A Comprehensive Investigation of Algorithms for Solving Mazes

Tao Zhu

School of Management Engineering, Nanjing University of Information Science and Technology, Nanjing, China

202383780040@nuist.edu.cn

Abstract. The maze problem is a classic research topic in artificial intelligence, algorithm design, and path planning. Its core lies in exploring how to find an efficient path from the starting point to the target location within a maze, possessing both theoretical research value and practical application significance. This article provides a comprehensive review of multiple maze-solving algorithms. These include wall-following algorithm, backtracking algorithm, M-DFS, A* algorithm, as well as Maze-solving robot and computer mouse. This paper first provides detailed explanations of the principles for each method. Subsequently, this paper uses tables to clearly compare the Application Scenarios, Time and Space Complexity, and Disadvantages of each approach. Furthermore, this paper analyzes the shortcomings of each method and discuss the future development prospects of maze-solving techniques in light of current technological advancements. Overall, this paper offers readers in-depth explanations of several algorithms, enabling them to better understand the current state of the maze field and its future outlook.

Keywords: Maze-solving algorithm; wall-following; backtracking.

1. Introduction

A maze, as a complex spatial structure composed of passages and obstacles, is characterized by the uncertainty of path exploration between the starting point and the endpoint. From traditional physical maze games to modern technological fields, the application scenarios of maze models have expanded across multiple dimensions: in the field of robotics, maze environments are often used to test the path planning capabilities of autonomous navigation systems; in network communications, the optimization of data transmission paths can be abstracted into a maze-solving model. The generation and solving of mazes constitute the two core research topics in this field. Generation algorithms determine the complexity of the maze's topological structure, while solving algorithms focus on efficiently finding feasible or optimal paths from the starting point to the endpoint.

Maze solving, as an important branch of intelligent algorithm research, holds irreplaceable application value in fields such as automation control and artificial intelligence. The Wall-Following Algorithm proposed in the 1950s achieved path exploration by maintaining constant contact with one side of a wall [1]; The Backtracking Algorithm, which emerged in the 1960s, utilized recursive thinking to record explored paths and, upon encountering a dead end, backtracked to the branching point to re-select a direction [2], becoming the first systematic method capable of handling complex mazes. Building on an analysis of the limitations of traditional solutions, researchers proposed the improved depth-first search algorithm M-DFS. This algorithm uses directed graphs to store mazes, reducing spatial complexity, and employs an improved depth-first search strategy to seek feasible paths [3]. With the development of computer technology, the A* algorithm introduced a cost evaluation function, using pathfinding and graph traversal to find the shortest path between the source and target points in a maze, and is widely applied in game AI and robot navigation fields [4]. With the innovative development of automation technology, researchers developed autonomous maze-solving robots with self-mapping and localization capabilities. This robot uses three infrared sensors to detect walls and obstacles to avoid collisions and employs a robotic arm to pick up and place objects, thereby clearing path obstacles [5]. In addition, a method involving the allocation and regulation of artificial potential fields for robot

autonomous path decision-making has been proposed. This method maintains the integrity of the potential field while providing local optimal path selection to achieve the target destination [6]. Researchers also proposed a maze search algorithm that combines the centripetal rule with the flood propagation method and incorporates pre-propagation functionality. This algorithm enables the computer mouse to expand the wall information it has obtained during exploration and uses the flood propagation method to eliminate branches in the paths selected by the centripetal rule that led to unreachable endpoints, thereby improving maze search efficiency [7].

This paper conducts a systematic study on maze-solving algorithms: first, it provides a detailed explanation of the core principles and implementation mechanisms of wall-following algorithms, backtracking algorithms, M-DFS algorithms, A* algorithms, maze-solving robots, and computer mouse-related algorithms; second it examines the applicable scenarios and the performance characteristics of each algorithm, summarizes the shortcomings of existing algorithms, and explores the development direction of emerging technologies; finally, it summarizes the research content of the entire paper to provide reference for the practical application and further research of maze-solving algorithms.

2. Methods

2.1. Wall-following Algorithm

Wall-following algorithms establish a fixed "following direction" during maze exploration to maintain contact with a single wall, leveraging the continuity of maze boundaries to constrain path decisions. By ensuring "boundary connectivity" and "directional consistency," they guarantee complete traversal from entrance to exit [8].

2.1.1 Maze Boundary Connectivity

In simple mazes without isolated loops, the entrance and exit boundaries reside within the same connected region. Starting from any wall adjacent to an entrance, continuous movement along that wall leads to the corresponding exit wall.

2.1.2 Direction Consistency and Relative Position Judgment

The algorithm maintains stable wall contact through "direction consistency," based on "current azimuth + environment detection of the follow side (left/right)": Set a fixed follow side; Prioritize movement along the "direction relative to the follow side based on the current azimuth."If no wall exists on the follow side, move to that side and update the direction .If a wall exists on the follow side, attempt to "move straight ahead." If blocked, turn toward the "opposite direction of the follow side" until a passable path is found.

This priority sequence— "follow side first → straight ahead second → turn last"—ensures continuous contact with the target wall and prevents disorderly turning.

2.2. Backtracking Algorithm

Starting from the maze entrance, the backtracking algorithm prioritizes deep exploration in a single direction. If no viable, unexplored path exists at the current position, it reverses the last decision, returns to the previous location, and attempts another direction until the exit is found or all paths are exhausted. This algorithm is based on two core principles: "depth-first traversal" and "state backtracking reset [9].

2.2.1 Depth-First Path Traversal

The maze employs a depth-first strategy: starting from the entrance, it selects one unexplored viable direction at a time to advance to a new position, repeating exploration until no further progress is possible.

2.2.2 Path Marking and Backtracking Reset

Visit Marking: An “access matrix” (matched to the maze's dimensions, where 0 = unvisited, 1 = visited) tracks explored positions. Upon reaching a location, its marking is set to 1 to prevent revisits. If no viable unexplored paths exist at the current position, its visit marking is reset to 0, removed from the path sequence, and the algorithm returns to a previous state to select a new direction.

This “mark → explore → backtrack → reset” cycle enables the algorithm to attempt alternative solutions after encountering invalid paths.

2.3. M-DFS Algorithm

The core concept of the M-DFS algorithm is to establish a priority order for exploration directions based on depth-first search, such as the common sequence left, down, up, right. When the algorithm encounters a path that cannot reach the exit during exploration, and all surrounding directions have been attempted or are infeasible, M-DFS -DFS further evaluates the selection of backtracking points. It assesses the length of paths currently in the stack and the potential cost of backtracking. If the stack contains too many path nodes, M-DFS tends to pop nodes closer to the starting point, skipping portions of invalid paths to restart exploration. This accelerates the search process and prevents excessive computational resource waste on dead-end paths [3].

2.4. A* Algorithm

The core concept of the A* algorithm centers on the precise design of the evaluation function f(n), which serves as the primary basis for prioritizing decision nodes in the algorithm. By integrating “cost incurred” with “predicted future cost,” it enables a comprehensive assessment of node value, expressed as $f(n) = g(n) + h(n)$. Here, g(n) represents the actual path cost from the starting node to the current node n. This value is typically the cumulative sum of edge weights or distances. h(n) represents the estimated heuristic cost from the current node n to the target node. This prediction of future costs directly impacts the algorithm's efficiency and performance.

The A* algorithm operates within a graph search framework, maintaining orderly node management through two key lists: the Open List stores discovered but unexplored nodes as candidates for subsequent exploration, while the Closed List stores nodes that have been fully explored.

The algorithm operates by abstracting each possible position or state in the search problem as a node in the graph. Connections between nodes correspond to feasible moves in the real-world scenario. Edges may carry weights to quantify the cost of moving between nodes, providing data support for calculating g(n). The heuristic function h(n) must ensure that the estimated cost does not exceed the actual cost to guarantee optimal path finding. Common heuristic functions include Manhattan distance, Euclidean distance, and Chebyshev distance. The specific choice depends on the movement rules of the actual search scenario.

The A* algorithm typically implements the open list using a priority queue. Nodes are sorted based on their f(n) values within the priority queue. Nodes with smaller f(n) values indicate lower estimated total costs from the start to the goal, thus receiving higher exploration priority. Each iteration retrieves the node with the smallest f(n) from the queue for expansion. This involves analyzing its adjacent nodes, calculating g(n), h(n), and f(n) for these neighbors, and then adding the node to either the open list or closed list based on its state [10].

2.5. Maze-solving Robot

The robot employs a combined strategy of “two runs + left-hand wall-following method + path optimization” to solve the maze.

2.5.1 First Run: Maze Exploration and Decision Logging

Employing an improved "Left-Hand Wall-Following Method," this approach assumes the "no-loop" property of the maze. Its core principle involves the robot using infrared sensors to detect intersections within the maze, prioritizing the leftmost viable direction, and recording decisions at each intersection. The corresponding codes are: "L = Left Turn, S = Straight Ahead, R = Right Turn, B = Turn Around." The maze exit is marked by a black square. When all 8 infrared sensors detect the black square, the robot determines it has reached the exit. At this point, the development board's green LED flashes, awaiting instructions for the second run.

2.5.2 Second Run: Optimized Path Solving

Based on the "intersection decision string" recorded during the first run, predefined logic simplifies the decision sequence. Core optimization formulas include: LBR=B (Left turn → U-turn → Right turn is equivalent to U-turn), RBL=B (Right turn → U-turn → Left turn is equivalent to U-turn), SBS=B (Straight → U-turn → Straight is equivalent to U-turn), LBL=S (Left turn → U-turn → Left turn is equivalent to Straight), SBL=R (Straight → U-turn → Left turn is equivalent to Right), LBS=R (Left turn → U-turn → Straight is equivalent to Right). The length of the optimized decision string matches the number of maze intersections. Starting from the initial position of the first run, the robot follows the optimized decisions directly along the shortest path to reach the exit.

2.5.3 Path Optimization Principle: Ineffective Decision Pruning

Through logical equivalence substitutions, redundant back-and-forth movements are eliminated, retaining only core decisions meaningful to the "shortest path." This enables the robot to bypass ineffective path segments during the second run, moving directly along the optimal route to enhance solution efficiency [5].

2.6. Computer Mouse Algorithm

The core of the Computer Mouse Maze Solving Algorithm lies in the integration of the "Centripetal Rule and Flood Search Method" combined with "Search Information Expansion."

2.6.1 Core Algorithm Integration Logic

The algorithm uses the "Centripetal Rule" as the foundational path selection criterion, combined with the "Flood Search Method" to preemptively eliminate invalid paths, forming a closed-loop logic of "Selection-Deduction-Decision": First, the 16×16 grid maze is divided into 4 equal regions. Then, based on the mouse's current location, it determines the priority direction toward the exit. Path selection follows the rule: "Prioritize straight movement → Secondary option: turn 90° → Prioritize turning toward the exit direction," ensuring the initial path consistently converges toward the goal. When facing branch choices, the algorithm uses acquired wall information to determine via "flood simulation" whether the selected branch can reach the goal. The simulation mimics floodwater flow, treating branch origins as "water sources." Based on wall information, it determines if the 'water' can reach the endpoint: if reachable, it explores that branch; if unreachable, it marks the branch and all paths covered by the simulation as "dead ends," excluding them from future searches to avoid futile mechanical movement.

2.6.2 Search Information Expansion Principle

Two 16×16 two-dimensional arrays are constructed: `MapBlock` stores actual detected wall information, while `LogBlock` stores both actual data and simulated expansion information. During initialization, `LogBlock` first marks the maze's perimeter as "no path," while interior cells are temporarily set as "path."

Each time the computer mouse explores a cell, it simultaneously updates the wall information of its surrounding four cells in `LogBlock`. This achieves "explore one cell, expand multiple cells," providing more effective data for subsequent path determination [7].

3. Analysis and Discussion

3.1. Algorithm Comparison

Table 1 provides the application scenarios, time and space complexity, and disadvantages of algorithms.

Table 1. Application scenarios, time and space complexity, disadvantages of algorithms

Algorithm	Application Scenarios	Time and Space Complexity	Disadvantages
Wall-following algorithm	Micro-robots with limited hardware resources, low-cost exploration devices, real-time navigation that does not require the optimal path	O(N) (N is the total number of passable cells; each cell is visited at most twice); O (1)	Strong scenario limitations; Non-optimal path; Relies on boundary integrity
Backtracking algorithm	scenarios requiring traversal of all potential paths (e.g. verifying multiple paths in a maze), accurate solving of small-scale mazes	O(4^N) (Worst case, all four directions of each cell need to be explored); O(N)	Extremely low efficiency; Recursion is prone to stack overflow; No path optimization
M-DFS algorithm	Scenarios balancing efficiency and complexity, exploration tasks requiring higher efficiency than traditional backtracking	O(N)~O(kN) (k is the repeated exploration coefficient caused by closed loops; O(N)	Non-optimal path; Sensitivity to closed loops; Dependence on priority
A* algorithm	Scenarios requiring the optimal path (e.g. AGV navigation, game AI, indoor positioning), high-performance devices	O(NlogN) (Ideal case, when the heuristic function is efficient, priority queue operations dominate); O(N) (the maximum size of both is N)	Complex implementation; High resource consumption; Dependence on heuristics
Maze-solving robot	Educational experiments, small-scale line-following maze competitions, the optimal path for the second run	Exploration Phase:O(N) (N is the total length of passable paths, each cell is visited at most once); 2. Optimization Phase:O(M) (M is the number of crossroads); O(M) (M is the number of crossroads)	1. Only applicable to "loop-free + line-guided" mazes; 2. The second run must start from the initial position, resulting in poor flexibility; 3. Difficult to adapt to scenarios with multiple exits or dynamic obstacles
Computer mouse algorithm	Micromouse competitions, requiring minimization of "search time × 1/30 + sprint time" and balancing efficiency with optimal path exploration	1.Search Phase:O(NlogN)(ideal loop-free scenario) ~O(kN)(k is the loop coefficient); 2. Sprint Phase:O(N) (N is the total number of cells); O(N)	1. Flood deduction requires large computational load; 2. Prone to misjudging valid paths in mazes with numerous complex loops; 3. High code complexity; 4. Weak fault tolerance

3.2. Challenges

Maze-solving algorithms face significant limitations. Wall-following algorithms may get stuck in looping paths within complex mazes, resulting in low efficiency. The efficiency of backtracking

depends on the selection list generation method and pruning strategy; poor strategies can lead to extensive futile searches. DFS algorithms are constrained by system stack depth limits, often encountering stack overflow issues when processing large-scale graphs. A* algorithms require maintaining open and closed lists, storing substantial node information, and frequently face memory bottlenecks when handling large-scale path optimization problems. Furthermore, improper heuristic function design can reduce algorithm efficiency or even prevent finding optimal paths.

Most algorithms adapt only to specific maze configurations, experiencing steep performance drops when applied to new layouts or dynamic environments, lacking cross-scenario universality.

3.3. Future Prospects

AI Technology Integration Optimization: Deep reinforcement learning (e.g., DQN variants) handles high-dimensional state spaces; meta-learning accelerates algorithm convergence and generalization.

Hardware Upgrades Overcome Limitations: FPGA and GPU implementations enable parallel computing, enhancing algorithm real-time performance; quantum computing holds promise to break traditional complexity bottlenecks, efficiently solving ultra-large-scale maze problems;

Multi-Technology Integration Expands Applications: Integration with SLAM and visual navigation enhances path planning capabilities in real-world environments; multi-agent collaboration improves solution efficiency and robustness, with future applications extending to VR/AR, intelligent transportation, and biological network analysis.

4. Conclusion

This paper provides a comprehensive review of various maze-solving algorithms. These include wall-following algorithm, backtracking algorithm, M-DFS, A* algorithm, and Maze-solving robot and computer mice. Subsequently, this paper compares these methods across three dimensions: application scenarios, time and space complexity, and disadvantages. Finally, by analyzing the shortcomings of each approach and considering current technological advancements, this paper outlines the future development prospects for maze-solving algorithms.

References

[1] Saman ABS, Abdramane I. Solving a reconfigurable maze using hybrid wall follower algorithm. 2013.

[2] Prasetyo TA, Chandra R, Simamora B, et al. Pathfinding solving in maze game using backtracking algorithms. Jurnal CoreIT. 2023;9(1):2599-3321.

[3] Liu XF. M-DFS: A DFS algorithm for solving maze problems. Journal of Tonghua Normal University. 2008;(02).

[4] Barnouti NH, Al-Dabbagh SSM, Naser MAS. Pathfinding in strategy games and maze solving using A* search algorithm. J Comput Commun. 2016;4(11).

[5] Kumar R, Jitoko P, Kumar S, et al. Maze solving robot with automated obstacle avoidance. Procedia Comput Sci. 2017; 105:57-61.

[6] Wyard-Scott L, Meng Q-HM. A potential maze solving algorithm for a micromouse robot. In: IEEE; 1995.

[7] Guo CS, Gong T, Li L. A maze search algorithm for micromouse. Journal of Huazhong University of Science and Technology (Natural Science Edition). 2013;(S1).

[8] Niemczyk R, Zawiślak S. Review of maze solving algorithms for 2D maze and their visualisation. In: International Conference of Students, PhD Students and Young Scientists "Engineer of the XXI Century". Cham: Springer International Publishing; 2018.

[9] Wang A. OCaml-based recursive backtracking and Aldous Broder algorithms in solving maze challenges. In: AIP Conference Proceedings. Vol. 3194. No. 1. AIP Publishing LLC; 2024.

[10] Chan KH, Mustapha A, Lee SC. Shortest pathfinding in a standard rectangular maze using A* search algorithm. Int J Integral Eng. 2024;16(3):244-56.