Simple Application of A* Algorithm on Robot path-planning for the Shortest Path

Rongshen Yin

School of Information Science and Technology, Shanghai Tech University, Shanghai, China

yinrsh2022@shanghaitech.edu.cn

Abstract. When the robot encounters circumstances where it has to go from its current position to a given goal in an undetermined environment, it has to deal with the path-planning problem. The A* algorithm, as a classic algorithm for efficient path searching in robot engineering, is an interesting and precious topic to study. As part of a project to design a cargo-delivering robot which will operate in a multi-purpose stadium, considering the complex structure of the building, an efficient path-planning algorithm must be implemented to help the robot fulfil its delivery. The heuristic function and its parameters, which are important for the speed of making decisions, should be determined under tests in different situations. By using MATLAB, the detailed process of the A* algorithm using different heuristic functions and parameters can be clearly shown in a 2-D grid map simulating the scene, including searched areas and the final path. With the help of the images and the output path length, the heuristic function and the parameters are finally decided. The method is now good for the designed robot to operate in simulations with static obstacles, but it can be improved to deal with automatic obstacle avoidance in real-life cases, where people, balls, etc. are actually moving.

Keywords: Mobile robot; path planning; A* algorithm; heuristic function

1. Introduction

Path planning is one of the key technologies for mobile robot navigation. Mobile robot path planning refers to planning the best or sub-optimal safe collision-free path from the start point to the endpoint in a complex environment [1]. In life cases, most of the situations won't stay simple and unchanged. Giving the robot the ability to plan for the shortest path in unknown environments, including random obstacles, starting points and goals, is vital for the robot to be practically used.

When the robot is low on battery and needs to be recharged, or urgent delivery orders are made where the robot has to reach the customers as soon as possible, the robot has to plan for the shortest path. The scenes require the robot to consume less energy while making fast and accurate decisions. A static scene in a multifunctional stadium with the walls and doors of the stadium as the settled obstacles is used to study the characters of the algorithm. The possible random obstacles such as people, balls and garbage, are considered static but at random places at each test. Although the scene is not practical enough, it is suitable to test simple static path planning algorithms, like A* and Rapidly Exploring Random Trees (RRT).

The path-planning algorithms intend to reach the goal state of the robot and return a quality-concerning path. The definition of the quality-concerning path is relative and keeps varying depending on robots and their applications. Sometimes it is about finding the shortest path and at others, concern is about safety. Sometimes the quality takes time consumption into account, but it might care about route length in other cases [2]. In this project, the quality emphasized the route cost but considered other factors as well. Since in real life, small errors are acceptable. It is worth trying when sacrificing a bit of accuracy for a faster algorithm.

In previous studies, detailed comparisons of various path planning algorithms such as the A*, the D*, the RRT, the RRT*, etc. have been made and simulated. However, the conclusions cannot be straightly used in the test case. For a unique situation, the choice of algorithm and the tests on the parameters should be reconsidered.

In the project, A* is picked after comparing it with the RRT algorithm. The reasons will be stated later. In path planning, the A* algorithm might fall into an infinite loop, which has the defects of poor

real-time performance, large amounts of calculation for each node and long operation time [3]. Therefore, some optimizations to its heuristic function and parameters can give it a hand. Some current references have given their ways. For example, combining an optimized heuristic function with the two-way search method [4], mixing 4-direction and 8-direction movement and dynamically changing the search region [5], and replacing unnecessary nodes with screen hops to reduce computations [6].

For example, dynamically enlarge the heuristic function to make the decision faster and use a closed set to store the nodes if they have been expanded [7]. Also, since the simulation of the algorithm is in a grid map, it's rather easy to create the nodes and edges. The final effect of the algorithms will be output as images using MATLAB.

2. Choice of Algorithm

Let's start with the differences between A* and RRT. The RRT is a data structure to plan paths using non-holonomic and differential constraints. It is a tree structure formed by uniformly distributed random points starting at the initial robot position. It randomly generates a vertex on the plain, connects the closest existing vertex in the tree to the newly generated one and creates the actual node on that line with unit weight on the edge, until the new vertex is inside a range of the goal. When encountering an obstacle, reject the new vertex and generate another. The RRT does not need to build a map [8]. To determine a motion plan, only a single query is needed. This makes the RRT and its variants a viable option for robot trajectory planning in dynamic environments with real-time requirements. Although it is easy for an unknown environment, the algorithm may never return a true answer because it only finds the relatively shortest path among all the routes it has expanded. Thus, it is simple but suboptimal.

The A* algorithm is an upgraded version of the Dijkstra algorithm. It combines the Dijkstra algorithm and greedy algorithm and is a heuristic search algorithm. Given a graph of vertices, the Dijkstra iterates throughout the graph. In each iteration, it picks the closest vertex A from the start as the expansion point and checks possible shorter paths from the goal to its neighbors passing A. The algorithm will check most of the vertices until the goal is selected as the expansion point, which is time-consuming. Therefore, an assumed distance from the vertices to the goal is introduced in A*, denoted as h(n). Instead of updating the true distance from the current location to the start depending on the true distance, shown as g(n), the A* algorithm picks the shortest hypothesised whole distance from the start to the goal after passing a certain vertex n, denoted as f(n), which is the heuristic function.

$$f(n)=h(n)+g(n) \tag{1}$$

The heuristic function gives the information of the goal. It acts like a rope connecting the robot and the goal, which keeps dragging the former to the latter. In this way, the choice of heuristic function should be considered. A good heuristic function can speed up the search while ensuring the accuracy. On the other hand, a bad way, say the assumption of the function is much larger than the true value, may result in a wrong path since the assumption is too optimistic that the robot takes turns on the wrong corners. Unlike the RRT, the A* requires a detailed scene map. Therefore, it is suitable for a known environment.

Comparing the two algorithms, the time complexities are quite the same. On each step, both of them should pick a vertex with the shortest distance as an expansion point and update the neighbors.

Suppose a graph has V vertices and E edges, the time complexity of the A* algorithm using heaps as storage and graph search is

$$T=O(E*\log(V)+ V*\log(V))=O(E*\log(V)) \tag{2}$$

The time complexity is the same for RRT.

Also, the RRT has no information about the goal. It will search throughout the map. But the A* can only browse the necessary ones. These two reasons imply that the A* should have higher efficiency. The conclusion is also proven by the previous studies.

Table 1. Computation time of A* [9]

Averages		
Processing(s)	Executing(s)	Distance(m)
0.0706	42.6095	11.3805
Standard Deviations		
Processing(s)	Executing(s)	Distance(m)
0.0144	0.1183	0.0554

Table 2. Computation time of RRT [9]

Averages		
Processing(s)	Executing(s)	Distance(m)
13.4712	48.7516	13.2409
Standard Deviations		
Processing(s)	Executing(s)	Distance(m)
6.7515	4.2092	1.1815

Table 1 and Table 2 are shown above. By comparing the total time consumption and the cost of the path, A* has the advantage in both cases.

Practically in the stadium assumption, the detailed map of the stadium is available and the positions of people, balls, etc. can be detected through a certain system (could be a visual system or portable chips) in the building. Therefore, it is feasible to generate a graph of the scene, which is good for A*. Thus A* becomes the final choice because of its higher accuracy.

3. A* Algorithm Description

As mentioned above, the choice of the heuristic function counts since it may influence the speed and accuracy of the algorithm. Trading off the two factors is the key.

There are two methods in executing the algorithm, graph search and tree search. The former means the algorithm will mark the expansion points as closed and will never check them again. The latter takes it for granted that it is searching on a tree instead of a graph. Since there is no ring in a tree, the algorithm won't mark any of the vertices as closed. That implies possible repetitive checks on a vertex, decreasing the efficiency but increasing accuracy.

It is easy to ensure the accuracy of tree search. If the assumed distance from the current vertex to the goal is always lower than the true shortest distance, the tree search will always return the shortest path. For graph search, the heuristic function should be consistent. That means the assumed distance from the current vertex to the goal is always the shortest among all possible paths from the current vertex to the goal. It also guarantees that the f(x)never decreases. As shown below, h(n) is the assumed distance from n. $c(n, n')$ is the cost from n to n'.

$$h(n) \leq c(n, n') + h(n') \tag{3}$$

If the heuristic function is consistent, graph search should be applied since it's more efficient.

The Approximation Cell Decomposition is chosen to generate the map. It is to map the real scene into a two-dimensional graph. The cost of each step in a grid can be set according to the height of hypsography. The obstacles are colored in black with infinity as their cost. This method is intuitive and feasible to all kinds of obstacles and is suitable for MATLAB simulation.

In a 2-D grid graph, there are several assumed distances $h(n)$ to choose from, such as Manhattan Distance, Diagonal distance, Euclidean distance, etc. In the experiment, the Manhattan distance and

the Euclidean distance are compared to see the effect of the heuristic functions on the algorithm. Suppose there are 2 nodes with coordinates $(a_1, a_2), (b_1, b_2)$. The Manhattan distance

$$h(n) = |b_2 - a_2| + |b_1 - a_1| \tag{4}$$

and the Euclidean distance (linear distance)

$$h(n) = \sqrt{(b\_2 - a\_2)^2 + (b\_1 - a\_1)^2} \tag{5}$$

Of course, on a 2-D grid graph, both of them are consistent.

4. Algorithm Test

The tests are intended to find the most efficient choice of the heuristic function and the parameters. It is necessary to show the difference between choices. Since the complexity of our true environment is not enough, we use MATLAB to generate random graphs where all the obstacles, the start and the goal are randomly generated.

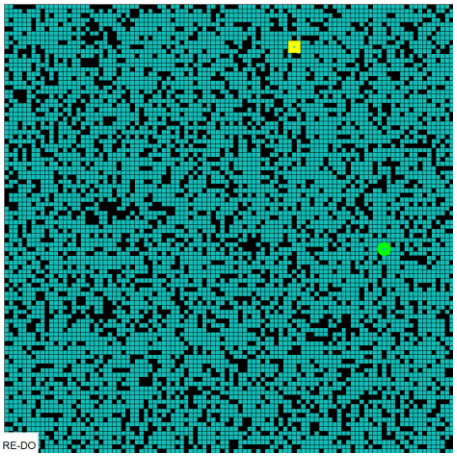

Fig. 1 Test environment (Photo credited: Original)

The random test environment created is shown in Figure 1. The green point is the start. The yellow one is the goal. And the black spots are obstacles.

First, test the difference between the two heuristic functions. Two environments were set up. One has large numbers of obstacles that require more turnings, and the other has a few obstacles so that the robot may move in straight lines.

Table 3. Comparison of heuristic function in route length and time consumption

Length (including diagonal) /Times of iteration	Manhattan	Euclidean
heavily blocked	108/305	108/2448
slightly blocked	68/650	47/1409

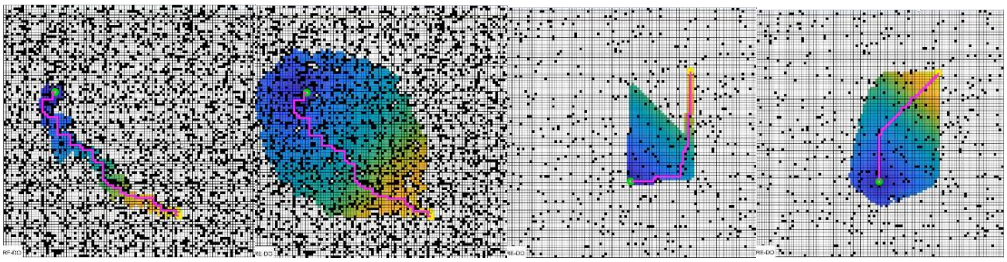

Fig. 2 Search area (colored area) and found path (purple line: a) Manhattan in the heavily blocked environment; b) Euclidean in the heavily blocked environment; c) Manhattan in the slightly blocked environment; d) Euclidean in the slightly blocked environment (Photo credited: Original)

In Table 3 above, the time of iteration stands for time consumption since the two heuristic functions run in the same environment. The Manhattan distance is always faster in both cases and works better in the heavily blocked environment. But in the real scene, the obstacles won't be as random as this. By comparison, the Euclidean distance is less heuristic. It is slightly slower in the slightly blocked environment, but it will return a shorter distance since it will find straight paths in the diagonal direction, which will save a lot of energy in moving. Therefore, we tend to select the Euclidean distance for its practicality. But it needs to be optimized for better efficiency.

In Figure 2, the colored area represents the visited area of the algorithm and the purple line represents the returned shortest way. The area of the colored plain can stand for the time consumption of the algorithm. The images visualize the differences between the two heuristic function, Manhattan Distance and Euclidean Distance, in simple and difficult circumstances.

There is a simple way to modify the heuristic function. The Euclidean distance has a higher requirement because it predicts the distance more cautiously. But being too careful sometimes lowers the calculation speed. If we multiply a parameter(p) by the heuristic function, the constraint on decisions will be loosed, thus speeding up the algorithm [10]. However, the parameter should be limited. If the prediction is too optimistic, the algorithm may return a wrong answer. The choice of the parameter should be studied.

$$f(n) = p * h(n) + g(n) \tag{5}$$

Table 4. Comparison of parameters in route length and time consumption

parameter =	0	1	2	3
Length	107	107	107	117
Times of iteration	5423	1954	162	139

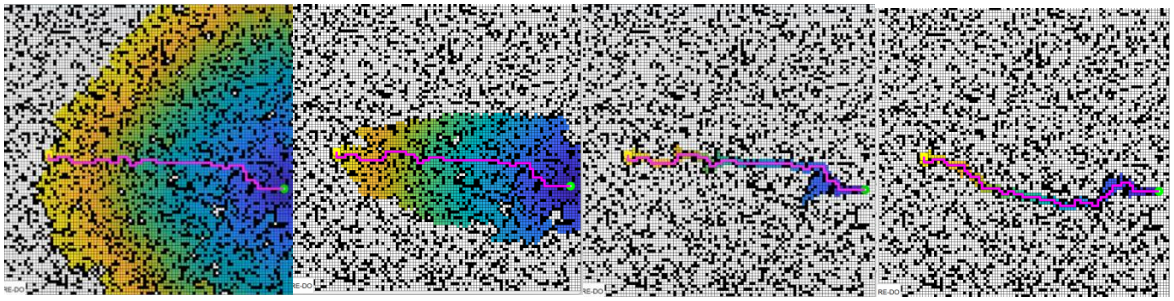

Fig 3. Search area (colored area) and found path (purple line: a) Euclidean with parameter = 0; b) Euclidean with parameter = 1; c) Euclidean with parameter = 2; d) Euclidean with parameter = 3 (Photo credited: Original)

From Table 4 above, with the parameter increasing, the algorithm becomes more heuristic and runs faster. When p=0, the A* algorithm degenerates to the Dijkstra algorithm, which transverses almost every node around and isn't heuristic. When p=1, it is the standard A* with Euclidean distance as the assumed distance. When p=2, the algorithm becomes much faster and returns a correct answer. When p=3, the algorithm found a longer path, which is a wrong answer. The test proves the hypothesis above. After several tests, p=2 is determined.

In Figure 3, the colored area represents the visited area of the algorithm and the purple line represents the returned shortest way. The area of the colored plain can stand for the time consumption of the algorithm. The images visualize the differences of choosing different parameters.

5. Scene Simulation

After determining the algorithm, it can be implemented in the real scene. The scene is a part of the stadium with 11 sections of different shapes. The obstacles are walls, doors, seats, possible people and trash. The walls and doors are settled obstacles, and the small obstacles are randomly generated.

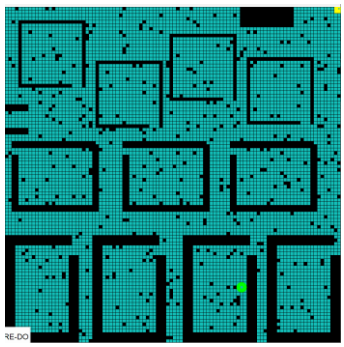

Fig. 4 Simulated Environment (Photo credited: Original)

The simulated test environment of the stadium created is shown in Figure 4. The green point is the start. The yellow one is the goal. And the black spots are obstacles.
The created simulating environment is shown above. Black spots are small obstacles, black lines are walls and doors, the green spot is the start, and the yellow one is the goal

Then run the adjusted A* algorithm on the graph of 5 different conditions. Meanwhile, we compared the optimized Euclidean method with the other two methods, unoptimized Manhattan and optimized Manhattan to double-check it.

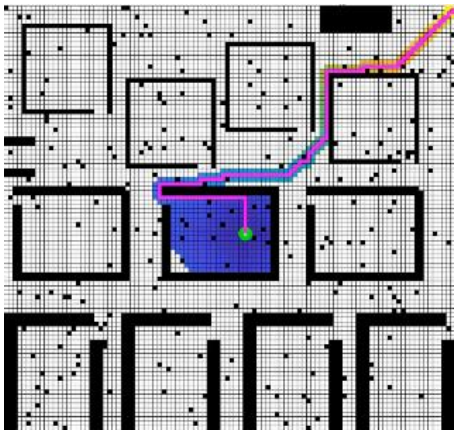

Fig. 5 Example of path-planning in the real scene (Photo credited: Original)

An example of path planning in the simulated scene is shown in Figure 5.

Table 5. Comparison of different methods in route length and time consumption

Euclidean (optimized）	1	2	3	4	5
Iteration	165	214	145	246	172
Length	89	65	118	106	87
Manhattan （unoptimized)					
Iteration	316	367	420	315	508
Length	98	71	121	117	96
Manhattan (scaled)					
Iteration	105	248	130	119	241
Length	102	71	127	119	100

Table 5 still implies that optimized Euclidean is the best since most of its results are optimal and increase efficiency. The unoptimized Manhattan acts as a reference, showing standard answers. However, for the optimized Manhattan, we can see it’s unstable since it usually gives wrong answers and doesn’t have a speed advantage over Euclidean.

Still, in the experiment, the shortage of direct multiplication was exposed. The method will cause the heuristic function to be neither admissible nor consistent, which leads to some suboptimal solutions.

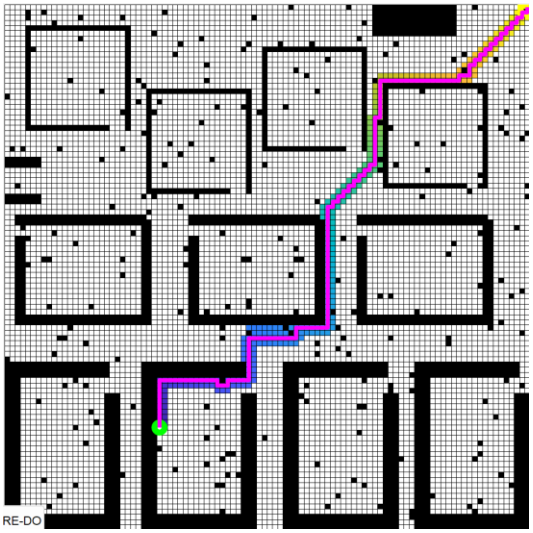

Fig. 6 Example of solving suboptimal solutions (Photo credited: Original)

An example of solving suboptimal solutions in the simulated scene is shown in Figure 6.

Table 6. Comparison of parameters in route length between static scalar and dynamic scalar

Length/ Iteration	1	2	3	4	5
scalar = 2	150/ 165	152/ 162	154/ 165	152/ 165	150/ 163
dynamic	150/ 341	150/ 340	150/ 338	150/ 340	150/ 331

In the circumstance shown in Table 6, the optimal solution generated by Dijkstra is 150. When the scalar equals 2, the algorithm returns suboptimal solutions for 3 times in 5. Although the error is small in the simulation, it might cause greater problems in real cases. To reduce the error while keeping the speed of the algorithm, the scalar can be changed during the search.

As the expanding point gets closer to the goal, the scalar will be reduced to be more careful. As the code here shows:

```
if(heo_func(i) >= 10)
        scalar = 2;
    elseif(heo_func (i) < 10 && heo_func(i) >= 5)
        scalar = 1.8;
    elseif(heo_func (i) < 5 && heo_func(i) >= 2)
        scalar = 1.5;
    else
        scalar = 1;
    end
```

In this way, the accuracy of the algorithm will be improved while limiting the losses on the speed.

6. Conclusion

By studying the differences between various path-planning algorithms and visually simulating with MATLAB, a relatively efficient A* algorithm is found in the test case, which uses the Euclidean distance as the heuristic function and multiplying it by 2. And considering the trade-off between speed and accuracy, the dynamic scalar of the heuristic function can be applied. The algorithm is fast in

calculation and the error is acceptable. The simple implementation of the A* algorithm is successful. However, there are still many aspects that we need to optimize. For example, small adjustments can be made to the algorithm structure. The priority queue can replace the simple array since it will maintain itself and always put the smallest member on the top. Using the structure on the open set in the algorithm can greatly decrease the time complexity on each iteration, improving the efficiency. Speaking of the static path planning itself, it is not the best solution for our scene. The people are always moving, and there might be balls and rubbish that suddenly appear in front of the robot. For unexpected situations, automatic obstacle avoidance should be implemented and a dynamic algorithm should be chosen, for example, the dynamic Dijkstra (D*) algorithm. The further work might be done in the future.

References

[1] M. Imran and F. Kunwar, "A hybrid path planning technique developed by integrating global and local path planner," 2016 International Conference on Intelligent Systems Engineering (ICISE), Islamabad, Pakistan, 2016, 118-122.

[2] C. Yi and X. Hongtu, "Global Dynamic Path Planning Based on Fusion of Improved A* Algorithm and Morphin Algorithm," 2019 Chinese Control And Decision Conference (CCDC), Nanchang, China, 2019, 6191-6196.

[3] Y. Huang and S. Guo, "Path planning of mobile robots based on improved A* algorithm," 2022 Asia Conference on Advanced Robotics, Automation, and Control Engineering (ARACE), Qingdao, China, 2022, 133-137.

[4] Lu Yi, Gao Yongping, Long Jiangteng. Research on A~* Algorithm in Path Planning of Mobile Robots [J]. Journal of Hubei Normal University (Natural Science Edition), 2022, 42(02): 59-65.

[5] Fu Lixia, Ren Yujie, Zhang Yong, Mao Jianlin. Path Planning of Mobile Robots Based on Improved Smooth A~* Algorithm [J]. Computer Simulation, 2020, 37(08): 271-276.

[6] Chen Jinghui, Cui Yan,Liu Xinglin, Li Yuqiang. Path planning method for mobile robots based on improved A~* algorithm [J]. Computer Application Research, 2020, 37(S1): 118-11 9.

[7] C. Wang and J. Mao, "Summary of AGV Path Planning," 2019 3rd International Conference on Electronic Information Technology and Computer Engineering (EITCE), Xiamen, China, 2019, pp. 332-335.

[8] S. M. LaValle, “Rapidly-exploring random trees: A new tool for path planning,” Compute. Sci. Dept., Iowa State Univ., Ames, IA, USA, Tech. Rep. 1998, 98-11.

[9] Braun, J.; Brito, T.; Lima, J.; Costa, P.; Costa, P. and Nakano, A.. A Comparison of A* and RRT* Algorithms with Dynamic and Real Time Constraint Scenarios for Mobile Robots. In Proceedings of the 9th International Conference on Simulation and Modeling Methodologies, Technologies and Applications - SIMULTECH; 2019, 398-405.

[10] Wang W, Pei D, Feng Z. The shortest path planning for mobile robots using improved A* algorithm[J]. Journal of Computer Applications, 2018, 38(5):1523-1526.