symmetry

MDPI

Article

Path Planning Algorithm Based on Obstacle Clustering Analysis and Graph Search

Lei Wang [1,*] and Lifan Sun [2,3]

[1] School of International Education, Henan University of Science and Technology, Luoyang 471023, China
[2] School of Information Engineering, Henan University of Science and Technology, Luoyang 471023, China; lifan.sun@haust.edu.cn
[3] Longmen Laboratory, Luoyang 471000, China
* Correspondence: 15038647836@126.com or 9905068@haust.edu.cn

Abstract: Path planning is receiving considerable interest in mobile robot research; however, a large number of redundant nodes are typically encountered in the path search process for large-scale maps, resulting in decreased algorithmic efficiency. To address this problem, this paper proposes a graph search path planning algorithm that is based on map preprocessing for creating a weighted graph in the map, thus obtaining a structured search framework. In addition, the reductions in the DBSCAN algorithm were analyzed. Subsequently, the optimal combination of the minPts and Eps required to achieve an efficient and accurate clustering of obstacle communities was determined. The effective edge points were then found by performing obstacle collision detection between special grid nodes. A straight-line connection or A* planning strategy was used between the effective edge points to establish a weighted, undirected graph that contained the start and end points, thereby achieving a structured search framework. This approach reduces the impact of map scale on the time cost of the algorithm and improves the efficiency of path planning. The results of the simulation experiments indicate that the number of nodes to be calculated in the search process of the weighted graph decreases significantly when using the proposed algorithm, thus improving the path planning efficiency. The proposed algorithm offers excellent performance for large-scale maps with few obstacles.

Keywords: DBSCAN algorithm; effective edge points; collision detection; A* algorithm; weighted graph

Citation: Wang, L.; Sun, L. Path Planning Algorithm Based on Obstacle Clustering Analysis and Graph Search. *Symmetry* **2023**, *15*, 1498. https://doi.org/10.3390/sym15081498

Academic Editors: Alice Miller and Jian-Qiang Wang

Received: 20 June 2023
Revised: 18 July 2023
Accepted: 25 July 2023
Published: 28 July 2023

1. Introduction

Mobile robot path planning involves finding a path that meets specified requirements for a mobile robot that moves autonomously in an unknown or known environment through certain algorithms and design strategies [1]. It is being actively studied in fields such as computer science, robotics, control engineering, and artificial intelligence [2]. Currently, robots are used in various applications; in addition, path planning—as one of the necessary technologies for robots—should be application-oriented so as to improve robot performance, solve practical problems, and increase productivity.

Global path planning is a type of static planning that determines an optimal path for a known global map [3]. Conventional global path planning algorithms include the A-Star (A*) [4–6] algorithm, D-Star (D*) [7] algorithm, the rapidly exploring random tree (RRT) [8] algorithm, the genetic algorithm (GA) [9], and the ant colony algorithm optimization (ACO) [10] method. Among them, A* is widely used; it employs a symmetric search process that progresses from the current node to surrounding nodes. The A* algorithm is a heuristic search algorithm that combines Dijkstra's algorithm [11–13] and the breadth-first search (BFS) algorithm [14]. The A* algorithm uses heuristic functions to guide the path-finding process and selects the best node to be extended by calculating the cost value of each node in the grid map until the location of the final target point is found. Although this algorithm offers strong environmental adaptability, as the map scale increases, the number of nodes to be searched increases dramatically, increasing the time required for path planning.

Previous studies have suggested improvements to reduce the time cost of the A* algorithm for large-scale maps. The bidirectional A* algorithm [15,16] uses an alternating forward and backward mechanism to search from the start and end points in opposite directions. It finally converges at the midpoint of the line connecting the start and end points, thus finding the shortest path and improving the search efficiency. The weighted A* algorithm [17] selects the optimal weight value by determining the relationship between the weights and the heuristic terms. It uses a dynamic weighting method [18–20] based on the distance between the starting point and the target point associated with the weight set in conjunction with a method that is based on the path cost associated with the weight set. An improved A* algorithm with updated weights was proposed in [21]; this algorithm considered the degree of channel congestion in a restaurant by using a gray-level model. The results showed that the service robot could effectively avoid crowded channels in the restaurant and complete various tasks in a restaurant environment. A novel A* algorithm that considers evaluation standards, guidelines, and key points was proposed in [22]. This improved A* algorithm used a new evaluation standard to measure the algorithm performance for several sets of parameters and selected the appropriate parameters for optimal performance. Although these methods improve path-finding efficiency, they fail to yield stable results and do not provide a balance between path cost and search time. Theta* [23] is an arbitrary angle path-finding algorithm that can find shorter paths than A*. It extends the A* algorithm by adding visibility detection between nodes, and for each extension of a node, the node visibility of the related nodes is calculated first, which increases the computational cost and time compared with A*. The relaxed A* algorithm [24] uses the cost of the first path that reaches node n to represent the cost of all paths from the starting point to n, i.e., the distance from the starting point to n is computed only once, and the multiple steps associated with comparing and updating the distance from the starting point to n—as in the original A* algorithm —are omitted. This shortens the search time but does not guarantee an optimal solution. Harabor et al. proposed the jump point search (JPS) algorithm [25,26], which improves the undifferentiated expansion rule of A* by expanding only representative grid points, thus significantly reducing the number of node visits and improving the efficiency of the algorithm. These algorithms have made improvements to the A* algorithm at the rule level, but they have failed to consider the overall distributional features of obstacles in the map. Specifically, these algorithms are still in the blind search stage and cannot avoid real-time problems in a large search space [27]. To mitigate the blindness of the search process, this paper proposes a map preprocessing method for analyzing the obstacle distribution features, as well as a new method for obtaining a weighted graph with obstacle distribution features. In addition, the Floyd–Warshall algorithm [28,29] is used to obtain the shortest path in the resulting weighted graph. This algorithm reduces the influence of the map scale on the path planning efficiency.

A clustering algorithm was used to preprocess the obstacles in the maps. Cluster analysis [30] is widely used in mathematics, statistics, networks, medicine, and commerce. It enables the reorganization of data without knowledge of the exact distribution structure, domain information, and the classification of data based on similarity. Numerous clustering algorithms have been proposed, including spectral clustering [31,32], mean-shift [33], K-means [34], the density-based spatial clustering of applications with noise (DBSCAN) [35–37], DHeat [38], and DCore [39]. According to various standards, these approaches are classified into different categories, such as partitioning clustering, hierarchical clustering, centroid-based clustering, and density-based clustering (which is one of the most popular paradigms). DBSCAN is an important density-based clustering algorithm that is used in the research on machine learning and data mining [40]; therefore, it has received considerable attention. This algorithm can identify data point distributions with similar densities and can divide high-density and low-density areas into different groups. The DBSCAN algorithm offers higher robustness and adaptability than conventional clustering algorithms such as K-means, and can handle datasets of different shapes, sizes, and densities without specifying the number of clusters. It can automatically discover clusters

of arbitrary shapes and can distinguish noise points; however, by using different combinations of parameters, it can significantly impact the runtime and clustering results of the algorithm. In practice, Eps and minPts are difficult to determine; the value of Eps affects the size and shape of clusters, whereas the value of minPts affects the number of noise points and the closeness of the clusters. Several studies have developed optimization and improvement methods for the parameter selection of the DBSCAN algorithm for non-uniform density datasets. In [41], minPts was set to 4 according to experience, and appropriate values were selected for Eps by considering multiple values. This method relies heavily on human observation, which makes it ineffective for the clustering of non-uniform density datasets. A dynamic density peak clustering algorithm based on the K-nearest neighbor (DDPC) method was proposed in [42]; this approach can reduce the parameter sensitivity and choose cluster centers automatically. In [43], the best Eps parameter neighborhood values were selected using the global search capability of the bird swarm method. This method can avoid manual intervention and realize adaptive parameter optimization in the clustering process; however, the algorithm also needs to manually adjust the size of minPts. In [44], a new algorithm, SA-DBSCANWBM, was proposed. This method selects a comprehensive index of clustered intra-cluster density and inter-cluster density to evaluate the optimal parameters. Although this method automatically and reasonably selects better parameters and has a good clustering effect, its time complexity is still high. Methods that adapt data with different density distributions by dynamically adjusting the neighborhood radius were proposed in [45–48]. However, these methods do not apply to grid maps with high-clustering-accuracy requirements. This paper proposes a method to determine the optimal parameter combination for DBSCAN based on the characteristics of a grid map. The main contributions of this study are summarized as follows.

(a) The DBSCAN algorithm is applied to the obstacle analysis of grid maps, and the eight-neighborhood attribution features of the obstacle nodes of grid maps are used to obtain the best combination of parameters for the DBSCAN algorithm, as well as to achieve the efficient and accurate clustering of obstacle nodes.
(b) A method is developed to construct a weighted graph in a grid map by calculating the effective edge points of obstacles, and for locating the structural framework of the weighted graph. The collision detection method and the A* algorithm are then used to refine each edge of the weighted graph. Finally, the shortest path is determined using the Floyd–Warshall algorithm, which can effectively reduce the search space for large-scale maps, as well as the correlation between the map scale and path planning process, improving the efficiency of the algorithm.

The remainder of this paper is organized as follows: Section 2 describes the steps of the DBSCAN algorithm, as well as the basis and exploration process of parameter combinations. A new concept is proposed: the effective edge point of the obstacle. The calculation process of the effective edge point of the obstacle is presented, and the process of constructing a weighted graph and specific applications of the A* algorithm therein are comprehensively described. Section 3 presents a two-part experiment designed for verifying the optimal combination of parameters of the DBSCAN algorithm and the effectiveness of the path planning algorithm. Further, the performance features, advantages, and disadvantages of the three path planning algorithms are discussed. Section 4 presents the conclusions and discusses the scope for future work.

2. Methods

2.1. Obstacle Clustering Analysis

The DBSCAN algorithm classifies data points into three types—core points, boundary points, and noise points—and clusters them based on the radius of the neighborhood and the minimum number of data points in the neighborhood. Eps and minPts are important input parameters of DBSCAN. They are defined as follows.

Definition 1. *(Eps neighborhood) For a given object point p, the region within the radius Eps is called the Eps neighborhood of the object.*

Definition 2. *(minPts) The least number of points contained in the Eps neighborhood.*

Definition 3. *(Core point) If the number of samples in the Eps neighborhood is greater than or equal to minPts, the object is called a core point.*

Definition 4. *(Boundary point) If the number of points in the Eps neighborhood of a point is less than minPts, but the point is in the Eps neighborhood of other core points, it is called a boundary point.*

Definition 5. *(Noise point) A point that is neither a core point nor a boundary point.*

The pseudocode of the DBSCAN algorithm is as follows (Algorithm 1).

Algorithm 1: DBSCAN

```
Input: dataset, Eps, minPts
function DBSCAN (dataset, Eps, minPts)
    clusters = 0
    foreach point in dataset
        if point is unvisited then
            mark point as visited
            neighbors = regionQuery(dataset, point, Eps)
            if size of neighbors < minPts then
                mark point as noise
            else
                clusterId = clusterId + 1
                add point to cluster with clusterId
                foreach neighbor in neighbors
                    if neighbor is unvisited then
                        mark neighbor as visited
                        newNeighbors = regionQuery(dataset, neighbor, Eps)
                        if size of newNeighbors >= minPts then
                            add newNeighbors to neighbors
                        end if
                        if neighbor is not a member of any cluster then
                            add neighbor to cluster with clusterId
                        end if
                    end if
                end foreach
            end if
        end if
    end foreach
end function
function regionQuery(dataset, point, Eps)
    neighbors = empty list
    foreach p in dataset
        if distance between point and p <= eps then
            add p to neighbors
        end if
    end foreach
    return neighbors
end function
```

The DBSCAN algorithm must traverse all the points to be clustered and obtain the clustering basis by calculating the number of points within the neighborhood radius of each point. The time complexity of this algorithm is $O(m \times n)$, where m is the number of points to be clustered, and n is the time required to find all the points within the neighborhood

radius. For a fixed m, the runtime of the DBSCAN algorithm can be effectively reduced by minimizing n. Therefore, reducing the neighborhood search time for each point by minimizing the neighborhood radius range can improve the algorithmic efficiency.

First, a map containing obstacles is divided into several square grids of the same size using the method of rasterizing the map. The square grids containing obstacles are called obstacle grid nodes, and the grids without obstacles are called free grid nodes. In the grid map, if an obstacle grid node exists in the eight neighborhood of the obstacle grid node, no robot passage path exists between the two obstacle grid nodes, which together form the entire obstacle and therefore belong to the same group. If no obstacle exists in the eight neighborhood of the obstacle grid node, a robot passage path may exist between the center point and other obstacle grid nodes that cannot belong to the same obstacle group. Thus, the accurate clustering of obstacle grid nodes can be achieved by searching only the eight neighborhoods in the grid map. In Figure 1, the black grid is the obstacle, the white grid is the passable grid, the grid node at the center of the circle is the center point, and the full circle is only required to cover eight neighborhoods. Assuming that the edge length of each square grid node is 1, the distances of the center point from the eight-neighborhood nodes are 1 and $\sqrt{2}$. This implies that when the neighborhood radius $Eps = 1.5$, then the eight neighborhoods of the nodes can be covered. As shown in Figure 1a, within a radius of 1.5 from the center point, all the obstacle grid nodes that exist can be clustered into the same family to form the whole obstacle. Further, Figure 1b shows that there exist passable paths between the center node and the obstacle grid nodes outside a radius of 1.5; therefore, these two nodes cannot be clustered into the same family. Based on this feature of the grid map, the following DBSCAN parameters are specified: $Eps = 1.5$ and $MinPts = 1$, which can ensure the shortest algorithm runtime while achieving an accurate clustering of the map obstacles.

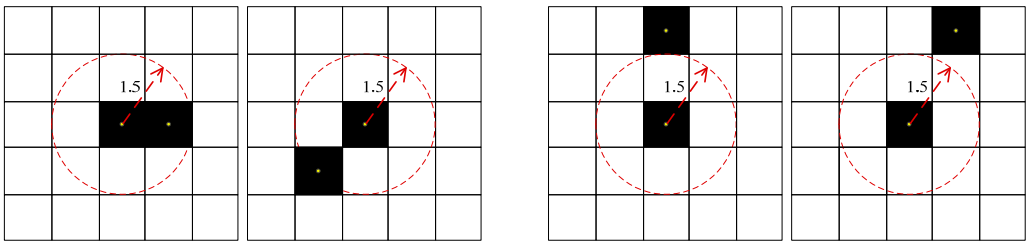

Figure 1. Clustering distribution of the obstacle grid nodes: (**a**) within a neighborhood radius of 1.5, where the obstacle grid nodes and central nodes are clustered into the same group; and (**b**) outside a neighborhood radius of 1.5, where the obstacle grid nodes and central nodes do not belong to the same group.

2.2. Graph Search

2.2.1. Collision Detection Algorithm

During the path planning process, whether the robot walks in a straight line or turns depends on whether obstacles exist on the straight line connecting the starting point and the target point. The obstacle collision detection algorithm determines whether this line collides with an obstacle. The Bresenham algorithm [49,50] is commonly used for collision detection and employs the incremental relationship of x and y between two consecutive grid nodes to calculate the coordinates of the next grid node. It detects whether an obstacle exists in the grid node along a straight line to determine whether this line intersects with an obstacle. Thus, the algorithm does not require the use of trigonometric functions and other complex calculations, making it simple, efficient, and easy to implement. However, the Bresenham algorithm has accuracy problems: it generates gaps and overlaps in some cases. To improve the accuracy of obstacle collision detection, this paper proposes a collision detection method that uses linear equations to control the step size, thereby improving

the accuracy of collision between a straight line and an obstacle. The basic principle of the algorithm is as follows.

To detect an obstacle along a straight line, first, the coordinates of the start and end points of this line should be used to establish the straight-line equation. Then, the next point (x, y) on the straight line is calculated with a fixed step. When the point (x, y) is inside the obstacle grid, a collision occurs; otherwise, the next point (x, y) is calculated. This process continues until all the points (x, y) in the grid are exhausted. Here, the start point is Ns, the end point is Ng, and the movement step is t. The algorithm moves from node Ns to node Ng along a straight line $Ns - Ng$, and it is constantly detecting whether the current point is located in the obstacle grid. If it is located inside the obstacle grid, a collision occurs; if no obstacle is detected until the end point Ng, nodes Ns and Ng are considered unobstructed or directly reachable.

As shown in Figure 2, collision detection is performed using a two-point linear equation from the start point Ns to the end point Ng, with a step size t of 0.1. The coordinates of the point to be detected (x, y) can be calculated using Equation (1).

$$\begin{cases} x_i = x_{Ns} + t \times i, i = 0, 1, 2 \cdots n \\ y_i = \frac{y_{Ns} - y_{Ng}}{x_{Ns} - x_{Ng}} \times x_i + y_{Ns} - x_{Ns} \times \frac{y_{Ns} - y_{Ng}}{x_{Ns} - x_{Ng}} \end{cases} \tag{1}$$

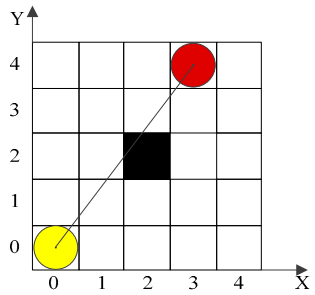

Figure 2. Detecting collisions with obstacles. Here, the black grid point is the obstacle, the yellow grid point is the starting point Ns, and the red grid point is the end point Ng.

When x_i is in the range of 1.5–2, y_i can be calculated to be in the range of 2–2.5. The point (x_i, y_i) on the line $Ns - Ng$ is located in the area within the obstacle grid (2, 2), and a collision between this line and the obstacle grid (2, 2) is detected. Using this method, obstacle collision detection can be performed for different accuracy requirements by adjusting the step size t.

2.2.2. Effective Edge Points of Obstacles

In this paper, we propose a method to find the effective edge points of obstacles, which are then used as the main framework of the weighted graph. The relative positional relationship between the obstacle group, the start point, and the end point determines the range of movement of the robot. Here, the start and end points are connected with a straight line, and this line is used as a reference to find the effective edge points of obstacle groups by calculating the sum of the distances from the obstacle grid nodes to the start and end points.

As shown in Figure 3, $W_m = TL_m + TR_m$, where TL_m and TR_m are the straight-line distances from the obstacle grid node N_m to the start and end points, respectively. The two relative positional relationships between the obstacle group and the reference system are as follows: 1. the obstacle group is distributed on both sides of the reference system, and 2. the obstacle group is located on the same side of the reference system. In Figure 3a, nodes N_1 and N_m (which are defined as the effective edge points of the obstacle group), are located on both sides of the reference system with the largest W value. In Figure 3b, N_1 and N_m are nodes with the largest and smallest W values, and are located on the same side of the reference system. Specifically, when the reference system collides with the obstacle group, the nodes farthest from both sides of the reference system are selected as the effective edge

points, and when the reference system does not collide with the obstacle group, the nodes that are closest and farthest from the reference system are selected as the effective edge points. The pseudocode to find the effective edge points is as follows (Algorithm 2).

Algorithm 2: Find Effective Edge Points

```
Input: All grid points of obstacle: G
Output: Effective edge points set: L
function FindEffectiveEdgePoints (G)
define: W is the sum of the distances between node Ni and the start and end points.
    List L = null; Node K1 = null; Node K2 = null;
    foreach Ni in G do
        num = 0;
        if L == null then L.add(Ni); continue;
        end if
        for (j = 0; j < L.length; j++) do
            if Ni.cluster == L[j].cluster then
                num++;
                if K1 == null then K1 = L[j];
                else if K2 == null then K2 = L[j];
                end if
            end if
        end for
        if num == 0 || num == 1 then
            L.add(Ni);
        else if num == 2 then
            L.RemoveAll(x => x == Ni.cluster);
            if there is no collision between G and straight-line start-end then
                L.add(Max{W(Ni),W(K1),W(K2)});
                L.add(Min{W(Ni),W(K1),W(K2)});
            else
                if Ni is on the same side of the line start-end as K1 and K2 then
                    L.add(Max{W(Ni),W(K1),W(K2)});
                    L.add(SecondMax{W(Ni),W(K1),W(K2)});
                else if Ni is on a different side of the line start-end compared with K1, and K2 then
                    L.add(Ni);
                    L.add(Max{W(K1),W(K2)});
                else if K1 is on a different side of the line start-end compared with K2 then
                    if Ni and K1 are on the same side of the line start-end then
                        L.add(K2);
                        L.add(Max{W(K1),W(Ni)});
                    else if Ni and K2 are on the same side of the line start-end then
                        L.add(K1);
                        L.add(Max{W(K2),W(Ni)});
                    end if
                end if
            end if
        end if
    end foreach
end function
```

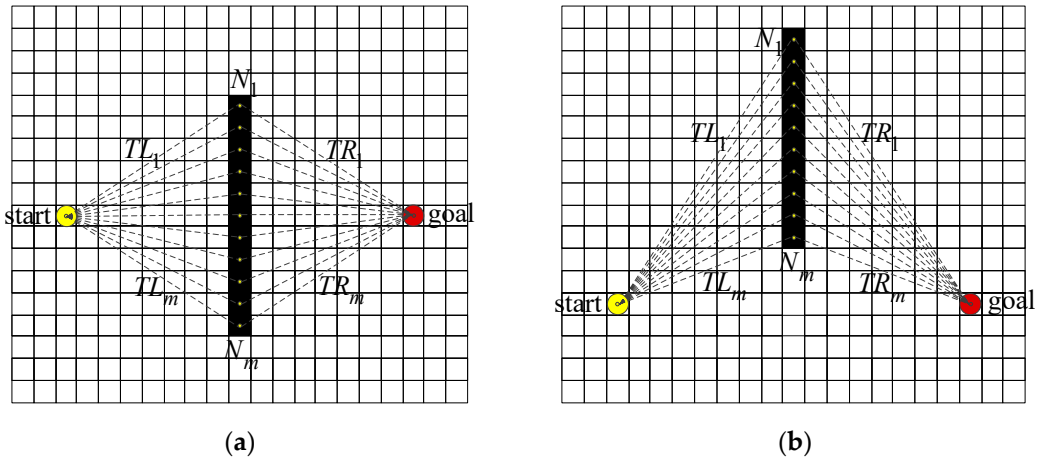

Figure 3. Representative example of the distribution of the obstacle group relative to the start–goal reference system. (**a**) The start–goal reference system collides with the obstacle group. (**b**) The start–goal reference system does not collide with the obstacle group.

2.2.3. Construction of the Weighted Graph

The effective edge points of the obstacle group are searched to construct a weighted graph frame and to achieve a basic localization of the graph structure. A suitable free node within the eight neighborhoods of each effective edge point is considered a node in the weighted graph. The principle of node extension is as follows:

Extended nodes are free grid nodes within the eight neighborhoods of effective edge points.

Extended nodes are the diagonal grid nodes of the effective edge points.

A maximum number of free nodes exists in the eight neighborhoods of the extended nodes.

As shown in Figure 4, the extended nodes of effective edge points are N_1, N_2, N_3, and N_4, and the shortest paths between the extended nodes are planned sequentially from the start point to obtain the complete weighted graph. First, collision detection is performed from the start node to the goal node, and the obstacle group B_1 is detected. The extended nodes of the effective edge points of B_1 are calculated as N_1 and N_2, and the shortest paths from the start node to nodes N_1 and N_2 are planned. Then, collision detection is performed from the starting point of node N_1 to the goal node, and obstacle group B_2 is detected. The extended nodes of the effective edge points of B_2 are calculated as N_3 and N_4, and the shortest paths from node N_1 to nodes N_3 and N_4 are planned. Finally, collision detection is performed for the goal node with nodes N_2, N_3, and N_4 as the starting point. As no collision occurs with obstacles other than those of its group, the shortest paths from nodes N_2, N_3, and N_4 to the goal node are planned. A weighted graph covering the edge regions of the start, goal, and obstacle groups is then constructed.

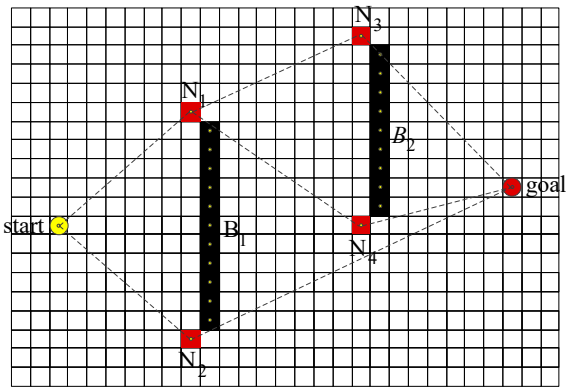

Figure 4. Example of a weighted graph construction.

Two possible situations for path planning between nodes exist: 1. The nodes are directly reachable, i.e., no obstacles exist between the two nodes, and a straight line can connect the nodes—for example, $start - N_1$, $start - N_2$, $N_1 - N_3$, and $N_4 - goal$. 2. The nodes are not directly reachable, i.e., obstacles exist between the two nodes, and a straight line cannot connect the nodes—for example, $N_1 - N_4$, $N_2 - goal$, $N_1 - goal$, and $N_3 - goal$. In the second situation, the A* algorithm is then used to search for the shortest path between the two nodes. The pseudocode is as follows (Algorithm 3):

Algorithm 3: Construct Weighted Graph

```
Input: the starting and end points: NS, NE
Output: matrix M
function ConstructWeightedGraph (NS, NE)
    List LS = null; List LB = null;
    LS.add(NS);
    foreach (Ni in LS) do
        LB = null;
        detect the first obstacle node along straight-line Ni to NE
        if straight-line Ni-NE is directly reachable then
            the Euclidean distance between Ni and NE is stored in M
        else
            Add the effective edge points of the first collided group to LS
            Add the effective edge points of the first collided group to LB
            LB. Add(NE);
            foreach Nj in LB
              detect the first obstacle node along straight-line Ni to Nj
              if straight-line Ni-Nj is directly reachable then
                    the Euclidean distance between Ni and Nj is stored in M
              else
                    if Nj == NE then continue;
                    else
                        AStarPathplanning (Ni, Nj);
                    end if
              end if
            end foreach
        end if
    end foreach
    return M
end function
```

2.2.4. The Search for the Shortest Path

The Floyd–Warshall algorithm is a simple algorithm based on the greedy algorithm and dynamic programming that finds the shortest path between points in a graph. It calculates distances between every pair of vertices, and if the distance between any two vertices can be shortened, the distance is updated. When the distances between all vertices are updated, the shortest path is obtained. The result of the algorithm is stored in two matrices, for distances and paths—the former is used to store the shortest distance from each point to all other points, and the latter is used to store the path of the shortest distance from each point to all other points.

As shown in Figure 5, the weighted graph is searched, and the shortest path sequence obtained from the start point to the end point is $start - N_1 - N_4 - goal$. Here, the path between nodes N_1 and N_4 is non-direct and cannot be connected with a straight line. Therefore, the A* algorithm is used to plan the shortest path from node N_1 to node N_4. The path-node sequence obtained via planning using the A* algorithm planning is $N_1 - N_{11} - N_{12} - N_{13} - N_{14} - N_{15} - N_{16} - N_{17} - N_{18} - N_4$, as shown in Figure 5. Finally, all the path nodes are connected to obtain the shortest path-node sequence from the start point to the end point: $start - N_1 - N_{11} - N_{12} - N_{13} - N_{14} - N_{15} - N_{16} - N_{17} - N_{18} - N_4 - goal$.

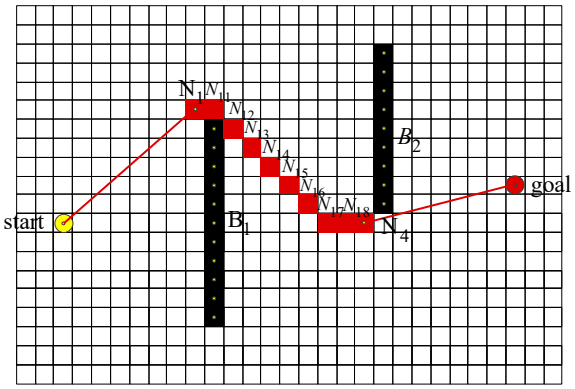

Figure 5. Search results of the weighted graph. Here, the black grid nodes indicate obstacles, and the red straight lines and red grid nodes indicate the shortest path.

3. Experiments and Discussion

3.1. Design of Experiments

The experiment was divided into two parts. In the first part of the experiment, the grid coordinates of obstacles in a map were stored in a txt file; each txt file contained a map. Then, the M function file for DBSCAN was developed in the MATLAB (R2013b, MathWorks, Natick, MA, USA) environment. The M file was created using the following steps: 1. the load() function is used to read the map information; 2. all obstacle grid nodes are stored in a dataset, and a kd-tree is established for the purpose of searching the nearest neighbor nodes in this dataset; 3. the DBSCAN algorithm is used to classify the dataset into families; and 4. randomly selected colors are assigned to families and the clustering results are visualized using the plot() function. Four maps with a 100 $\times$ 100 grid and different obstacle distribution features were used in the experiment. The effects of different parameter combinations on the runtime and clustering results of the DBSCAN algorithm were analyzed. The number of nodes to be clustered in these maps was 172, 277, 625, and 3411. According to the features of the neighborhood combinations between obstacles in the grid map, this experiment set the minimum number of data points in a neighborhood minPts to 1. Based on this, the search radius was continuously adjusted to observe its effect on the clustering results and runtime of the algorithm. The optimal combination of parameters was determined by analyzing the mechanism of the DBSCAN algorithm.

In the second part of the experiment, the graph search path planning algorithm with DBSCAN was developed using C# on the Visual Studio Ultimate 2012 platform. The code was developed using the Windows Forms Application template of Microsoft Visual C# in the .NET framework (4.5.2). First, the main interface of the program was developed to display the map overview, graph, and the results of the path planning. The main interface contained buttons for A*, JPS, and the proposed algorithm. The GridBox class, DBSCAN algorithm, collision detection algorithm, Floyd–Warshall algorithm, A* algorithm, JPS algorithm, and the proposed algorithm were then added to the program. Among them, GridBox could display obstacle grid nodes, free grid nodes, start and end points, as well as nodes accessed during the algorithm search. The proposed algorithm included the following steps: 1. read the txt file and load the map; 2. call the DBSCAN algorithm for the clustering analysis of obstacles; 3. call the collision detection algorithm to calculate the effective edge points of each obstacle group and perform local expansion; and 4. construct a weighted graph and call the Floyd–Warshall algorithm to search for the shortest path. As in the first part of the experiment, four maps were used. The proposed algorithm was compared with the A* and JPS algorithms in terms of features, work efficiency, and applicability, which were based on experimental data.

The hardware platform used in the experiment included a computer with a six-core i5 CPU, 8GB RAM, and 1TB SSD.

3.2. Results and Discussion

In DBSCAN, the kd-tree [51,52] algorithm is used to find the nearest neighbor nodes within the neighborhood Eps; therefore, the runtime of DBSCAN is closely related to that of the kd-tree algorithm. In addition, to achieve a precise clustering of the grid maps, minPts was set to 1. Finally, different values are assigned to Eps and the runtimes of DBSCAN are calculated, as listed in Table 1. Evidently, the overall runtime of the DBSCAN algorithm increases rapidly with an increase in Eps; it exhibits a trend of gradual growth in the early stage and rapid growth in the later stage. To analyze the variation trend, the data of the four maps are fit considering Eps as the independent variable and the runtime of DBSCAN as the dependent variable, as shown in Figure 6. The data conform to the power function characteristics, i.e., $y = ax^b + c$. Here, x is Eps; y is the runtime of DBSCAN; and a, b, and c are the real-valued constants. For a given map, when Eps changes, the runtime of DBSCAN changes with the trend of a fixed power function. Overall, the runtime of DBSCAN varies linearly with an increase in the number of cluster nodes. However, the number of clustering nodes in map(a) was lower than that in map(b), but the clustering time for map(a) was significantly longer than that for map(b). This phenomenon is related to the operating mechanism of the kd-tree algorithm. The distribution of obstacle grid nodes in map(a) was concentrated, whereas that in map(b) was relatively scattered. The differences in the distribution characteristics of the obstacles caused significant differences in the establishment and query process of the binary trees in the kd-tree algorithm. Therefore, in the case of a small difference in the number of obstacle nodes, the runtime of DBSCAN did not follow a linear pattern. Based on this analysis, to effectively reduce the time cost of the algorithm, Eps was set to 1.5, and minPts was set to 1. The results are presented in Figure 7.

The path planning process in the four grid maps using the A*, JPS, and the proposed algorithms was experimentally analyzed, and the results are presented in Figure 8. The shortest paths obtained in the four maps using these algorithms differ, and the path planned with the proposed algorithm had the fewest turns and therefore was the shortest path.

Table 1. Results of the DBSCAN clustering algorithm.

Parameter Setting	Map(a) Runtime (s)	Map(b) Runtime (s)	Map(c) Runtime (s)	Map(d) Runtime (s)
Eps = 1.5	0.031	0.031	0.063	0.484
Eps = 2	0.016	0.031	0.078	0.531
Eps = 3	0.031	0.031	0.062	0.969
Eps = 4	0.031	0.031	0.062	1.953
Eps = 5	0.032	0.047	0.094	4.718
Eps = 6	0.031	0.032	0.109	7.030
Eps = 7	0.046	0.047	0.109	12.312
Eps = 8	0.062	0.047	0.125	26.900
Eps = 9	0.094	0.047	0.156	47.852
Eps = 10	0.110	0.063	0.187	77.868
Eps = 11	0.156	0.063	0.234	171.859
Eps = 12	0.218	0.078	0.281	391.516
Eps = 13	0.282	0.094	0.500	830.276
Eps = 14	0.359	0.094	0.656	970.134
Eps = 15	0.453	0.109	0.812	-
Eps = 16	0.516	0.125	1.031	-
Eps = 17	0.641	0.141	1.234	-
Eps = 18	0.781	0.157	1.664	-
Eps = 19	0.906	0.172	2.015	-
Eps = 20	1.031	0.203	2.703	-

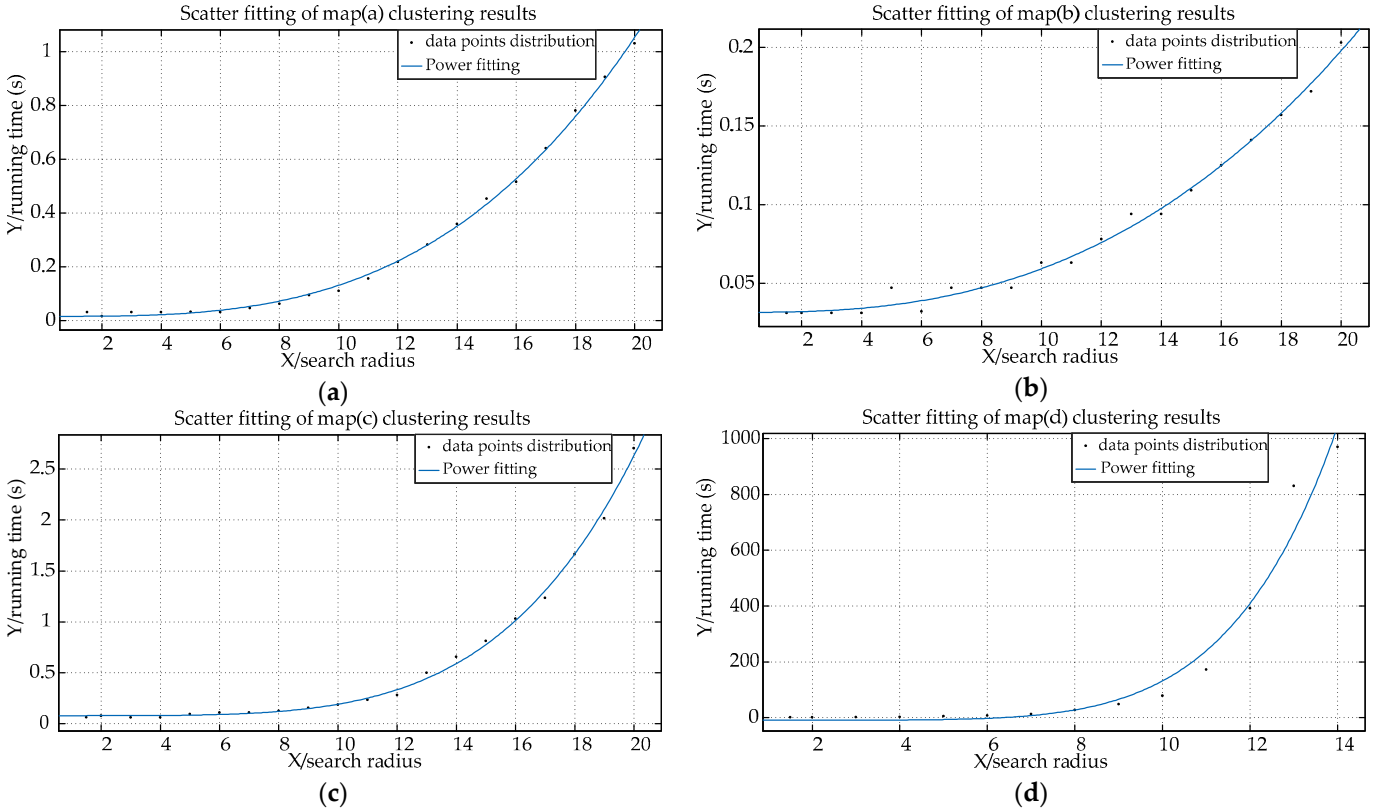

Figure 6. Experimental data fitting for the different maps when using DBSCAN. Curve fitting of the DBSCAN runtime with neighborhood radius Eps for map (**a**) a, (**b**) b, (**c**) c, and (**d**) d.

As is evident from the experimental data in Table 2, map(a) contains a simple arrangement of obstacles, and the A* algorithm calculates a large number of nodes during the search, incurring a significant time cost. The JPS algorithm improves efficiency by searching for jump points, but it still incurs a time cost in the process of map preprocessing. By contrast, the proposed algorithm uses DBSCAN to perform the clustering analysis of the obstacle grid nodes, which is less time consuming and outperforms the other algorithms owing to the presence of fewer obstacles.

Map(b) contains an enclosed area with only one exit. As the heuristic search strategy of the A* algorithm is based on the distance to the end point and not on the distribution features of the obstacles, it searches a large number of nodes in a large-scale map with a low obstacle density. By contrast, the proposed algorithm eliminates the blindness in path planning and optimizes the search strategy by finding effective edge points. The time required for the proposed algorithm was 30.9 ms, which is 31.3% of that of the A* algorithm. However, in the process of constructing a weighted graph, the proposed algorithm searches a large number of redundant nodes, which decreases its efficiency and increases its runtime compared with that of the JPS algorithm (8.88 ms).

Map(c) contains a maze, and, in the partially closed space, the A* algorithm searches the highest number of nodes and requires the longest time. By comparison, the proposed algorithm is highly efficient because it employs effective edge points with a clear orientation; therefore, its runtime is only 22.7% of that of the A* algorithm. However, the presence of a large number of obstacle grid nodes increases the time required for map clustering, resulting in a runtime that is 4.2 times that of the JPS algorithm.

Map(d) contains a cluttered room with a large density and irregular distribution of obstacles. The low number of free nodes results in a smaller searchable space for the A* algorithm as it requires only 3.2 ms, indicating its advantage in this case. By contrast, the proposed algorithm performs a clustering analysis of 3411 obstacle grid nodes, which significantly increases the time required for map preprocessing, resulting in the worst

performance for this map. Its runtime was 34.4 times that of the A* algorithm and 9.6 times that of the JPS algorithm.

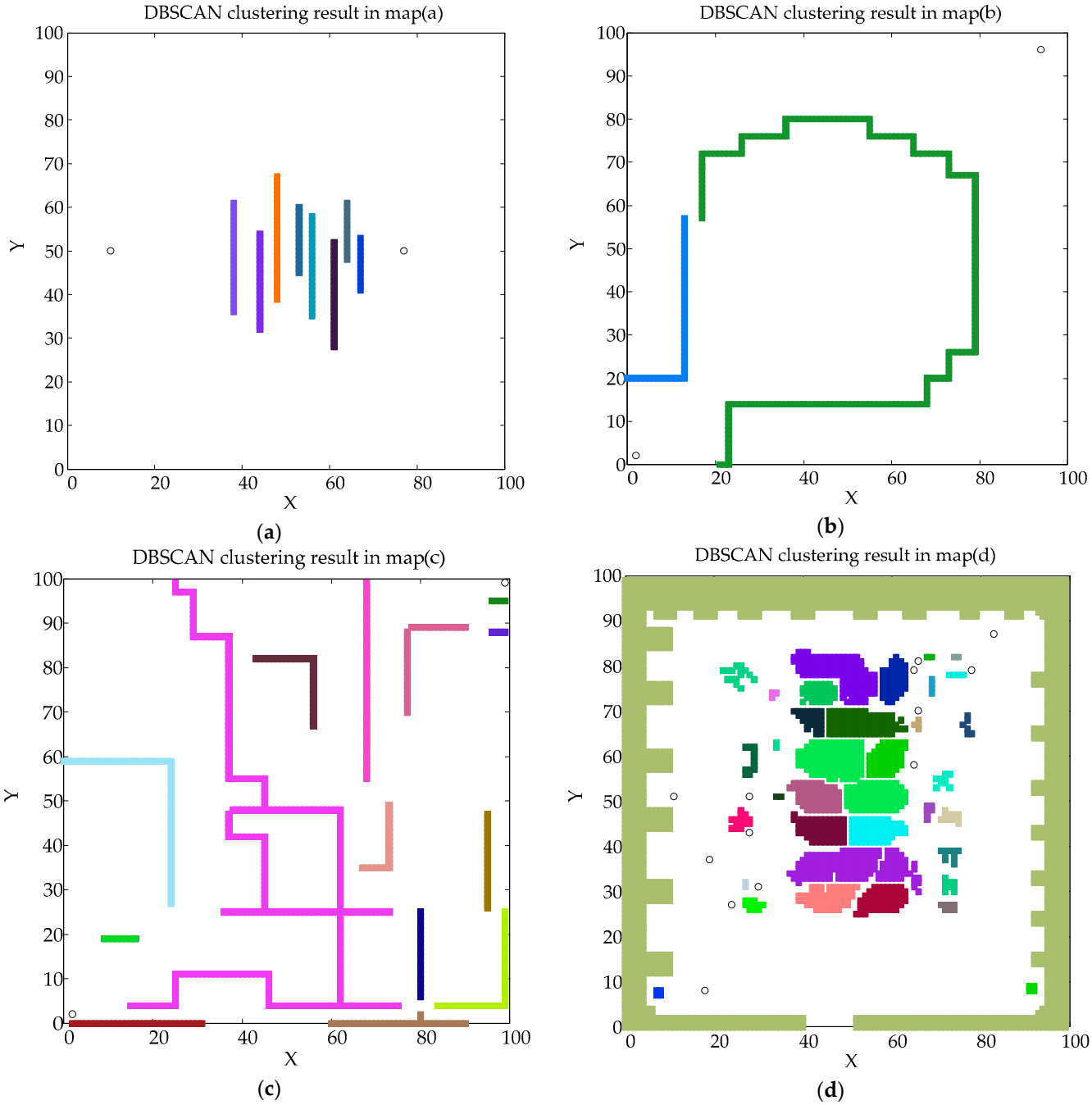

Figure 7. Graphical representation of the clustering results of obstacles in the maps with $Eps = 1.5$ and $MinPts = 1$. (**a**–**d**) In maps a, b, c, and d, the white part represents the free and accessible area, while the colored part represents obstacles. Based on the clustering results of DBSCAN, this paper renders obstacles of the same family into the same color, and clearly identifies all obstacles in the 4 maps by randomly selecting colors. Here, the circular nodes in the lower left and upper right corners indicate the start and end points, respectively, and the remaining circular nodes indicate noise points.

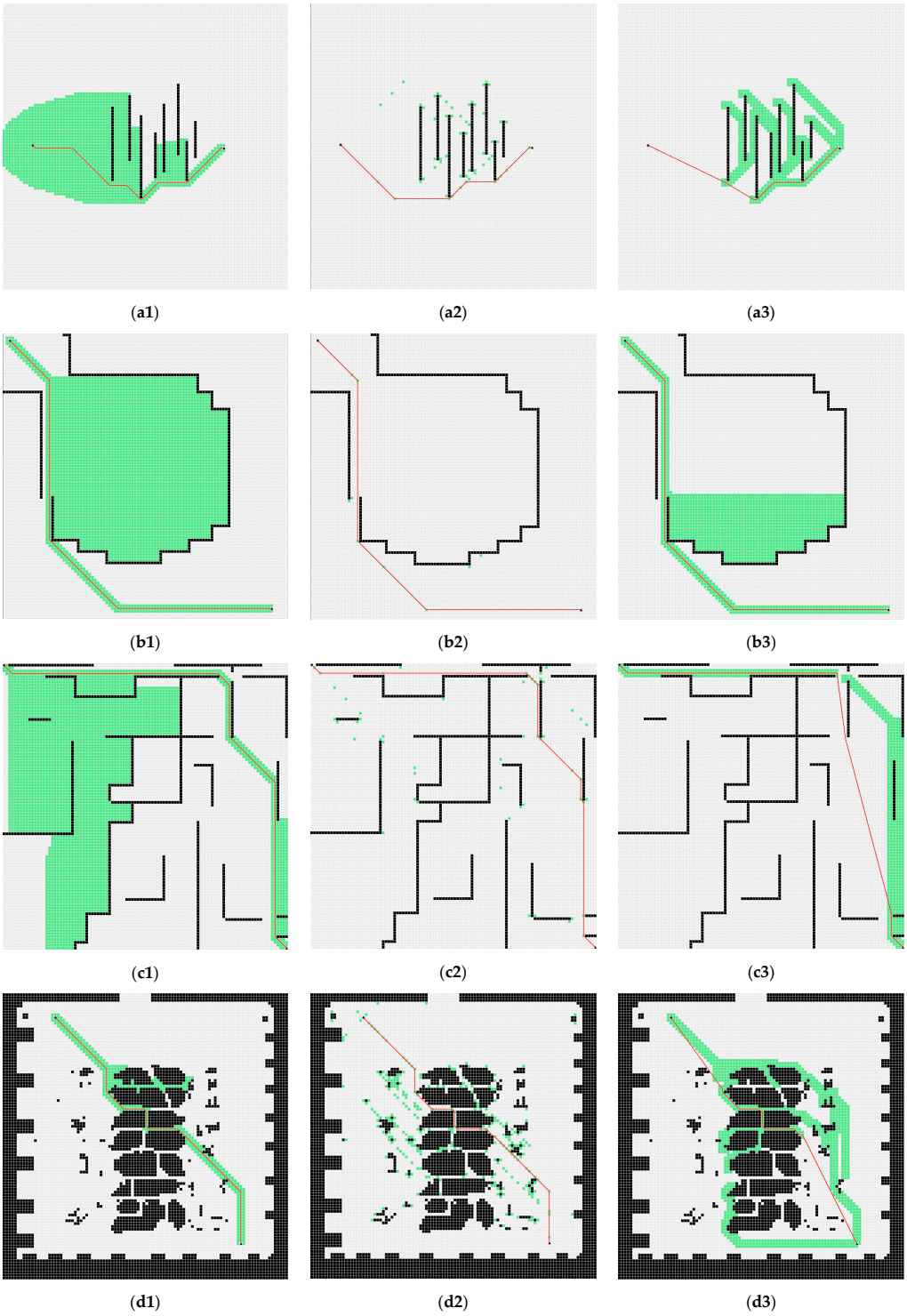

Figure 8. The path planning diagrams of the three algorithms. (**a1**–**a3**) respectively showcases the path planning results of A* algorithm, JPS algorithm, and the proposed algorithm in map(a); (**b1**–**b3**) shows the path planning results of A* algorithm, JPS algorithm, and the proposed algorithm in map(b), respectively; (**c1**–**c3**) shows the path planning results of A* algorithm, JPS algorithm, and the proposed algorithm in map(c), respectively; (**d1**–**d3**) shows the path planning results of A* algorithm, JPS algorithm, and the proposed algorithm in map(d). Here, the circular grid node was the starting point, the triangular grid node was the end point, the black square grid node was the obstacle, and the rest were free grid nodes. The light green grid node indicates the area searched by the algorithm, and the red connecting line indicates the shortest path.

Table 2. Runtimes and path lengths of the three algorithms.

Map	Path Length of A *	Runtime of A * (ms)	Path Length of JPS	Runtime of JPS (ms)	Path Length of the Proposed Algorithm	Runtime of the Proposed Algorithm (ms)
(a)	82.31	28.90	82.31	16.95	79.08	15.90
(b)	163.70	98.73	163.70	8.88	163.70	30.90
(c)	182.18	92.32	182.18	4.98	177.15	20.96
(d)	115.87	3.20	115.84	11.47	111.25	110.00

In summary, the proposed algorithm can effectively perform path planning for grid maps and can calculate the shortest path. Compared with the A* and JPS algorithms, which require more time for searching free nodes, the proposed algorithm primarily analyzes and calculates the obstacle nodes. Therefore, it is more practical and offers better performance for large-scale maps with fewer obstacle nodes.

4. Conclusions, Limitations, and Future Research

This study used the DBSCAN algorithm for the clustering analysis of map obstacles. The neighborhood radius was set to 1.5, and the minimum number of data points in the neighborhood was set to 1 in order to efficiently and accurately cluster the obstacle groups. A weighted graph was then constructed by calculating the effective edge points of each cluster, and the graph was improved using the collision detection method and the A* algorithm. Finally, the Floyd–Warshall algorithm was used to calculate the shortest path in the weighted graph. The optimal combination of parameters of DBSCAN, as well as the effectiveness and applicability of the proposed algorithm were verified through experiments. This paper addressed two problems. The first problem involved the clustering analysis of map obstacles. According to the characteristics of grid maps, the influence of each parameter of the clustering algorithm (DBSCAN) on the clustering results was examined, and the optimal parameter combination was determined to achieve a precise clustering of the obstacles in grid maps. The second problem involved constructing a weighted graph. A new method that uses obstacle collision detection between special nodes was developed to establish a weighted graph framework. This method constructs a weighted graph by using different strategies to perform the path planning between the nodes in the weighted graph. For large-scale maps with fewer obstacle nodes, establishing a search model for the weighted graph can effectively improve the efficiency of path planning.

Although the proposed algorithm provides a new approach for path planning, the results show that it has an important limitation in the case of large-scale data clustering (map(d) in Table 2). The time cost of the algorithm increases geometrically with an increase in clustering data. This is mainly because of the high time cost of DBSCAN for large-scale datasets. Although several methods to improve DBSCAN have been proposed, they cannot meet the requirements of an accurate clustering of grid maps.

Therefore, in future studies, we will carefully examine the positional relationship between obstacle nodes in a grid map to find additional rules and to reduce the complexity of grid map clustering. In addition, further research on large-scale data clustering will be conducted to find a better balance between clustering accuracy and efficiency.

Author Contributions: Conceptualization, L.W.; methodology, L.W. and L.S.; software, L.W.; formal analysis, L.S.; investigation, L.W. and L.S.; data curation, L.W.; writing—original draft preparation, L.W.; writing—review and editing, L.S.; supervision, L.S. All authors have read and agreed to the published version of the manuscript.

Funding: The research was funded by the National Natural Science Foundation of China (No. 62271193); the Aeronautical Science Foundation of China (No. 20185142003); the Natural Science Foundation of Henan Province, China (No. 222300420433); the Science and Technology Innovative Talents in Universities of Henan Province, China (No. 21HASTIT030); and the Young Backbone Teachers in Universities of Henan Province, China (No. 2020GGJS073); Major Science and Technology Projects of Longmen Laboratory (No. 231100220200).

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Antonyshyn, L.; Silveira, J.; Givigi, S.; Marshall, J. Multiple Mobile Robot Task and Motion Planning: A Survey. *ACM Comput. Surv.* **2023**, *55*, 213. [CrossRef]
2. Liu, L.; Wang, X.; Yang, X.; Liu, H.; Li, J.; Wang, P. Path planning techniques for mobile robots: Review and prospect. *Expert Syst. Appl.* **2023**, *227*, 120254. [CrossRef]
3. Nazarahari, M.; Khanmirza, E.; Doostie, S. Multi-objective multi-robot path planning in continuous environment using an enhanced genetic algorithm. *Expert Syst. Appl.* **2019**, *115*, 106–120. [CrossRef]
4. Ni, Y.; Zhuo, Q.; Li, N.; Yu, K.; He, M.; Gao, X. Characteristics and Optimization Strategies of A* Algorithm and Ant Colony Optimization in Global Path Planning Algorithm. *Int. J. Pattern Recognit.* **2023**, *37*, 2351006. [CrossRef]
5. Wang, P.; Liu, Y.; Yao, W.; Yu, Y. Improved A-star algorithm based on multivariate fusion heuristic function for autonomous driving path planning. *Proc. Inst. Mech. Eng. Part D-J. Automob. Eng.* **2022**, *237*, 1527–1542. [CrossRef]
6. Zhang, Y.; Li, L.; Lin, H.; Ma, Z.; Zhao, J. Development of Path Planning Approach Using Improved A-star Algorithm in AGV System. *J. Internet Technol.* **2019**, *20*, 915–924. [CrossRef]
7. Chen, G.; Liu, D.; Wang, Y.; Jia, Q.; Zhang, X. Path planning method with obstacle avoidance for manipulators in dynamic environment. *Int. J. Adv. Robot. Syst.* **2018**, *15*, 1729881418820223. [CrossRef]
8. Iram, N.; Amna, K.; Khurshid, A.; Zulfiqar, H. A Path-Planning Performance Comparison of RRT*-AB with MEA* in a 2-Dimensional Environment. *Symmetry* **2019**, *11*, 945. [CrossRef]
9. Chien-Ming, C.; Shi, L.; Jirsen, N.; Jimmy, W. A Genetic Algorithm for the Waitable Time-Varying Multi-Depot Green Vehicle Routing Problem. *Symmetry* **2023**, *15*, 124. [CrossRef]
10. Chen, T.; Chen, S.; Zhang, K.; Qiu, G.; Li, Q.; Chen, X. A jump point search improved ant colony hybrid optimization algorithm for path planning of mobile robot. *Int. J. Adv. Robot. Syst.* **2022**, *19*, 17298806221127953. [CrossRef]
11. Bai, X.; Wang, L.; Hu, Y.; Li, P.; Zu, Y. Optimal Path Planning Method for IMU System-Level Calibration Based on Improved Dijkstra's Algorithm. *IEEE Access* **2023**, *11*, 11364–11376. [CrossRef]
12. Yilmaz, A.; Ozturk, O. Designing a LoRa Network Using Dijkstra's Algorithm. In Proceedings of the 6th International Conference on Smart City Applications, Safranbolu, Turkey, 27–29 October 2021; pp. 1047–1056. [CrossRef]
13. Liu, L.; Lin, J.; Yao, J.; He, D.; Zheng, J.; Huang, J.; Shi, P. Path Planning for Smart Car Based on Dijkstra Algorithm and Dynamic Window Approach. *Wirel. Commun. Mob. Comput.* **2021**, *2021*, 356–368. [CrossRef]
14. Banerjee, N.; Chakraborty, S.; Raman, V.; Satti, S.R. Space Efficient Linear Time Algorithms for BFS, DFS and Applications. *Theor. Comput. Syst.* **2018**, *62*, 1736–1762. [CrossRef]
15. Lai, W.K.; Shieh, C.S.; Yang, C.P. A D2D Group Communication Scheme Using Bidirectional and InCremental A-Star Search to Configure Paths. *Mathematics* **2022**, *10*, 3321. [CrossRef]
16. Wang, H.; Qi, X.; Lou, S.; Jing, J.; He, H.; Liu, W. An Efficient and Robust Improved A* Algorithm for Path Planning. *Symmetry* **2021**, *13*, 2213. [CrossRef]
17. Zeyad, A.A.; Mohd, S.S.; Afnizanfaizal, A. A new weighted pathfinding algorithms to reduce the search time on grid maps. *Expert Syst. Appl.* **2017**, *71*, 319–331. [CrossRef]
18. Lang, L.; Zhou, S.; Zhong, M.; Sun, G.; Pan, B.; Guo, P. A Big Data Based Dynamic Weight Approach for RFM Segmentation. *CMC-Comput. Mater. Contin.* **2023**, *74*, 3503–3513. [CrossRef]
19. Liu, G.; Zhou, X.; Xu, X.; Wang, L.; Zhang, W. Fault diagnosis of diesel engine information fusion based on adaptive dynamic weighted hybrid distance-taguchi method (ADWHD-T). *Appl. Intell.* **2022**, *52*, 10307–10329. [CrossRef]
20. Wang, Z.; Yao, L. Control allocation technology based on fault diagnosis for the unmanned aerial vehicle system subject to physical constraints and fault reconfiguration mismatch. *Asian J. Control* **2022**, *25*, 1675–1683. [CrossRef]
21. Yang, R.; Cheng, L. Path planning of restaurant service robot based on a-star algorithms with updated weights. In Proceedings of the 2019 12th International Symposium on Computational Intelligence and Design (ISCID), Hangzhou, China, 14–15 December 2019; pp. 292–295. [CrossRef]
22. Shang, E.; Dai, B.; Nie, Y.; Zhu, Q.; Xiao, L.; Zhao, D. A Guide-line and Key-point based A-star Path Planning Algorithm For Autonomous Land Vehicles. In Proceedings of the 23rd IEEE International Conference on Intelligent Transportation Systems (ITSC), Rhodes, Greece, 20–23 September 2020. [CrossRef]
23. Sun, J.; Sun, Z.; Wei, P.; Liu, B.; Wang, Y.; Zhang, T.; Yan, C. Path Planning Algorithm for a Wheel-Legged Robot Based on the Theta* and Timed Elastic Band Algorithms. *Symmetry* **2023**, *15*, 1091. [CrossRef]
24. Ammar, A.; Bennaceur, H.; Chaari, I.; Koubaa, A.; Alajlan, M. Relaxed Dijkstra and A* with linear complexity for robot path planning problems in large-scale grid environments. *Soft. Comput.* **2016**, *20*, 4149–4171. [CrossRef]
25. Harabor, D.; Grastien, A. The JPS pathfinding system. In Proceedings of the 5th Annual Symposium on Combinatorial Search, Niagara Falls, ON, Canada, 19–21 July 2012; pp. 207–208. [CrossRef]
26. Harabor, D.; Grastien, A. Online graph pruning for pathfinding on grid maps. In Proceedings of the AAAI Conference on Artificial Intelligence, San Francisco, CA, USA, 7–11 August 2011; pp. 1114–1119. [CrossRef]

27. Fahed, J.; Mohammed, H. Exploiting Obstacle Geometry to Reduce Search Time in Grid-Based Pathfinding. *Symmetry* **2020**, *12*, 1186. [CrossRef]
28. Toroslu, I.H. The Floyd-Warshall all-pairs shortest paths algorithm for disconnected and very sparse graphs. *Softw. Pract. Exp.* **2023**, *53*, 1287–1303. [CrossRef]
29. Yang, J. Application of Floyd Algorithm in the Design of a Coastal Tourism Route Optimization System. *J. Coastal. Res.* **2020**, *106*, 668–671. [CrossRef]
30. Cai, J.; Hao, J.; Yang, H.; Zhao, X.; Yang, Y. A review on semi-supervised clustering. *Inf. Sci.* **2023**, *632*, 164–200. [CrossRef]
31. Bai, L.; Liang, J.; Zhao, Y. Self-Constrained Spectral Clustering. *IEEE Trans. Pattern Anal.* **2023**, *45*, 5126–5138. [CrossRef]
32. Wen, J.; Xu, Y.; Liu, H. Incomplete Multiview Spectral Clustering with Adaptive Graph Learning. *IEEE Trans. Cybern.* **2020**, *50*, 1418–1429. [CrossRef]
33. Cariou, C.; Le, S.; Chehdi, K. A Novel Mean-Shift Algorithm for Data Clustering. *IEEE Access* **2022**, *10*, 14575–14585. [CrossRef]
34. Sinaga, K.P.; Yang, M.S. Unsupervised K-Means Clustering Algorithm. *IEEE Access* **2020**, *8*, 80716–80727. [CrossRef]
35. Zhou, W.; Wang, L.; Han, X.; Wang, Y.; Zhang, Y.; Jia, Z. Adaptive Density Spatial Clustering Method Fusing Chameleon Swarm Algorithm. *Entropy* **2023**, *25*, 782. [CrossRef]
36. Cheng, D.; Xu, R.; Zhang, B.; Jin, R. Fast density estimation for density-based clustering methods. *Neurocomputing* **2023**, *532*, 170–182. [CrossRef]
37. Jain, P.; Bajpai, M.; Pamula, R. A Modified DBSCAN Algorithm for Anomaly Detection in Time-series Data with. *Int. Arab. J. Inf. Technol.* **2022**, *19*, 23–28. [CrossRef]
38. Chen, Y.; Tang, S.; Pei, S.; Wang, C.; Du, J.; Xiong, N. DHeat: A Density Heat-Based Algorithm for Clustering with Effective Radius. *IEEE Trans. Syst. Man Cybern. Syst.* **2018**, *48*, 649–660. [CrossRef]
39. Chen, Y.; Tang, S.; Zhou, L.; Wang, C.; Du, J.; Wang, T.; Pei, S. Decentralized clustering by finding loose and distributed density cores. *Inf. Sci.* **2018**, *433*, 510–526. [CrossRef]
40. Akopov, A.S.; Beklaryan, L.A.; Thakur, M. Improvement of Maneuverability Within a Multiagent Fuzzy Transportation System with the Use of Parallel Biobjective Real-Coded Genetic Algorithm. *IEEE Trans. Intell. Transp.* **2022**, *23*, 12648–12664. [CrossRef]
41. Zhu, L.; Zhu, J.; Bao, C.; Zhou, L.; Wang, C.; Kong, B. Improvement of DBSCAN Algorithm Based on Adaptive Eps Parame-ter Estimation. In Proceedings of the International Conference on Algorithms, Computing and Artificial Intelligence (ACAI 2018), Sanya, China, 21–23 December 2018; pp. 1–7. [CrossRef]
42. Du, H.; Zhai, Q.; Wang, Z.; Li, Y.; Zhang, M. A Dynamic Density Peak Clustering Algorithm Based on K-Nearest Neighbor. *Secur. Commun. Netw.* **2022**, *2022*, 7378801. [CrossRef]
43. Wang, L.; Wang, H.; Han, X.; Zhou, W. A novel adaptive density-based spatial clustering of application with noise based on bird swarm optimization algorithm. *Comput. Commun.* **2021**, *174*, 205–214. [CrossRef]
44. Chen, S.; Yi, M.; Zhang, Y.; Hou, X.; Shang, Y.; Yang, P. A self-adaptive DBSCAN-based method for wafer bin map defect pattern classification. *Microelectron. Reliab.* **2021**, *123*, 114183. [CrossRef]
45. Chen, F. An Improved DBSCAN Algorithm for Adaptively Determining Parameters in Multi-density Environment. In Proceedings of the 2nd International Conference on Artificial Intelligence and Information Systems (ICAIIS), Chongqing, China, 28–30 May 2021; pp. 1–4. [CrossRef]
46. Chen, Y.; Hu, X.; Fan, W.; Shen, L.; Zhang, Z.; Liu, X.; Du, J.; Li, H.; Chen, Y.; Li, H. Fast density peak clustering for large scale data based on KNN. *Knowl.-Based Syst.* **2020**, *187*, 104824. [CrossRef]
47. Chen, Y.; Zhou, L.; Pei, S.; Yu, Z.; Chen, Y.; Liu, X.; Du, J.; Xiong, N. KNN-BLOCK DBSCAN: Fast Clustering for Large-Scale Data. *IEEE Trans. Syst. Man Cybern. Syst.* **2021**, *51*, 3939–3953. [CrossRef]
48. Kang, Z.; Pan, H.; Hoi, S.C.H.; Xu, Z. Robust Graph Learning from Noisy Data. *IEEE. Trans. Cybern.* **2020**, *50*, 1833–1843. [CrossRef] [PubMed]
49. Zhou, G.; Zhou, X.; Li, W.; Zhao, D.; Song, B.; Xu, C.; Zhang, H.; Liu, Z.; Xu, J.; Lin, G.; et al. Development of a Lightweight Single-Band Bathymetric LiDAR. *Remote Sens.* **2022**, *14*, 5880. [CrossRef]
50. Shukla, S.; Banka, H. Monophonic music composition using genetic algorithm and Bresenham's line algorithm. *Multimed. Tools Appl.* **2022**, *81*, 26483–26503. [CrossRef]
51. Bi, W.; Ma, J.; Zhu, X.; Wang, W.; Zhang, A. Cloud service selection based on weighted KD tree nearest neighbor search. *Appl. Soft Comput.* **2022**, *131*, 109780. [CrossRef]
52. Zhao, Y.; Zhang, J.; Fu, C.; Xu, M.; Moritz, D.; Wang, Y. KD-Box: Line-segment-based KD-tree for Interactive Exploration of Large-scale Time-Series Data. *IEEE Trans. Vis. Comput. Graph.* **2022**, *28*, 890–900. [CrossRef] [PubMed]