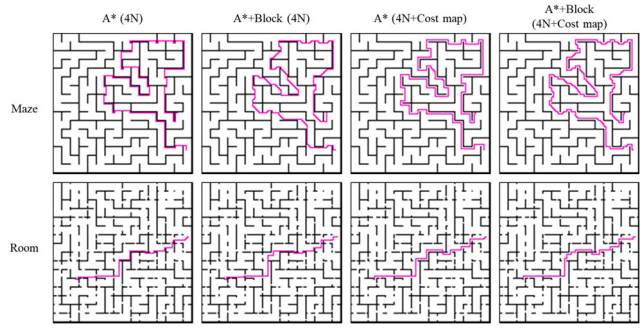

FIGURE 21. Examples of the paths with and without blocking. The proposed blocking method tends to keep the path away from the obstacle compared to the nonblocking method. However, the optimality of the path is preserved as blocking does not alter it.

observed that maze and room graphs experienced a reduction of approximately 80% and 60% in the number of free-space vertices, respectively. On the other hand, the blocking process in random graphs terminated more quickly, resulting in a nearly halved number of free-space vertices. An interesting observation is that, while the number of free-space vertices decreases, the number of explored vertices does not decrease significantly, and compared to other types of graphs, the number of explored vertices is much lower. This suggests that the random graph has paths that are nearly straight between the start and end vertices, and the shortest paths are also easily found in the original graphs. In other words, in random graphs, blocking concludes faster compared to other graph types, and a lower maximum iteration setting is sufficient.

Fig. 19 presents the change in the grid map according to iterations of blocking. It supposes that only 20 to 50 iterations are needed to reduce the free space sufficiently.

C. ADDITIONAL COMPARISON AND QUALITATIVE VALIDATION

PBGG is an algorithm that decreases the number of free-space vertices while ensuring the optimality of the shortest path. To the best of our knowledge, research focusing on reducing the search range while preserving path optimality is relatively limited. One such algorithm is known as iteratively monotonically bounded A* algorithm (IMBA*) [25].

Most of SP algorithms explore all paths with costs lower than the estimated cost of the shortest path between the start and end vertices. In contrast, IMBA* which is based on A* algorithm initially reduces the search range to the vicinity of the start and end vertices, thereby decreasing the number of paths that require pre-examination. Then, it progressively expands the search range based on the results of A* algorithm. However, in graphs with a significant number of obstacle vertices between two vertices, IMBA* must execute A* algorithm multiple times to find the shortest path in a wide area of free space vertices while avoiding obstacle vertices. This leads to slower performance. Upon observing the average execution times of IMBA*, it was found to be approximately three times slower than when using A* algorithm alone, and around thirteen times slower compared to 'PBGG+A*'. On the other hand, PBGG is robust to the shape of obstacles, requiring only a single path search.

Examples of blocking various graphs are provided in Fig. 20, where free-space vertices are sufficiently reduced. Furthermore, Fig. 21 demonstrates the property of the blocking method in keeping the path away from the obstacle, which can be desirable for certain applications, such as robots or vehicles. The presented examples suggest that the proposed blocking method can improve the overall pathfinding performance and can be beneficial for various practical scenarios.

VI. CONCLUSION

We proposed a method to reduce free space in a grid graph for efficient single-pair SP problems and guarantee that the SP algorithm can find at least one optimal path. This approach involves parallelly detecting various vertices using pattern-matching techniques based on their respective local 3×3 subgraphs. The experiments found that the proposed approach reduced the total pathfinding time by an average of 71% for A$*$ algorithm across various problems and conditions. The proposed algorithm can be applied to tasks requiring efficient pathfinding on large graphs, such as for robots in a warehouse or virtual spaces for games.

APPENDIX A

The appendix provides pseudo codes to aid in understanding this paper. Algorithm 1 demonstrates the implementation details of pattern matching, which is consistently used in the Pattern-Based Grid Graph (PBGG).

Algorithm 1 Pattern Matching

```
def PatternMatching (X, K):
    # This function is operated by GPUs.
    # Inputs: X: Input map, (H, W)-shape tensor
    #         K: Convolution kernels consisting of weight and bias.
    # K.weight: (C, 3, 3)-shape tensor.
    #K.bias:C-shape tensor.

    # Pad boundaries of the grid map as obstacles
1:  X' =pad(X, padding_size=[1,1,1,1], value=−1)

    # Get correlation map between 3 × 3 subgraphs and 3 × 3 convolution kernel weights
2:  Y =convolution(X', K); # (C, H, W)

    # For every vertex whose 3 × 3 subgraph is matched at least one of kernel weights, assign 1.
3:  M = max(Y, dim=0); # (H, W)
4:  M =greater(M, 0);

5:  return M; # Result (H, W) tensor of pattern matching
```

Algorithm 2 PBGG When the Cost Map Is Not Given

def PBGG1(M, s, e, maxIter; convKernels):
This function is processed on GPUs.
Inputs: M: Grid map, (H, W)-shape binary tensor
s, e: start/end vertices
maxIter: the maximum number of iterations
Parameters: convkernels (4N: [$\text{Kernel}_{deadend4N}$, $\text{Kernel}_{avoidable4N}$, $Kernel_{\alpha 4N}$, $Kernel_{nonblock4N}$], 8N: [$\text{Kernel}_{\text{deadend8N}}$, $\text{Kernel}_{\text{avoidable8N}}$, $\text{Kernel}_{\alpha 8N}$, $\text{Kernel}_{\text{nonblock8N}}$])
M_s =MakeStartEndMap(s, e); # start/end is True, otherwise False.
i= 0;
while do
i= i+ 1;
Make (H, W) tensor, 1 is a deadend.
M_{deadend} = PatternMatching(M, $\text{Kernel}_{\text{deadend}}$);
Make (H, W) tensor, 1 is an avoidable vertex.
$M_{\text{avoidable}}$ = PatternMatching(M, $\text{Kernel}_{\text{avoidable}}$);
Make (H, W) tensor, 1 is an α-type vertex.
M_α = PatternMatching(M, Kernel_α);
Mark α-type/start/end vertices in one tensor.
$M_{\alpha+s}$ = logicalOR(M_s, M_α);
Count α-type/start/end vertices in each 3 × 3 subgraphs. If two or more exist, the vertex is nonblockable.
M_{nonblock} = PatternMatching($M_{\alpha+s}$, $\text{Kernel}_{\text{nonblock}}$);
Blockable: deadend or avoidable
M_{block} = logicalOR(M_{deadend}, $M_{\text{avoidable}}$);
But– nonblockable or start/end is excluded.
M_{block} = logicalAND(M_{block}, logicalNOT(logicalOR(M_{nonblock}, M_s)));
Apply blocking to the grid map
$M = \text{clip}(M - 2 \times M_{\text{block}}.\text{type(int)}, -1, 1)$;
# Check end condition
if endCondition(|M_{block}-M|, i, maxIter) **then**
break;
end if
end while
return M; # (H, W) tensor that the blocking is applied.

In Algorithm 2, we illustrate PBGG without a given cost map. When a cost map is given, the identification of avoidable vertices varies depending on the 4N and 8N

Algorithm 3 Identify Avoidable Vertices in 4N (w/ Vertex Costs)

def GetAvoidable4NwithVertexCost(M, C; $\text{Kernel}_{\text{cost4N}}$, $\text{Kernel}_{\text{avoidable4N}}$):
This function is processed on GPUs.
Inputs: M: Grid map, (H, W)-shape binary tensor
C: Cost map, (H, W)-shape tensor
Parameters: predefined 3 × 3 conv. kernels. ($\text{Kernel}_{\text{cost4N}}$ and $\text{Kernel}_{\text{avoidable4N}}$)
Condition1: obstacles are placed as Fig.13(a)
$M_{\text{grid-cond}}$ = convolution(M, $\text{Kernel}_{\text{avoidable4N}}$);
$M_{\text{grid-cond}}$ = greater($M_{\text{grid-cond}}$, 0); # (4, H, W)
Condition2: another path is equal to or cheaper than the center path.
$M_{\text{cost-cond}}$ = convolution(C, $\text{Kernel}_{\text{cost4N}}$);
$M_{\text{cost-cond}}$ = greater_equal($M_{\text{cost-cond}}$, 0); # (4, H, W)
Check subgraphs matched to 4 cases that satisfy both conditions
$M_{\text{avoidable}}$ = logicalAND($M_{\text{cost-cond}}$, $M_{\text{grid-cond}}$); # (4, H, W)
For all vertices whose 3 × 3 subgraph matched, assign 1 indicating avoidable vertices.
$M_{\text{avoidable}}$ = max($M_{\text{avoidable}}$, dim=0); # (H, W)
return $M_{\text{avoidable}}$; # (H, W) tensor, 1 is the avoidable

Algorithm 4 Identify Avoidable Vertices in 8N (w/ Vertex Costs)

def GetAvoidable8NwithVertexCost(M, C; $\text{Kernel}_{\text{cost8N}}$, $\text{Kernel}_{\text{avoidable8N}}$, $\text{Kernel}_{\text{grid-case}}$, λ):
This function is processed on GPUs.
Inputs: M: Grid map (H, W)-shape tensor.
C: Cost map (H, W)-shape tensor.
Parameters: predefined 3 × 3 conv. kernels. ($\text{Kernel}_{\text{cost8N}}$, $\text{Kernel}_{\text{avoidable8N}}$, $\text{Kernel}_{\text{grid-case}}$)
λ: 2×(diagonal edge cost – directional edge cost)
Condition1: obstacles are placed as Fig. 13 (b-d)
$M_{\text{grid-cond}}$ = convolution(M, $Kernel_{avoidable8N}$);
$M_{\text{grid-cond}}$ = greater($M_{\text{grid-cond}}$, 0); # (8, H, W)
Condition2: another path is equal to or cheaper than the center path.
$M_{\text{cost-cond}}$ = convolution(C, $Kernel_{\text{cost8N}}$) -λ· convolution(M, $\text{Kernel}_{\text{grid-case}}$);
$M_{\text{cost-cond}}$ = greater_equal($M_{\text{cost-cond}}$, 0); # (8, H, W)

Algorithm 4 *(Continued.)* Identify Avoidable Vertices in 8N (w/ Vertex Costs)

```
    # Check subgraphs matched to one of 8 cases that both conditions are satisfied
5:  M_avoidable = logicalAND(M_cost−cond, M_grid−cond); # (8, H, W)

    # For all vertices whose 3 × 3 subgraph matched, mark as 1 indicating avoidable vertices.
6:  M_avoidable = max(M_avoidable, dim=0); # (H, W)

7:  return M_avoidable; # (H, W) tensor, 1 is the avoidable
```

Algorithm 5 PBGG When the Cost Map Is Given

```
def PBGG2(M, s, e, C_vertex, maxIter; convKernels):
    # This function is processed on GPUs.
    # Inputs: M: Grid map (H, W)-shape tensor.
    #         s, e: Start/end vertices
    #         C_vertex: Cost map (H, W)-shape tensor.
    #         maxIter: the maximum number of iterations
    # Parameters: convKernels (4N: [Kernel_deadend4N, Kernel_avoidable4N, Kernel_α4N, Kernel_nonblock4N, Kernel_cost4N], 8N: [Kernel_deadend8N, Kernel_avoidable8N, Kernel_α8N, Kernel_nonblock8N, Kernel_cost8N, Kernel_grid−case])
    # Make (H, W) tensor, 1 is start/end vertex.
1:  M_s =MakeStartEndMap(s, e);
2:  i = 0;
3:  while do
4:      i= i+ 1;
        # Make (H, W) tensor, 1 is a deadend.
5:      M_deadend =PatternMatching(M, Kernel_deadend);

        # Make (H, W) tensor, 1 is an avoidable vertex.
6:      if config is 4N then
7:          M_avoidable = GetAvoidable4NwithVertexCost(M, C, Kernel_cost4N, Kernel_avoidable4N);
8:  else then # 8N
9:      M_avoidable = GetAvoidable8NwithVertexCost (M, C, Kernel_cost8N, Kernel_grid−case, Kernel_avoidable8N);
10: end if
        # Make a (H, W) tensor, 1 is an α-type vertex.
11:     M_α = PatternMatching(M, Kernel_α);
        # Mark α-type/start/end vertices in one tensor.
12:     M_α+s = logicalOR(M_s, M_α);

        # Count α-type/start/end vertices in 3 × 3 subgraphs. If two or more exist, the vertex is nonblockable.
13:     M_nonblock = PatternMatching(M_α+s, Kernel_nonblock);

        # Blockable: deadend or avoidable
14:     M_block = logicalOR(M_deadend, M_avoidable);
        # But– nonblockable or start/end is excluded.
15:     M_block = logicalAND(M_block, logicalNOT(logicalOR(M_nonblock, M_s)));

        # Apply blocking to the grid map
16:     M = clip(M − 2 × M_block.type(int), -1, 1);
17:     if endCondition(|M_block-M|, i, maxIter) then
18:         break;
19:     end if
20: end while
21: return M; # (H, W) tensor that the blocking applied.
```

Algorithm 6 Interface of SP Algorithm

```
def ShortestPathfinding(M, s, e, C_vertex, mode):
    # Inputs: M: Grid map
    #         s, e: start/end vertices
    #         C_vertex: vertex costs (default value is 0).
    #         mode: dijkstra, A*, JPS, ...
1:  if mode is 'dijkstra' then
2:      SP_method = Dijkstra;
3:  else if mode is 'A*' then
4:      SP_method = A*;
5:  else if mode is 'JPS' then
6:      SP_method = JPS;
7:  end if
8:  path = SP_method(M, s, e, C_vertex);
9:  return path; # list of vertices from s to e.
```

configurations, as described in Algorithms 3 and 5, respectively. Algorithm 7 combines these approaches to showcase PBGG with a given cost map. Algorithm 8 introduce the interface of SP algorithms. Finally, in Algorithm 9, we present the process that combines PBGG with the SP algorithm.

A. PATTERN MATCHING

See Algorithm 1.

B. PBGG (w/o vertex costs)

See Algorithm 2.

C. IDENTIFICATION OF AVOIDABLE VERTICES IN 4N (w/ vertex costs)

See Algorithm 3.

D. IDENTIFICATION OF AVOIDABLE VERTICES IN 8N (w/ vertex costs)

See Algorithm 4.

Algorithm 7 Improved SP Utilizing PBGG

```
def SP_with_PBGG(M, s, e, maxIter, C_vertex, mode):
    # Inputs: M: Initial grid map (H, W)-shape binary
    tensor. 1 is free-space, 0 is obstacle.
    #       s and e: start/end vertices
    #       maxIter: the maximum blocking iterations
    #       C_vertex: cost map
    #       mode: Dijkstra's / A* / JPS / ...
1:  M ← 2M- 1; # -1 is obstacle, 1 is free-space.

    # Preprocessing: grid graph reduction
2:  if C_vertex is not given then
3:      M = PBGG 1 (M, s, e, maxIter);
4:  else
5:      M = PBGG 2 (M, s, e, C_vertex, maxIter);
6:  end if

    # 1 is free-space, 0 is obstacle or a blocked vertex.
7:  M =(M+1) // 2;

    # Apply SP algorithm
8:  path=ShortestPathfinding(M, s, e, C_vertex, mode);

9:  return path; # list of vertices from s to e
```

E. PBGG (w/ vertex costs)

See Algorithm 5.

F. INTERFACE OF SP ALGORITHM

See Algorithm 6.

G. IMPROVED SHORTEST PATHFINDING UTILIZING PBGG

See Algorithm 7.

REFERENCES

[1] Y. Zhou and J. Zeng, "Massively parallel A* search on a GPU," in *Proc. AAAI Conf. Artif. Intell.*, Feb. 2015, vol. 29, no. 1, pp. 1248–1254.
[2] A. Paszke, "PyTorch: An imperative style, high-performance deep learning library," in *Proc. NeurIPS.*, Dec. 2019, vol. 33, no. 721, pp. 8026–8037.
[3] Z. Hong, P. Sun, X. Tong, H. Pan, R. Zhou, Y. Zhang, Y. Han, J. Wang, S. Yang, and L. Xu, "Improved A-star algorithm for long-distance off-road path planning using terrain data map," *ISPRS Int. J. Geo-Inf.*, vol. 10, no. 11, p. 785, Nov. 2021.
[4] N. R. Sturtevant and S. Rabin, "Canonical orderings on grids," in *Proc. IJCAI*, 2016, pp. 683–689.
[5] D. Harabor and A. Grastien, "The JPS pathfinding system," in *Proc. Int. Symp. Combinat. Search*, 2012, pp. 207–208.
[6] D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid map," in *Proc. AAAI Conf. Artif. Intell.*, 2011, pp. 1114–1119.
[7] K. Pavel and S. David, "Algorithms for efficient computation of convolution," in *Design and Architectures for Digital Signal Processing*. London, U.K.: InTech, Jan. 2013, doi: 10.5772/51942.
[8] A. I. Panov, K. S. Yakovlev, and R. Suvorov, "Grid path planning with deep reinforcement learning: Preliminary results," *Proc. Comput. Sci.*, vol. 123, pp. 347–353, Jan. 2018.
[9] J. Peng, Y. Huang, and G. Luo, "Robot path planning based on improved A* algorithm," *Cybern. Inf. Technol.*, vol. 15, no. 2, pp. 171–180, 2015.
[10] M. Reischl, C. Knauer, and M. Guthe, "Parallel near-optimal pathfinding based on landmarks," *Comput. Graph.*, vol. 102, pp. 1–8, Feb. 2022.
[11] M. Husár, "Reduction-based solving of multi-agent pathfinding on large maps using graph pruning," in *Proc. 21st Int. Conf. Auton. Agents Multiagent Syst.*, 2022, pp. 624–632.
[12] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," *Int. J. Comput. Sci. Netw. Sec.*, vol. 11, no. 1, 2011, pp. 125–130.
[13] G. Tang, C. Tang, C. Claramunt, X. Hu, and P. Zhou, "Geometric A-star algorithm: An improved A-star algorithm for AGV path planning in a port environment," *IEEE Access*, vol. 9, pp. 59196–59210, 2021.
[14] A. Skrynnik, A. Andreychuk, K. Yakovlev, and A. Panov, "Pathfinding in stochastic environments: Learning vs planning," *PeerJ Comput. Sci.*, vol. 8, p. e1056, Aug. 2022.
[15] A. Rafiq, T. A. A. Kadir, and S. N. Ihsan, "Pathfinding algorithms in game development," *IOP Conf. Mater. Sci. Eng.*, vol. 769, Sep. 2020, Art. no. 012021.
[16] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.
[17] S. Roy and Z. Zhang, "Route planning for automatic indoor driving of smart cars," in *Proc. IEEE 7th Int. Conf. Ind. Eng. Appl. (ICIEA)*, Apr. 2020, pp. 743–750.
[18] Y. Sazaki, A. Primanita, and M. Syahroyni, "Pathfinding car racing game using dynamic pathfinding algorithm and algorithm A," in *Proc. 3rd Int. Conf. Wireless Telematics (ICWT)*, Jul. 2017, pp. 164–169.
[19] J. Yao, C. Lin, X. Xie, A. J. Wang, and C.-C. Hung, "Path planning for virtual human motion using improved A* star algorithm," in *Proc. 7th Int. Conf. Inf. Technol., New Generat.*, 2010, pp. 1154–1158.
[20] Z. He, C. Liu, X. Chu, R. R. Negenborn, and Q. Wu, "Dynamic anti-collision A-star algorithm for multi-ship encounter situations," *Appl. Ocean Res.*, vol. 118, Jan. 2022, Art. no. 102995.
[21] H. Wang, S. Lou, J. Jing, Y. Wang, W. Liu, and T. Liu, "The EBS–A* algorithm: An improved A* algorithm for path planning," *PLoS ONE*, vol. 17, no. 2, Feb. 2022, Art. no. e0263841.
[22] S. Erke, D. Bin, N. Yiming, Z. Qi, X. Liang, and Z. Dawei, "An improved A-star based path planning algorithm for autonomous land vehicles," *Int. J. Adv. Robotic Syst.*, vol. 17, no. 5, Sep. 2020, Art. no. 172988142096226.
[23] K. R. Apt and T. Hoare, Eds., "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, vol. 45, 1st ed. New York, NY, USA: Association for Computing Machinery, 2022.
[24] Y. Li, H. Zhang, H. Zhu, J. Li, W. Yan, and Y. Wu, "IBAS: Index based A-star," *IEEE Access*, vol. 6, pp. 11707–11715, 2018, doi: 10.1109/ACCESS.2018.2808407.
[25] D. Burkett, D. Hall, and D. Klein, "Optimal graph search with iterated graph cuts," in *Proc. AAAI Conf. Artif. Intell.*, Aug. 2011, vol. 25, no. 1, pp. 12–17.

CHAN-YOUNG KIM received the B.S. degree in computer information science from Korea University, South Korea, in 2018, where he is currently pursuing the Ph.D. degree with the Department of Electronic Engineering. His research interests include machine learning, deep learning, and computer vision.

SANGHOON SULL (Senior Member, IEEE) received the B.S. degree (Hons.) in electronics engineering from Seoul National University, South Korea, in 1981, the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology, in 1983, and the Ph.D. degree in electrical and computer engineering from the University of Illinois Urbana–Champaign, in 1993. From 1983 to 1986, he was with Korea Broadcasting Systems, working on developing the teletext systems. From 1994 to 1996, he researched motion analysis with the NASA Ames Research Center. From 1996 to 1997, he researched video indexing/browsing and was involved in developing the IBM DB2 video extender with the IBM Almaden Research Center. He joined the School of Electrical Engineering, Korea University, as an Assistant Professor, in 1997. He is currently a professor. His current research interests include artificial intelligence, computer vision, and machine learning. He was an Associate Editor of IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, from 2006 to 2013.