An approximate path finding algorithm for bridging special obstacles

Sun Qiaoyu¹,², Pan Yinrong²
¹Department of Electronic Engineering, Huaihai Institute of Technology, Lianyungang, Jiangsu, China
qiaoyusun@163.com
²Department of Computer Science and Technology, East China Normal University, Shanghai, China

Received 11 February 2008; Accepted 11 November 2008

ABSTRACT
This paper presents an algorithm, which is used to find the shortest path between two given nodes in a grid with sparse polygon obstacles. Among these obstacles some of them may be bridged. The algorithm executes the search using a “don’t change direction” heuristic along the line towards the target node. The path has eight searching directions, which are decided by the opposition of sources and target nodes. According to the result of many contrastive and on-the-fly experiments, the algorithm reduces the size of searching region. The path length is also shorter than those found by using traditional algorithms. Performance analysis shows that this algorithm is efficient.

Key words: Shortest path, sparse obstacle, minimum detour, grid graph.

1. INTRODUCTION
Finding shortest paths in the presence of obstacles in a grid network is an important problem in robotics, VLSI design, and geographical information systems. In VLSI design, circuit components or previously laid out wires are treated as obstacles. On the other hand, finding an obstacle-avoiding shortest path between a pair of nodes is a fundamental operation used in many layout algorithms. There are two fundamental classes of shortest path algorithms: maze-running algorithms and line-search algorithms.

Since the maze grid is a special kind of graph, many of its properties can be used to simplify the problems of finding the shortest path between two given points in a grid network. The maze-running algorithms can be characterized as target-
directed grid propagation. The first such algorithm was Lee’s algorithm [1], which is an application of the breadth-first shortest path search algorithm. The run time of Lee’s algorithm in a typical worst case requires $O(mn)$ for $m\times n$ grid graphs.

There are many variations and modifications of the original Lee’s algorithm. Some of them reduced the memory requirement and others reduced the run time requirement. Line search algorithms [1] have been proposed to improve performance. Since such algorithms search a path by locating a sequence of line segments of variable lengths, they save memory and quickly find a simple-shaped path. However, some line-search algorithms do not guarantee the finding of the shortest path. Almost all of these algorithms are based on a graph that is sparser than the original grid and contains a path between the pair of given points. Such graph is known as a connection graph or a track graph. The track graph is not a strong connection graph in the sense that it may not contain a shortest path between a source node and a target node. However, their algorithms are able to detect the track graph and handle such cases appropriately in order to obtain shortest paths [1].

The path programming technique is an important branch of the robot research and is used to obtain an optimum path between two nodes in the work plane according to one or more rules (e.g. minimum work cost, shortest path length or least time etc.) which are able to avoid obstacles.

Historical robots were created to replace human in doing laborious work, and hence they in fact would ideally imitating human’s actions for such work. While human is able to quickly avoid or bridge an obstacle in a grid network in real life, robots are not able to avoid such obstacles in a simple way. It is hoped that the moveable robots is able to bridge an obstacle if it has a concave surface and is sufficiently narrow. In such cases the length of the path are shorter.

This paper proposes a new obstacle-avoiding path find algorithm which may be used to bridge some obstacles in grids. There are eight directions in expanding the obstacle-avoiding path and is able to reduce the length comparatively more than those with four expanding directions (The four directions are up, down, right and left [11]). The expanding direction of a path is usually directed towards the target node until it hits an obstacle.

When an obstacle is hit, it is required to search all of the obstacles extremes in order to decide if it is necessary to go around the obstacle. If an obstacle can be bridged, it is expanded directly above it as if it is not an obstacle. However if one needs to go around an obstacle, an extreme which diving the path less detour to expand must be chosen. This may change the current direction of the path. After it goes around the last extreme of the obstacle, the path will change
its direction according to the opposition of target node. The proposed algorithm is able to find the shortest path between a pair of given nodes in a simple way and has high precision.

This paper is organized as below. First the searching technique is introduced in section 2. Various different situations are considered and the methods of handling these situations are discussed. Second implementation issues are discussed using a pseudo-language approach in section 3. Some performance of the proposed algorithm has been studied and results are presented in section 4. Finally conclusion is drawn.

2. THE SEARCHING TECHNIQUE OF THE ALGORITHM

This algorithm finds the shortest path between two given nodes in an N × N grid containing sparse obstacles which occupy less than 10% of the total area of the grid. There are two kinds of elements in the grid: obstacle elements and nonobstacle elements. An obstacle element is always situated next to some neighboring obstacle elements. A source node is a nonobstacle element at which search journey begins. A target node is a nonobstacle element at which search journey ends. The distance between two neighboring elements is of unit length. The area of an element is 1. The obstacle elements are randomly distributed across the network.

![Fig. 1. Definition of expanding directions.](image-url)
Each node has eight expanding directions: up ($dir = 2$); down ($dir = -2$); left ($dir = -3$); right ($dir = 3$); right-up ($dir = 4$); right-down ($dir = 5$); left-up ($dir = -5$); left-down ($dir = -4$). Here $dir$ denotes the direction in which an integer is used to represent it.

Suppose the coordinates of the source and the target node are denoted as $S(X_s, Y_s)$ and $T(X_t, Y_t)$ respectively. The length of the shortest path between the two points is $\sqrt{2} \min (|Y_t - Y_s|, |X_t - X_s|) + ||Y_t - Y_s| - |X_t - X_s||$ if there is no obstacles. It will increase when hitting obstacle. The difference is decided by the detour length when the path tries to avoid the obstacles. In essence the path should decide on its expanding direction which has a less detour length.

The algorithm executes the search using a “don’t change direction” heuristic along the line towards the target node. The algorithm includes several key components as described below.

2.1. The original searching direction
The searching direction $dir$ is determined according to the coordinates of the source node and the target node as below:

- If $X_t > X_s$ and $Y_t = Y_s$: $dir = 3$;
- If $X_s > X_t$ and $Y_t = Y_s$: $dir = -3$;
- If $X_t > X_s$ and $Y_t > Y_s$: $dir = 4$;
- If $X_t < X_s$ and $Y_t < Y_s$: $dir = -4$;
- If $Y_t > Y_s$ and $X_s = X_t$: $dir = 2$;
- If $Y_t < Y_s$ and $X_s = X_t$: $dir = -2$;
- If $Y_t < Y_s$ and $X_t > X_s$: $dir = 5$;
- If $Y_t > Y_s$ and $X_t < X_s$: $dir = -5$.

2.2. Operations after hitting an obstacle
If the search hits an obstacle one needs to work out whether the obstacle has a concave surface or not. In addition the width of the obstacle must be smaller than the step length of robot, i.e. $W_m$. In other words if $(X_f - X_b) < W_m$ the obstacle may be concavely bridged and return onto its original direction.

The concept is depicted in Figure 2. Let the shadow filled with dots represents a concave surface such that

$$|X'_f - X'_b| < W_m$$

It is obviously that $B_2$ is an obstacle that can be bridged.
If the obstacle cannot be bridged, search the coordinate of the obstacle’s extreme and look for the direction around the obstacle. The direction which gives the path a shorter detour should be selected.

In case one of the edges of the obstacle is adjacent to the border of grid, the path needs to go around the obstacle in the opposite direction.

Suppose the current direction point to ‘right’. When the path hits an obstacle one needs to calculate the smaller detour of the path. As depicted in Figure 3, supposing...
It is easy to show that

\[ |y_j - y_n| + |y_n - y_a| = |y_j - y| + |y_h - y_j| = |y_h - y_j| = L \]

It is easy to show that

\[ |y_h - y_n| + |y_n - y_j| = L - |y_j - y_a| + |y_n - y_j| , \]

and

\[ |y_j - y_n| + |y_j - y_a| = L - |y_h - y_j| + |y_j - y_a| . \]

Assuming that

\[ |y_j - y_n| + |y_j - y_a| \geq |y_h - y_j| + |y_h - y_a| , \]

it can be shown that

\[ L - |y_h - y_j| + |y_j - y_a| \geq L - |y_j - y_a| + |y_h - y_j| , \]

leading to \( |y_j - y_a| \leq |y_h - y_j| \) and hence \( y_m = y_h \).

As a result the path will change its direction towards right-up.

If the opposition of the target node is out of the band region of the obstacle (the shadow area as Figure 4), the path will select its direction directly according the vertical coordinate of the source node and the target node. In order to reduce the detour of the path, this process will stop when \( y_n = y_{m'} \).

Fig. 4. Selection of the expanding direction when the path hits an obstacle that the target node is outside of the shadow area.
There are two circumstances that might happen while going round an obstacle. These are explained below.

2.2.1. Expanding to the extreme of the obstacle B₁
In Figure 5(a) when the path is expanded to the top extreme $Y_h$ is the vertical coordinate, $X_b$ and $X_f$ being are the horizontal coordinates of the nearer side and the farther side of the extreme respectively; and $Y_l$ is the vertical coordinate of the bottom extreme, its direction is changed towards the right and expand to the coordinates $(X_f, Y_h)$.

Choosing the expanding direction from eight original direction according to the opposition of current node and target node and expanding the path until it hits an other obstacle.

When the path expands to the edge of obstacle B, as in Figure 5(b), where it’s the coordinates are given by $(X_b, Y_n)$. The extreme being selected is not $(X_b, Y_n)$. The best route is to change the current direction towards up and expand to the coordinates $(X_b, Y_m)$, where $Y_m$ is the vertical coordinate of the selected extreme such that $Y_m = Y_h$. It is then followed by changing the direction towards the right and expand to the coordinates $(X_f, Y_h)$. Choose the expanding direction according to the opposition of the current node and target node and go on expanding until it hits an obstacle.

2.2.2. Hitting the obstacle B₂ during expanding
In this situation one should have chosen to go around B₂ first as shown in Figure 6. If the obstacle is to the right of path, it will then go back to $Y_m$, i.e. the vertical coordinate of the bottom extreme B₂, and changing the expanding direction to the same as the horizontal sub_direction of the current direction.
In Figure 6 (a), the direction is in the right until expanding to the coordinates \((X_f', Y_m)\). It should be carried on in expanding in the original direction (right-up) until the path expands to the coordinates \((X_b, Y_m)\).

Referring to Figure 6(b). Suppose the path hits a new obstacle \(B_3\) before it arrives \((X_b, Y_m)\) and \(B_3\) is at the top of path, the expanding direction is required to be changed to the same as the horizontal sub_direction of the current direction. As shown in Figure 6 (b) the direction is in the right until expanding to \((X_f', Y_m)\). It should be carried on expanding in the original direction (right-up) until it hits \(X_b\) or expands to \(Y_m\).

2.2.3. Hitting the obstacle \(B_2\) while going round the last extreme of \(B_1\)
An \(x-y\) convex polygonal obstacle may be divided into some adjacent rectangle obstacles as depicted in Figure 7.
The path has expanded to \((X_b, Y_m)\) in the right-up direction. It will change its expanding direction to right and expand. Before expand to the coordinates \((X_f, Y_m)\), it hits obstacle B\(_2\). Since B\(_2\) is adjacent to B\(_1\), the path should go back to its last corner (node M) and go around B\(_2\). It will then expand in the right-up direction which has been selected as the path hits B\(_1\) until the path expands to \((X_f', Y_m)\). The expanding direction should then be chosen as according to the opposition of the current and target nodes and should be carried on expanding until it hits an obstacle.

2.3. Tracing out the path

This part of the algorithm is used to record the information at each corner. Information such as coordinates, expanding direction and tracing direction of the path in a sequence should be recorded and a stack is usually required to store the information of these corners. After the searching process, the path is to be traced out according to the tracing direction of the corners.

3. ALGORITHMIC IMPLEMENTATION

Suppose \(s\) is a source node, \(t\) is the target node, \(n\) is the current node; There are three basic features of a node: the coordinates \((X, Y)\), the expanding direction \((dir)\) and the tracing direction \((state)\). The basic features of the current node \(n\) are denoted with the subscript \(n\) as \((X_n, Y_n), dir_n\) and \(state_n\); that of the source \(s\) are \((X_s, Y_s), dir_s\) and \(state_s\); and that of the target node \(t\) are \((X_t, Y_t), dir_t, state_t\).

A stack stores the tracing direction of each corner in sequence. So the basic features of the top cell \(head\) are \((X_{head}, Y_{head}), dir_{head}, state_{head}\).

Using \(count\) to denote the number of nodes that have been searched and noting that the initial value of \(count\) is 0. In each expanding step \(count\) will be
incremented by one. Count will be accessed before each expanding step. If count > N*N, the algorithm is stopped and a failure information is shown.

Suppose $X_m$ and $Y_m$ are the coordinates of the extreme of obstacle. The extreme is selected when the path goes around the obstacle. Let $W_m$ be the maximum width of the concave obstacle which can be bridged and Length be the length of path. $i$ and $j$ are flags of the expanding direction.

**Step 1:** $s \rightarrow n$.
//Process of searching path

**Step 2:**

```pseudocode
WHILE $(X_n \neq X_i \| Y_n \neq Y_i)$
{
    //Case 1: Expanding along horizontal direction
    WHILE $(X_n \neq X_i)$
    {
        Determine $dir_n$ as Figure 1;
        PUSH $(n)$;
        WHILE $(X_n \neq X_i \&\& \text{ not hit obstacle})$
        Expand along $dir_n$ and then store $state_n$;
        //Operation of horizontal expanding when hit obstacle
        IF (hit obstacle)
        {
            Searching obstacle’s extremes: $Y_h, Y_l, X_b, X_f$;
            IF (the obstacle can be briged)
            {
                WHILE $(X_n \neq X_f)$
                Expand along $dir_n$ and then store $state_n$;
                BREAK;
            }
            //Target node is inside of the band region of obstacle
            ELSE IF $(Y_l < Y_i < Y_h)$
            Choose the direction around the obstacle as Fig. 3;
            //Target node is outside of the band region of obstacle
            ELSE
                Go round the obstacle as Figure 4.
        }
        //Target node is inside of the band region of obstacle
        ELSE IF
       Choose the direction around the obstacle as Fig. 3;
        //Target node is outside of the band region of obstacle
        ELSE
            Go round the obstacle as Figure 4.
        IF (The obstacle and the boundary are neighboring)
        Change the direction to be opposite;
        $n = \text{POP}(n)$;
    }
}
```

388 An approximate path finding algorithm for bridging special obstacles
Decide the value of $dir_n$ according to $i$ and $j$;

**PUSH** ($n$);

**WHILE** ($Y_n \neq Y_m - j$ && not hit obstacle)

Expand along $dir_n$ and then store $state_n$;

**IF** (hit obstacle)

{  
//hit obstacle while expanding along horizontal
//direction

**IF** ($X_n = X_b$)

Operate as Figure 5 (b)

**ELSE**

Operate as Figure 6;

}

**ELSE**

Operate as Figure 5 (a)

}

//Case 2: Expanding along vertical direction

**WHILE** ($Y_n \neq Y_t$)

{

Determine $dir_n$ as Figure 1;

**PUSH** ($n$);

**WHILE** ($Y_n \neq Y_t$ && not hit obstacle)

Expanding along $dir_n$ and then store $state_n$;

//Operation of vertical expanding when hit obstacle

**IF** (hit obstacle)

Same operations as the horizontal expanding;

}

}

**Step 3:** The path has been found. Show success information.

**Step 4:** (Tracing process)

$dir_n = state_n$;

**WHILE** ($X_n \neq X_s$ || $Y_n \neq Y_s$)

{
Show the coordinate of current node;
\[ \text{IF} \ (X_n = X_{head} \&\& \ Y_n = Y_{head}) \]
\[ \text{dir}_n = \text{state}_{head}; \]
\[ \text{Trace back along dir}_n; \]
\}

\textbf{Step 5:} Show the length of path and stop the whole process.

\section*{4. PERFORMANCE OF ALGORITHM}

A network of $N \times N$ grid is considered. Let $K$ be the number of obstacles in the grid and $L$ be the number of obstacles that can be bridged. $L$ is smaller than $K$. All these obstacles are located stochastically covering less than 10\% of the total area of the grid.

This algorithm has two processes, a searching process and a tracing process. The searching process has an iterative loop which contains two inner loops. If an obstacle is hit the iteration process through the loop and one of the two inner loops will be executed again.

The position of an obstacle opposition is arbitrarily placed and its area is randomly assigned as well. Each obstacle may contain one or more elements. The total area of all obstacles in the grid should be less than 10\% of the total area of the grid. Let the percentage of the obstacles be $p$\%(p\leq10). The value of $K$ lies in between 1 and $p$\% of $N^2$ and the value of $L$ lies in between 0 and $K$.

\textbf{1. $K < p$\% of $N^2$}

In the worst condition, searching four extremes of an obstacle requires $2(N - 1)$ steps when the length and width of the obstacle is $N - 1$ and 1 respectively. The largest number of obstacles the searching algorithm will hit and go round is $K - L$.

The algorithm executes the search using a “don’t change direction” heuristic along the line towards the target node. Hence the maximal number of obstacles the path should round is $N/2 - 1$.

The time complexity of the process is $2(K - L) (N - 1)$. If the search is failed, the initial direction should be changed and the search must start again. Same thing happens during hitting the first obstacle twice, so the time complexity of the process is $2(K - L + 1) (N - 1)$.

The time complexity of the searching process is $2(N - 1) (K - L + 1)$. The tracing process requires $(K - L) (N - 1) + (N - 1) = (N - 1) (K - L + 1)$.

The total time complexity of the algorithm is given by


\[ 2(N - 1) (K - L + 1) + (K - L + 1) (N - 1) = 3 (K - L + 1) (N - 1) = (3K + 1 - 3L) N - 3(K - L) \]

Hence the time complexity of the algorithm is \( O (3K - 3L + 1) N) \).

If searching an obstacle requires \( 2(N - 1) \) steps, the area of the obstacle will be \( N - 1 \). Then \( K \) can be calculated as

\[ K \leq 10\% \frac{N^2}{(N - 1)}. \]

The time complexity of the algorithm is

\[ 3 (K - L + 1) (N - 1) \leq 3 (10\% \frac{N^2}{(N - 1)} - L + 1) (N - 1) = 0.3 N^2 - 3(L - 1)/(N - 1). \]

Under this condition, the time complexity of the algorithm is \( O (0.3N^2) \).

2. \( K = p\% \) of \( N^2 \)

Each obstacle composes of only one element. Searching four extremes of an obstacle requires 2 steps and the maximal number of obstacles the path should go round is \( N/2 - 1 \). The time complexity of the searching process will be \( 2*(N/2 - 1) \).

The maximal length of the path in the worst condition is given by

\[ 2(N/2 - 1) + N = N - 2 + N = 2(N - 1). \]

The worst time complexity of the algorithm is

\[ 2*(N/2 - 1) + 2(N - 1) = 3N - 4. \]

Hence the time complexity is \( O (3N - 4) \).

The algorithm proposed in this paper is related to the grid with sparse obstacles. If the percentage is increase, the time complexity will increase too. The more the percentage increases, the more the time complexity increases.

5. CONCLUSION

In robot design technique, when a robot is looking for a path in order to avoid obstacles, its expanding direction and moving step are more flexible than VLSI design. This algorithm fits the constriction of mobile robot’s expanding direction. At the same time, the length of the path can be reduced for its eight expanding directions. The algorithm proposed in this paper speeds up and reduces the searching region and is served as an efficient approximation algorithm.
REFERENCES

[1] S. Q. Zheng, Joon Shink Lim, and S. Sitharma Iyengar. Finding Obstacle-Avoiding Shortest Paths Using Implicit Connection Graphs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 15, No. 1, 1996, pp. 103–110.

[2] T. Ohtsuki. Maze-running and Line-search algorithms. *IEEE Trans. on Electron. Comput.*, Vol. EC-16, 1967, pp. 97–98.

[3] R. C. Larson, V. O. K. Li. Finding Minimum Rectilinear Distance Paths in The Presence of Barriers. *Networks*, 1981, 11: 285–304.

[4] Jeffrey H. Hoel. “Some Variations of Lee’s Algorithm”. *IEEE Trans. on Electron. Comput.*, Vol. 25, No. 1, 1976, pp. 19–24.

[5] Narsingh Deo & Chi-yin Pang. Shortest-Path Algorithms: Taxonomy and Annotation. *Networks*, 1984 14:275–323

[6] Hadlock F. O. A Shortest Path Algorithm for Grid Graphs. *Networks*, 1977, 7: 323–334.

[7] D. T. Lee, F. P. Freparata. Euclidean Shortest Paths in The Presence of Rectilinear Barriers. *NETWORKS*. 1984 14:393–410.

[8] Zhou Zhi, Chen Guoliang, Gu Jun. Finding Obstacle Avoiding Shortest Path Using Connection Graph With O(t log t) Edges. *Chinese Journal of Computers*, 1999, 22(5): 519–524.

[9] Zhang Farong. “A Searching Algorithm for Approximately Shortest Path Among Obstacles”. *Computer Engineering*, Vol. 25, No. 3, 1999, pp. 15–16.

[10] Sun Qiaoyu, Pan Yinrong etc., “An Approximately Maze-Algorithm for a Shortest Path with Sparse Obstacles”. *Computer Applications And Software*, Vol. 20, No. 5, 2003, pp. 37–39.

[11] Sun Qiaoyu etc., “A Maze-Algorithm with Eight Directions”. *Computer Engineering*, Vol. 30, No. 1, 2004, pp. 90–91.