Scaling Analysis of Maze-Solving Algorithms: From BFS And Dijkstra to Heuristic A*

Kewen Zhu

Nanjing Foreign Language School, Nanjing, China

wilishereo.o@gmail.com

Abstract. The maze solving has long been regarded as a fundamental problem in computer science and artificial intelligence. By modeling a maze as a weighted graph through the skeletonization, the problem can be approached as a path-finding task. This paper presents an empirical study that evaluates the classical search algorithms including BFS, DFS, Dijkstra alongside heuristic search methods (standard A* with Manhattan distance and a turn-penalized A* variant). Using mazes of varying and averaging results across multiple random seeds, the experiments measure performance in terms of expanded nodes, and runtime. Furthermore, log–log regression is applied to characterize the empirical scaling behavior of each algorithm. The results show that all algorithms exhibit near-linear growth in both node expansions and runtime, but differ significantly in constant factors. The turn-penalized A* demonstrates the influence of the heuristic design on search cost and path smoothness, highlighting trade-offs between the human-inspired intuition and the algorithmic optimality.

Keywords: Maze solving; heuristic function; turn penalty; algorithm evaluation.

1. Introduction

Maze solving is a central problem in computer science, and has been investigated for a long time. Applying skeletonization techniques [1], a maze can be viewed as a weighted graph with nodes representing the junctions, edges connecting the nodes representing the one-way paths connecting the junctions, and weight of each edges representing the distance between two junctions. Solving a maze then becomes a path-finding problem: determining a route from a given start point to a goal [2].

A wide range of classical algorithms has been applied to this problem. There are two simple categories, which are blind search and heuristic search [3]. A blind search is often referred to as an uninformed search because it operates without prior knowledge of the problem domain. Its sole ability lies in recognizing whether a state is a goal state or not. Some classic examples of blind search can be found. Breadth-first Search (BFS)guarantees finding the shortest path in unweighted graphs; however, its time cost grows rapidly as the maze size increases [4]. In contrast, Depth-First Search (DFS) may expand fewer nodes and thus require less time in certain cases, but it cannot ensure an optimal path [2]. Dijkstra's algorithm provides guaranteed optimal solutions in weighted settings but at high cost in time and memory [5]. To improve efficiency, Heuristic search have been considered. Heuristics, commonly the Manhattan or Euclidean distance, are practical strategies or guidelines—often described as rules of thumb—that can assist in solving a problem, although they do not always ensure a definite solution. They represent domain-specific knowledge that can steer the search process more efficiently toward the desired goal. A* is one of the most widely used algorithms in the heuristic search family. These approaches represent the standard toolkit for computational path-finding, and their theoretical properties are well understood.

This paper conducts a systematic evaluation of classical and heuristic-guided maze-solving algorithms within a unified experimental framework. The baselines include BFS, DFS, Dijkstra's algorithm, and the standard A* with Manhattan distance as its heuristic [6]. In addition, a human-inspired variant of A* is implemented, where the cost function incorporates a small penalty for turns in order to mimic human navigation preferences. The evaluation is carried out across multiple maze families and sizes, and metrics such as solved rate, path length, number of turns,

expanded nodes, and runtime are recorded and averaged over randomly generated instances. The study aims to clarify under what structural conditions heuristic-guided A* variants perform competitively, and what the results reveal about the trade-offs between intuitive heuristics and algorithmic optimality. By combining skeletonization-based analysis with large-scale experiments, the work provides insight into the scaling behavior of search algorithms and highlights the contexts in which simple heuristics remain surprisingly effective.

2. Method

2.1. Design of a maze

This paper focuses on the performance of each algorithm in two-dimensional maze generation.

To better display a maze, this paper applies the idea of cell decomposition [1]. Based on the idea of cell decomposition, this study introduces a convenient maze data structure. In this approach, the maze is represented as a collection of cells, with each cell having four potential edges. Unlike traditional cell-based maze representations, this method does not explicitly store spaces occupied by obstacles. Instead, it uses cells as the fundamental unit: a two-dimensional array encodes the spatial arrangement of cells in the maze, while an integer at each position records the presence or absence of "walls" around the corresponding cell (i.e., whether its edges permit traversal).

Formally, each cell is encoded as an unsigned 8-bit integer acting as a bitmask. Four bits are reserved to represent the four directions (north, south, west, east). A bit value of 1 denotes the existence of a wall, while 0 denotes an open passage. For example, the value 1111_2 (decimal 15) indicates a cell completely enclosed by walls, whereas 1011_2 (decimal 11) corresponds to a cell whose east edge is open but all other sides remain blocked. By using this compact representation, maze operations such as carving passages or testing connectivity reduce to efficient bitwise operations. Table 1 provides a clear illustration of the encoding scheme used to represent maze cell boundaries, while Fig. 1 presents a 3×3 maze example together with its corresponding computer representation.

Table 1. Encoding scheme of maze cell edges using 8-bit representation

Direction	Bit position	Bit value	Meaning when bit = 1
North	0	0b0001 (1)	Wall exists on the north edge
East	1	0b0010 (2)	Wall exists on the east edge
South	2	0b0100 (4)	Wall exists on the south edge
West	3	0b1000 (8)	Wall exists on the west edge

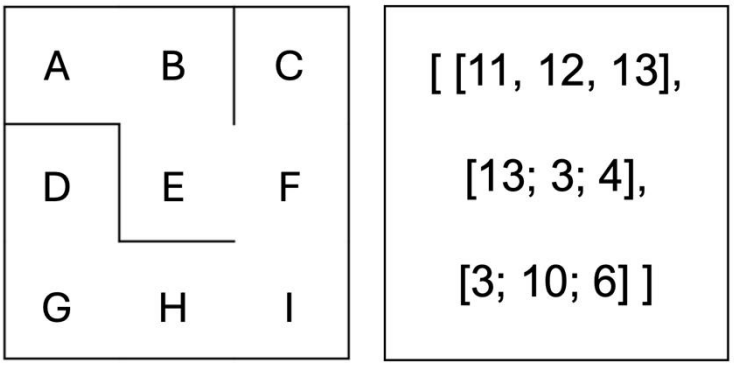

Fig. 1 A 3×3 maze example (left) and its representation in computer encoding (right)
(Picture credit: Original)

For better algorithmic analysis, it is much easier to analyze a maze from the perspective of skeletonization [1], or graph abstraction techniques. In this view, the maze is reduced to its topological skeleton: cells are classified into junctions (i.e., branching or turning points) and

corridors without branching. Junctions are treated as nodes in a graph, while straight corridors between them are converted into edges. By counting the number of cells (or steps) along each corridor, the weight of each edge corresponds to the number of constituent cells along the corridor.

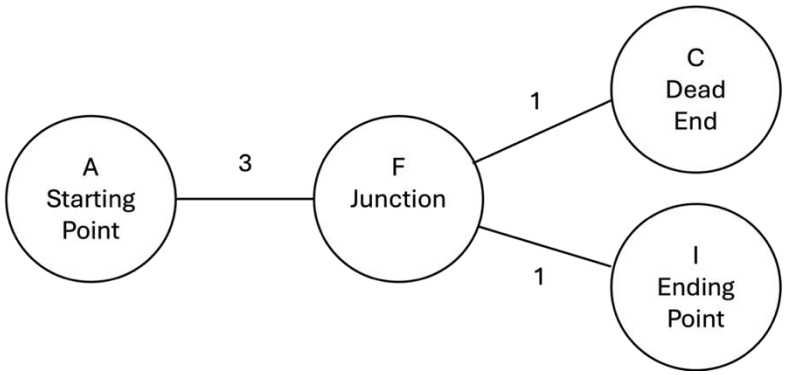

Fig. 2 Skeletonization of the example 3×3 maze (Picture credit: Original)

Fig. 2 illustrates the skeletonization of the example 3×3 maze. Only four kinds of nodes are preserved: the start cell (A), a junction (F) (in other maze may have plenty), a dead-end (C)) (in other maze may have plenty), and the goal (I). Corridors consisting of degree-2 cells are collapsed into weighted edges. For example, the corridor from A to F passes through three intermediate cells, resulting in an edge weight of 3. Importantly, cells such as D, G, and H never appear as nodes in the skeletonized graph.

Formally, let $G = (V, E)$ be the skeleton graph, where $V = \{s, t\} \cup \{v \mid deg(v) \neq 2\}$. Each maximal chain of degree-2 cells between $u, v \in V$ is contracted into a single edge (u, v), with weight equal to the number of cells in that chain.

2.2. DFS

Depth-First Search (DFS) explores as far as possible along one path before backtracking. Starting from the source node, the algorithm immediately proceeds to one unvisited neighbor and continues recursively (or with an explicit stack, in order to decrease space complexity) until it reaches a dead end or a node whose neighbors have all been visited. It then backtracks to the most recent branching point and explores the remaining unexplored branches. DFS does not guarantee shortest paths.

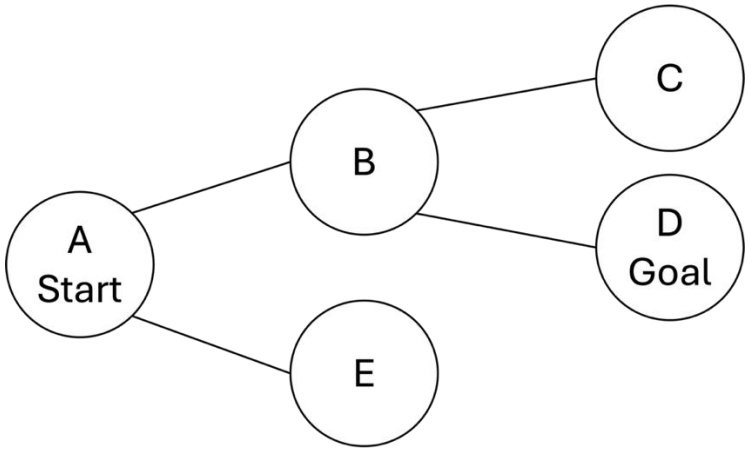

Fig. 3 Illustration of how algorithm work (Picture credit: Original)

According to Fig. 3, BFS will check nodes in the order of A-B-C-D.

2.3. BFS

Breadth-First Search (BFS) explores the graph in a level-by-level manner. BFS uses a FIFO queue to do this. Specifically, when visiting a node, the algorithm enqueues all of its adjacent nodes and then processes them in order until the ending nodes is found.

BFS ensures that all nodes at the current depth are explored before moving to the next depth level. As a result, BFS naturally computes shortest paths from the source to all other nodes in an unweighted graph.

According to Fig. 3, BFS will check nodes in the order of A-B-E-C-D.

2.4. Dijkstra

Dijkstra's algorithm can be regarded as an extension of BFS. While BFS naturally computes shortest paths in unweighted graphs, Dijkstra's algorithm generalizes this idea to weighted graphs with non-negative edge costs.

The key mechanism is the use of a priority queue to always expand the node with the smallest tentative distance.

In this way, the algorithm guarantees the discovery of the shortest path from the source to all other nodes, including the optimal path between a specific pair of nodes. It should be noted, however, that the algorithm is not applicable to graphs with negative edge weights. For maze problems, where edge weights are inherently non-negative, Dijkstra's algorithm is sufficient and effective.

2.5. A*

The A* algorithm is derived from Dijkstra's algorithm. Its key modification lies in the priority queue logic: instead of prioritizing nodes solely by their current cost $g(n)$ A* uses

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

where $g(n)$ denotes the actual cost from the start to node n, and $h(n)$ denotes a heuristic estimate of the remaining cost from n to the goal [7]. As a result, the design of the heuristic function h is critical [8]. In most cases, a well-chosen heuristic effectively guides the search and significantly reduces computation time. However, this is not guaranteed, and in some cases a poor heuristic may even cause A* to expand more nodes than Dijkstra's algorithm.

This paper evaluates two variants of the A* search algorithm with different heuristics:

1. Classical A*: The heuristic is the Manhattan distance,

$$h(n) = |n_x - goal_x| + |n_x - goal_y| \tag{2}$$

which aligns with the 4-connected grid structure and effectively guides the search in mazes. The following paper refers to this method as A* for short.

2. Human-inspired A*: In addition to the standard cost function, a turn penalty is introduced. The state space is augmented with orientation, and each turning action incurs a small additional cost. This modification reflects the human tendency to prefer straight paths and avoid frequent turns, thereby producing more "natural" trajectories in certain maze configurations. The following paper refers to this method as A*_turn for short.

2.6. Experimental design

The experimental procedure is as follows:

1. Maze generation: For each specified size, a two-dimensional array is initialized with every cell fully walled. The top-left cell is set as the start and the bottom-right cell as the goal. A randomized depth-first backtracking algorithm is first applied to guarantee at least one solvable path. Then, additional passages are carved from previously untouched cells to enrich the maze's complexity and structural variety [9].

2. Algorithm execution and evaluation metrics: Each candidate algorithm is executed on the generated mazes, and the following metrics are recorded:

expanded: the number of nodes expanded during search,

time_sec: runtime in seconds.

3. Maze sizes and averaging: Experiments are conducted on seven maze sizes: 100*100,140*140,200*200,280*280,400*400,560*560,800*800. For each size, five random mazes are generated. The final reported result is the average across the five runs.

4. Data processing: To characterize the empirical complexity, this paper performed log–log regression of the form

$$logY = \alpha \cdot logV + \beta \tag{3}$$

where V is the number of vertices in the maze graph and Y is either the number of expanded nodes or the runtime. The slope α corresponds to the growth rate and indicates an empirical complexity of approximately $O(V^{\alpha})$. The intercept β corresponds to the multiplicative constant $k = e^{\beta}$, which is less relevant for asymptotic analysis but still provides insight into constant factors in implementation. In practice, values of α close to 1 suggest linear growth, consistent with $O(V)$; values significantly above 1 (e.g., 1.2–1.3) are consistent with $O(VlogV)$. Extremely small or large slopes are usually not meaningful, as they may reflect noise or insufficiently large problem sizes rather than the true asymptotic behavior.

3. Result and Discussion

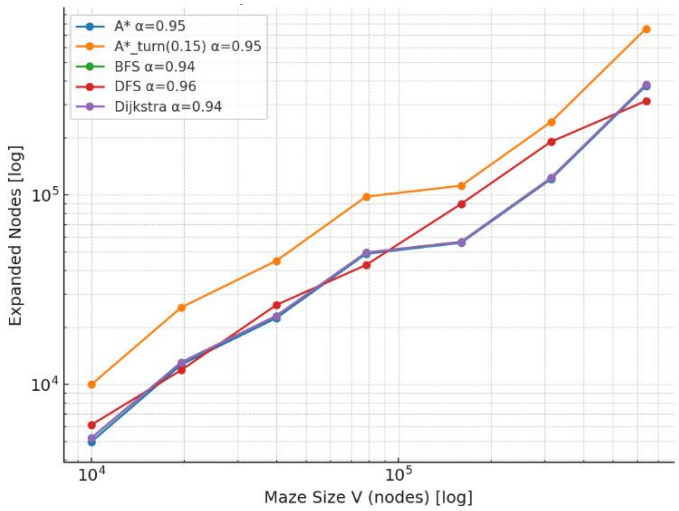

Fig. 4 Log–log plot of the number of expanded nodes versus maze size for different algorithms (Picture credit: Original)

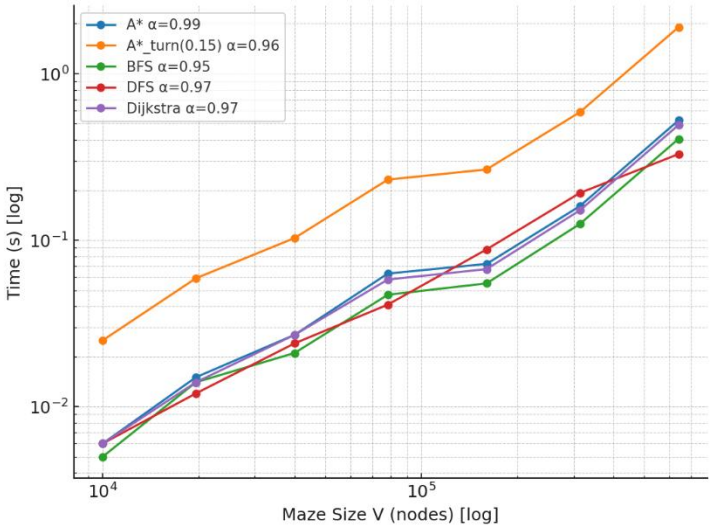

Fig. 5 Log–log plot of runtime versus maze size for different algorithms (Picture credit: Original)

Table 2. Averaged empirical scaling results for all algorithms

Algorithm	$\alpha_{expanded}$	$k_{expanded}$	α_{time}	k_{time}	
A*	0.95	0.96	0.98	0.0022	
A*_turn	0.95	1.30	0.96	0.0045	
BFS	0.94	1.01	0.96	0.0023	
DFS	0.96	0.97	0.98	0.0021	
Dijkstra	0.94	1.01	0.98	0.0022	

Fig. 4 and Fig. 5 present log–log plots of expanded nodes and runtime versus problem size, with fitted slopes labeled in the legends. Table 2 summarizes the average metrics at each scale.

Across all maze sizes from 100*100 to 800*800, BFS and DFS both exhibited nearly linear growth, with slopes of $\alpha \approx 0.94 - 0.96$ for expanded nodes and runtime. This is consistent with their theoretical $O(V)$ complexity. A* achieved similar slopes ($\alpha_{expanded} \approx 0.95, \alpha_{time} \approx 0.98$) but consistently expanded fewer nodes than BFS and DFS, reflecting the effectiveness of heuristic guidance. Dijkstra's algorithm also produced slopes near 1 ($\alpha_{expanded} = 0.94, \alpha_{time} = 0.98$). While theory predicts $O(VlogV)$ complexity due to heap operations, the logarithmic factor was not clearly visible within the tested scale (up to 6.4×10^5). A* with turn penalty showed slopes close to 1.0 ($\alpha_{expanded} = 0.95, \alpha_{time} = 0.96$) but consistently larger constants, because each state encodes both position and direction, effectively enlarging the search space.

The experimental slopes align well with theoretical expectations: BFS and DFS scale linearly with the number of vertices, and A* also remains linear but with fewer expansions thanks to heuristic information. Dijkstra is theoretically O(VlogV) but the additional logarithmic factor was not dominant in the tested range, explaining the nearly linear empirical slope. Nevertheless, its constant factors are higher due to repeated priority-queue operations, making it slower in practice compared to A*.

The turn-penalized A* demonstrates how state-space design influences practical cost. While its asymptotic order remains linear, the constant factor is larger because each cell is effectively split into four directional states. This highlights a trade-off between path quality (fewer turns) and computational overhead (more expansions and time).

Overall, the combination of averaged metrics (Table 1) and scaling analysis provides evidence that all tested algorithms are near-linear in complexity, but differ significantly in constant factors. These differences determine their relative practical efficiency: A* is the most efficient in practice,

BFS/DFS are reliable baselines, and Dijkstra, while theoretically optimal, is less competitive in this setting.

Recent literature has explored similar ideas to the turn-penalized A* variant implemented in this study. For example, an et al. proposed a hexagonal-grid A* with an adaptive movement cost and a turn-penalty term [10], which reduced unnecessary turns and node expansions while preserving optimality. Inspired by such approaches, future work could explore adaptive tuning of the turn penalty depending on local maze structure, as well as integrating any-angle path planning techniques.

4. Conclusion

This study systematically compared multiple maze-solving algorithms under a unified experimental framework. BFS and DFS demonstrated near-linear scaling but suffered from higher expansion counts or lack of optimality. Dijkstra's algorithm achieved optimality in weighted graphs but incurred larger constant factors due to repeated priority-queue operations. Standard A* with Manhattan distance proved to be the most efficient in practice, consistently reducing the number of expanded nodes while maintaining optimality. The turn-penalized A* variant produced paths with fewer turns, reflecting human-like navigation behavior, but introduced higher constant costs because of the enlarged state space.

Overall, the experiments show that simple heuristic modifications can significantly affect practical efficiency and path characteristics, even without changing asymptotic complexity. These findings suggest that human-inspired heuristics provide a valuable perspective: while they may not always improve performance, under certain structural conditions they yield more natural paths and remain computationally feasible. This work bridges classical algorithmic analysis with intuitive reasoning strategies, contributing both to theoretical understanding and to potential applications in robotics, navigation, and artificial intelligence.

References

[1] Abd Algfoor Z, Sunar MS, Kolivand H. A comprehensive study on pathfinding techniques for robotics and video games. Int J Comput Games Technol. 2015; 2015:736138.

[2] Fangfang. Research on power load forecasting based on Improved BP neural network. Harbin Institute of Technology; 2011.

[3] Barnouti NH, Al-Dabbagh SSM, Naser MAS. Pathfinding in strategy games and maze solving using A* search algorithm. J Comput Commun. 2016;4(11):15-25.

[4] Ansari A, Sayyed MA, Ratlamwala K, Shaikh P. An optimized hybrid approach for path finding. Int J Found Comput Sci Technol. 2015;5(2):47-58.

[5] Beamer S, Asanović K, Patterson D. Direction-optimizing breadth-first search. Sci Program. 2013;21(3-4):137-48.

[6] Foead D, Ghifari A, Kusuma MB, Hanafiah N, Gunawan E. A systematic literature review of A* pathfinding. Procedia Comput Sci. 2021; 179:507-14.

[7] Botea A, Bouzy B, Buro M, Bauckhage C, Nau D. Pathfinding in games. In: Lucas SM, Mateas M, Preuss M, Spronck P, Togelius J, editors. Artificial and computational intelligence in games. Dagstuhl Follow-Ups. Vol. 6. Schloss Dagstuhl–Leibniz-Zentrum für Informatik; 2013. p. 21-31.

[8] Cui X, Shi H. A*-based pathfinding in modern computer games. Int J Comput Sci Netw Secur. 2011;11(1):125-30.

[9] Kallem SR. Artificial Intelligence Algorithms. IOSR J Comput Eng. 2012;6(3):1-8.

[10] Gabrovšek P. Analysis of maze generating algorithms. IPSI Trans Internet Res. 2019;15(1):23-30.

[11] A Z, Rui X, Gao C. Improved A* navigation path-planning algorithm based on hexagonal grid. ISPRS Int J Geo-Inf. 2024;13(5):166.