Proceedings of the 2023 International Conference on Machine Learning and Automation
DOI: 10.54254/2755-2721/34/20230284

Enhancing a star algorithm for robot path planning

Xuanming Zhang

University of Southampton, University Rd, Southampton SO17 1BJ, UK

xz6a23@soton.ac.uk

Abstract. This paper describes the importance of robot path planning in artificial intelligence and control theory, and proposes three improvements to the A-algorithm: bi-directional A-search, improved heuristic functions and pruning strategies. The performance of the different algorithms in terms of computation time, path length and number of nodes is compared through experiments. Moreover, it is emphasised in the article that in practical applications suitable algorithms and their improvements are selected according to the characteristics of the specific problem and reasonable evaluation criteria are used to measure the performance of the algorithms.

Keywords: robot path planning, A* Algorithm enhancement, heuristic optimization.

1. Introduction

Path planning plays an important role in robotics, which involves the solution of complex problems such as uncertainty, multiple robots and dynamics. In artificial intelligence and control theory, path planning involves decision-theoretic ideas (e.g., Markov decision processes, incomplete state information, learning methods, game-theoretic equilibria) and control-theoretic concepts (e.g., stability, feedback, and optimality problems) [1]. Path planning algorithms are a key component in robotics and are designed to solve the optimal path of a robot from a starting point to a goal point. Path planning algorithms have a wide range of applications in different fields and can be selected and customised according to the characteristics and needs of the problem [2]. There are several robot path planning algorithms and their comparison below:

A* Algorithm: The A* algorithm is a classical heuristic search algorithm that combines the features of breadth-first search and greedy search to find the shortest path quickly. It uses a heuristic evaluation function to evaluate the cost of a node and makes search decisions by combining cost and known information [3].

Dijkstra's Algorithm: Dijkstra's algorithm is a commonly used single source shortest path algorithm that gradually expands the search region centred on the starting point until the target node is found or the search is complete. Dijkstra's algorithm performs path selection by dynamically maintaining the shortest distance of the nodes [4].

RRT Algorithm: Rapidly-Exploring Random Trees (RRT) algorithm is a fast path planning algorithm for high dimensional spaces. It continuously explores the space and generates feasible paths through random sampling and tree structure building [5].

D* Algorithm: D* Algorithm is an incremental search algorithm that updates the path planning in real time with known changes in the cost information. D* Algorithm achieves dynamic path planning and replanning by introducing the factor of cost changes [6].

DOI: 10.54254/2755-2721/34/20230284

One of the common tasks in robot path planning is navigation in indoor environments. It involves autonomous navigation of robots in complex indoor environments such as offices, shopping malls, etc [7]. In indoor environment navigation, a common research approach is to use grid or metric maps to represent the state space and use relevant path planning algorithms for path search and planning [8].

The commonly used A* algorithms, although and their maturity, also have very high computational and storage requirements and can be greatly improved in terms of computational efficiency. In this paper, we explore the improvement of the A* algorithm by several improvement measures such as bidirectional A* search, the improvement of heuristic function, and the application of pruning strategy, and find out the improvement methods suitable for different situations.

Algorithm Evaluation Criteria are: In robot path planning, the merit of an algorithm can be evaluated by several criteria. Some common evaluation criteria include execution time, path length and number of search nodes. According to the specific application requirements, the appropriate evaluation criteria are selected to measure the performance of the algorithm.

The following sections of the paper unfold as such: The second section delves into the principles and procedural steps of the A* algorithm. In the third section, we present our improvement strategies for the A* algorithm. The fourth part explores the influence of our enhanced A* algorithm on path planning across diverse grid maps, and compares the effects of path planning. The final section encapsulates the optimization results of the A* algorithm, discusses the limitations of the current research, and proposes potential avenues for future investigation.

2. A* algorithm

2.1. Algorithm principle

The A* algorithm selects the next node to be expanded by taking two factors into account, firstly, the cost of the path that has been consumed (g(n)): i.e., the cost of the path from the starting node to the current node, which can be obtained by calculating the sum of the cost of the individual edges in the path. The second is the estimated cost of the heuristic evaluation function (h(n)): i.e., the estimated path cost from the current node to the target node, including Euclidean distance, Manhattan distance, etc.

The A* algorithm evaluates the total cost of each node during the search process by calculating f(n) = g(n) + h(n), where f(n) is the composite cost of node n. It selects the node with the smallest f(n) as the current node to achieve the most promising direction in the search space.

2.2. The steps to implement the A algorithm are as follows:*

- Start by assembling a set of nodes that will be subject to search, and establish the initial node.
- For each individual node, compute the cost of the path that has already been traversed, denoted as g(n), and the estimated cost from the heuristic evaluation function, h(n). Subsequently, add the node to the open list.
- Continuously execute the following steps until either the target node is identified or the open list is exhausted:
- Choose the node that features the smallest f(n) value from the open list to serve as the current node.
- If the current node is the target node, the search ends.
- Otherwise, mark the current node as visited and generate its neighbouring nodes.
- For the unvisited neighbouring nodes, calculate their already consumed path cost g(n) and the estimated cost h(n) of the heuristic evaluation function and add them to the open list.
- If the open list is empty but no target node is found, the path does not exist.

The A* algorithm may produce different search results depending on the heuristic evaluation function. If the heuristic evaluation function is accurate, the A* algorithm is able to find the shortest path. However, choosing an inappropriate heuristic evaluation function may cause the algorithm to fall into a suboptimal solution or the search becomes less efficient.

The A* algorithm is a commonly used heuristic search algorithm that finds the shortest path by combining known cost information and heuristic evaluation functions. Its core principle is to perform

DOI: 10.54254/2755-2721/34/20230284

node selection by calculating the integrated cost of nodes. A* algorithm has a wide range of applications in path planning and graph search problems, and can strike a balance between high search efficiency and accuracy.

3. Improvement of A* algorithm

3.1. Bidirectional a search

The Bidirectional A* search strategy is designed to simultaneously initiate searches from both the starting and ending points, aiming to decrease the spatial and time complexity of the search process. This approach utilizes two priority queues: one commencing from the initial node and the other from the destination node. During each iteration, the algorithm picks the node with the lowest projected cost from both priority queues for expansion. This cost is the sum of the actual accumulated cost from the path and the anticipated remaining cost as estimated by the heuristic function. When the search from both directions intersects at a single node, the algorithm has discovered a potential path. Nonetheless, this does not ensure that the shortest path has been identified, hence the algorithm proceeds with the search until the minimum projected cost of all other conceivable paths exceeds the length of the path currently identified. This strategy is ideally suited for scenarios with extensive problem space and where the target node's location is already known.

3.2. Improvement of heuristic function

In A* search, Heuristic Function (HF) is used to predict the minimum cost from the current node to the target node and thus determine the priority of the search. The choice of heuristic function has a great impact on the performance of the algorithm. Ideally, the heuristic function accurately predicts the minimum cost to the goal so that A* can find the shortest path immediately. In this paper, Manhattan distance has been used as the heuristic function in the maze pathfinding problem. The Manhattan distance is a good heuristic function for grid search problems that do not allow diagonal movement. However, when the robot can walk diagonally on the path i.e. can move diagonally, the Manhattan distance is no longer optimal. In this case, replacing the Manhattan distance by using Euclidean distance (straight line distance) or diagonal distance would be a better choice [9].

3.3. Pruning strategy

The pruning strategy is used in the A* algorithm to reduce unnecessary node expansion during the search process in order to improve the efficiency of the algorithm [10]. By checking whether the total cost of a node (the sum of actual and estimated costs) exceeds the currently known shortest path length, the pruning strategy identifies and skips those ineffective nodes, thus avoiding further expansion of these nodes.

The pruning strategy is introduced to cope with the problem of a large search space and a large number of invalid nodes. By pruning, the algorithm can effectively narrow down the search scope and reduce unnecessary computational and storage overheads. The effectiveness of the pruning strategy depends on several factors, including the size of the problem, the accuracy of the heuristic function, the characteristics of the problem structure, and the specific implementation of the pruning strategy.

The use of the pruning strategy may cause the algorithm to fail to find an optimal solution. The pruning operation may discard some potentially shorter paths, which may cause the algorithm to find a sub-optimal solution. However, in large-scale problems, time efficiency is usually an important consideration, so in practice, speeding up the execution of the algorithm by reducing the search space is considered an acceptable compromise.

And in this paper, the pruning strategy is mainly applied in the following improvements:

*3.3.1. L*imiting the search depth. Since the maze problem is large-scale, the A* search algorithm may consume a large amount of resources. In this case, set a maximum search depth or a maximum number of iterations. If this limit is reached, the search is stopped and an error message is returned. This is a

DOI: 10.54254/2755-2721/34/20230284

protective pruning strategy that not only prevents the algorithm from running indefinitely without a solution, but also greatly improves the search efficiency [11].

3.3.2. Skip-point search. This approach can be very effective. Skip-point search tries to skip many unnecessary nodes and focuses only on the nodes that may lead to a change in the path, thus greatly reducing the number of nodes that need to be searched. However, implementing skip-point search requires a large modification to your code, and in some cases, skip-point search may not improve efficiency instead of a large number of redundant paths.

4. Experiments

In order to verify the described algorithm, a random maze generated by python was used, and 3 different mazes were used to perform 200 calculations using different algorithms to take the average of the calculation time, path length, and number of nodes, and in this way to compare and verify whether the improved algorithm of A* is effective.

4.1. Generation of maze map

The maze generation procedure based on depth-first search using python code represents the maze as a two-dimensional array, with 1 representing the wall and 0 representing the passage. It's shown on figure 1**.** The main methods in it are:

- init: Initialise the maze, setting all positions as walls and every other cell position as a channel, as well as initialising the access set and search stack.
- create_maze: generates the maze using depth-first search, choosing unvisited directions to chisel through the walls and create new channels.
- print_maze and draw_maze: Used to output the state of the maze, the former prints out the maze, the latter displays it as an image.
- save_maze_to_file and load_maze_from_file: Save and load maze state.

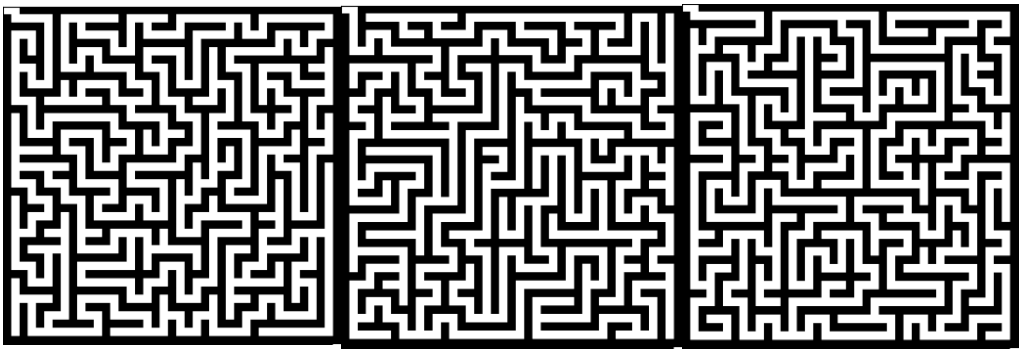

Figure 1. Three different maps randomly generated.

4.2. Computational parameters of A and A*-improved algorithms*

The average running time, average path length, and average number of check nodes can be obtained after performing the pair maze solving operation using the four A* and A* improved algorithms.

Table 1. A* and A* improved algorithms for solving maze related parameters.

Arithmetic	Computational time (ms)	Length of path (cells)	Examined cells (cells)
A*	261	1460	1277
Bidirectional A search	439	1412	1145

DOI: 10.54254/2755-2721/34/20230284

Table 1. (continued).

Improvement of heuristic function	1035	1048	1298
Pruning strategy	42	1672	66

It can be seen from Table 1 that the improvement of two-way A* search can search fewer nodes and can result in relatively shorter paths, but it takes a little more time; the improvement of heuristic function can find almost the shortest paths, but it will take the longest time; and the pruning strategy can use the least amount of time and search the least number of nodes, but the paths obtained are imprecise, and they are the longest paths among these.

Table 2 is a summary of the advantages and disadvantages of the various algorithms.

Table 2. Advantages and disadvantages of the various algorithms.

Arithmetic	advantages	drawbacks
A*	A* algorithms tend to be more efficient than violent search algorithms and are guaranteed to find a solution when the problem space is discrete and finite.	High memory requirements, need to store all nodes that have been visited, more dependent on the quality of the heuristic function.
Bidirectional A search	Compared to unidirectional A* search, it can significantly reduce the search space and improve the search efficiency.	The implementation is relatively more complex and needs to ensure consistency between the two sides of the search. Explicit information about the target node is required.
Improvement of heuristic function	It can significantly improve the efficiency of the A* algorithm and reduce the search space.	It may increase the time complexity of the algorithm.
Pruning strategy	Effectively reduces the search space, thereby improving search efficiency and reducing the time and space complexity of the computation.	The design and implementation is more complex and it may not be possible to find an optimal solution.

5. Conclusion

The A* search algorithm is an effective path finding algorithm that can improve search efficiency while ensuring that the shortest path is found by using heuristic functions to guide the search direction. However, for large-scale or complex problems, the A* algorithm may face the problems of large memory requirements and search efficiency being greatly affected by the heuristic function.

Bidirectional A* search is an improved way of A algorithm, which starts searching from the start point and the end point at the same time, and the path is found when the two search directions meet. This approach can significantly reduce the search space and thus improve the search efficiency. However, the implementation of bi-directional A* search is relatively complex and requires explicit information about the target node. The heuristic function is the core of the A* algorithm, which determines the search direction and search efficiency. The efficiency of the A algorithm can be further improved by improving and optimising the heuristic function. However, good heuristic functions often need to depend on the properties of the specific problem, require significant domain knowledge, and may increase the time complexity of the algorithm. The pruning strategy is another way to optimise the

DOI: 10.54254/2755-2721/34/20230284

A* search algorithm, which can effectively reduce the search space by deleting some search directions that are unlikely to produce an optimal solution, thus improving the search efficiency. However, the design and implementation of the pruning strategy can be relatively complex and may result in failure to find the optimal solution if the pruning strategy is not properly designed.

Overall, improvement of the A* algorithm usually involves optimising the algorithm from different aspects (e.g., search strategy, heuristic function, pruning strategy, etc.) to suit different problem scenarios and requirements. In practice, we may need to select or design appropriate versions of the algorithm and optimisation strategies according to the characteristics and needs of specific problems.

References

[1] LaValle, S. M. (2006). Planning Algorithms. Cambridge University Press, New York, USA.
[2] Van Den Berg, J., Abbeel, P., & Goldberg, K. (2011). LQG-MP: Optimized path planning for robots with motion uncertainty and imperfect state information. The International Journal of Robotics Research, 30(7), 895-913.
[3] Russell, S., & Norvig, P. (2016). Artificial Intelligence: A Modern Approach. Malaysia; Pearson Education Limited.
[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT press.
[5] Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research, 30(7), 846-894.
[6] Nash, A., Daniel, K., Koenig, S., & Felner, A. (2007). Theta*: Any-angle path planning on grids. In Proceedings of the National Conference on Artificial Intelligence, 1177-1183.
[7] Kostavelis, I., & Gasteratos, A. (2015). Semantic mapping for mobile robotics tasks: A survey. Robotics and Autonomous Systems, 66, 86-103.
[8] Thrun, S. (2002). Robotic mapping: A survey. In Exploring artificial intelligence in the new millennium (pp. 1-35).
[9] Ge, X., & Liu, Y. (2002). Ant colony optimization technique for robot path planning with a fuzzy logic controller. In Proceedings. 2002 IEEE International Symposium on Computational Intelligence in Robotics and Automation (pp. 315-320). IEEE.
[10] Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. Journal of Game Development, 1(1), 7-28.
[11] Stern, R., Puzis, R., & Felner, A. (2011). Potential-based bounded-cost search. In Proceedings of the Twenty-Second international joint conference on Artificial Intelligence (pp. 823-829).