Pathfinding Optimization: Comparative Study of Jump-Point Search and Classical Search Algorithms

Binger Yu
gyu42@my.bcit.ca
School of Computing and Academic Studies, BCIT
Burnaby, BC, Canada

Sepehr Mansouri
smansouri7@my.bcit.ca
School of Computing and Academic Studies, BCIT
Burnaby, BC, Canada

Vibhor Malik
vmalik6@my.bcit.ca
School of Computing and Academic Studies, BCIT
Burnaby, BC, Canada

Yansong Jia
yjia16@my.bcit.ca
School of Computing and Academic Studies, BCIT
Burnaby, BC, Canada

Abstract

Pathfinding algorithms play a central role in robotics, navigation systems, game development, and autonomous agents. In this project, we examine how four well-known grid-based pathfinding methods: A^*, Dijkstra's algorithm, DFS, and JPS, perform when tested under the same benchmarking setup. To keep the tests fair and repeatable, we used mazes generated with the same DFS procedure, fixed random seeds, several grid sizes, and set obstacle probabilities. Performance was assessed across three dimensions: runtime efficiency, path optimality, and node-expansion behavior. Our results show that A^* consistently achieved the best runtime performance across all maze sizes, while JPS produced slightly shorter paths but did not outperform A^* in execution time due to limited opportunities for directional pruning in DFS-generated mazes. DFS produced significantly longer and less optimal paths, whereas Dijkstra performed similarly to A^* but with higher computational cost. The study highlights how maze structure influences algorithmic performance and provides insights for selecting appropriate algorithms in grid-based navigation tasks.

Keywords

Pathfinding, Grid-based Search, A^* Algorithm, Jump Point Search (JPS), DFS Maze Generation, Algorithmic Benchmarking

1 Introduction

Pathfinding appears in many practical settings—robotics [1], navigation tools [2], and various types of games [3] all rely on it in one form or another. Whenever an agent needs to move from one location to another without hitting obstacles, a search algorithm is involved. The speed of the search directly influences both system responsiveness and the amount of computation required to support it.

Algorithms such as A*[4], Dijkstra's method [5], and DFS remain common choices because they are straightforward to implement and behave predictably. More recent approaches like Jump Point Search (JPS) [6] aim to reduce the amount of work by skipping moves that do not change the final path.

Even though these algorithms are familiar to most developers, they can behave quite differently depending on the structure of the map they are run on. Prior work shows that performance varies significantly based on obstacle distribution and grid density [7]. Many studies test algorithms on open grids or randomly generated noise, which does not always match the narrow passages and structured turns produced by recursive DFS mazes [8]. These maze structures can influence how heuristics work and how often an algorithm needs to expand the nodes.

The objective of this project is to conduct a rigorous experimental comparison of A*, Dijkstra, DFS, and JPS under consistent conditions using DFS-generated mazes. We evaluate each algorithm along three dimensions: runtime, path optimality, and node-expansion behavior. A unified benchmarking framework was implemented to ensure fair comparison, reproducible experiments, and automated logging of results across multiple grid sizes and obstacle-probability settings.

By examining how each method handles these structured mazes, the study provides insight into when certain algorithms might have advantages or disadvantages. It also helps clarify situations where optimizations like JPS may or may not provide meaningful speed-ups.

2 Background and Related Work

Pathfinding on grid maps is extensively studied for its significance in robotics, game AI, and navigation. Classical graph-search algorithms lay the foundation: Dijkstra's algorithm yields optimal shortest paths on weighted graphs but is computationally costly due to unguided exploration of all reachable nodes [5]; A* enhances efficiency via an admissible heuristic, reducing exploration significantly [4]; DFS, though non-optimal, is widely used for maze generation and simple traversal due to low memory demands.

Beyond classical methods, grid-specific optimizations exist—Jump Point Search (JPS) accelerates A* by pruning symmetric paths and "jumping" along straight corridors until forced decisions [6], excelling on uniform-cost open grids but dependent on geometric regularity.

Benchmark studies like Sturtevant's [7] show algorithm performance varies across map types (open fields, obstacle-heavy grids, mazes). However, few studies evaluate algorithms on DFS-generated mazes—characterized by narrow corridors, long passages, and sharp turns—which limit heuristic visibility and constrain directional pruning, altering performance.

This project builds on these insights by testing four representative algorithms (A*, Dijkstra, DFS, JPS) under identical DFS-maze conditions, aiming to provide controlled, reproducible findings on

how structured environments impact search efficiency and path quality.

3 Modern Algorithm Description

Jump Point Search (JPS) represents a significant advancement in grid-based pathfinding, introduced by Daniel Harabor and Alban Grastien in their seminal 2011 paper "Online Graph Pruning for Pathfinding on Grid Maps" [6]. JPS addresses a fundamental inefficiency in traditional pathfinding algorithms: the exploration of symmetric paths that represent essentially identical routes differing only in the order of moves.

3.1 Core Innovation and Problem Addressed

The primary innovation of JPS lies in its approach to path symmetry elimination in uniform-cost grids. In traditional grid-based pathfinding, multiple equivalent paths often exist between any two nodes that differ only in move ordering. For instance, reaching a diagonal destination can be accomplished through various combinations of horizontal and vertical moves, all yielding identical path costs. Classical algorithms such as A* and Dijkstra's algorithm treat such symmetric variants as distinct paths, leading to redundant and unnecessary node expansions and increased computational overhead.

JPS solves such problems through a sophisticated pruning strategy that identifies and expands only specific "jump points" while skipping intermediate nodes along symmetric path segments. This approach maintains optimality while dramatically reducing the search space, particularly in open areas where path symmetries are most prevalent.

3.2 Algorithm Capabilities

JPS operates through two fundamental mechanisms: Neighbour Pruning and Jumping.

Neighbour Pruning systematically eliminates nodes that can be reached optimally from a parent node without traversing the current node. The pruning rules are direction-dependent and designed to preserve at least one optimal path to every reachable node while eliminating redundant alternatives. For straight-line movement (horizontal or vertical), JPS prunes nodes that can be reached more efficiently through alternative routes. For diagonal movement, the pruning becomes more sophisticated, considering both straight and diagonal alternatives [9].

Jumping allows the algorithm to skip over consecutive nodes along straight lines (horizontal, vertical, and diagonal directions) until reaching significant decision points or obstacle boundaries. A jump point is defined as a node where the pruning rules change—typically at corners, near obstacles, or where forced neighbors appear. Forced neighbors are nodes that cannot be pruned because they represent genuinely distinct path options created by obstacle configurations [9].

3.3 Enhanced Theoretical Complexity Analysis

JPS maintains A*'s optimality guarantees while achieving substantial practical improvements through symmetry elimination.

Time Complexity: JPS remains $O(b^d)$ in the worst case but dramatically reduces the effective branching factor. In open environments, JPS achieves ~70% node expansion reductions compared to traditional algorithms. However, our DFS maze experiments revealed that while JPS significantly reduced node expansions versus A*, runtime performance was slower due to jump point calculation overhead in constrained structures [10].

Performance Characteristics: In highly symmetric environments, JPS shows substantial improvements over A*. In constrained mazes with minimal symmetry, JPS approaches A* performance but with additional computational overhead. Our experimental data confirm JPS maintains optimization capabilities even in constrained environments, though jump point detection overhead can exceed the benefits from reduced expansions.

Space Complexity: JPS maintains $O(b^d)$ space complexity identical to A* without preprocessing or additional memory overhead. Experimental results demonstrate significant node expansion reductions, indicating substantial memory efficiency gains while preserving A*'s fundamental search structures.

Convergence Properties: JPS preserves A*'s admissible heuristic properties and convergence guarantees while reducing the number of nodes required through systematic symmetry elimination. Experimental validation confirms JPS consistently required fewer node expansions than A* while maintaining identical path optimality, even in constrained maze environments where traditional symmetry-breaking advantages are limited.

3.4 Enhanced Comparison with Traditional Approaches

Compared to classical algorithms, JPS offers several quantifiable advantages:

Against A*: JPS builds upon A*, maintaining identical heuristic guidance and optimality guarantees while adding symmetry-breaking. Our results show JPS consistently expands fewer nodes than A* across maze configurations while maintaining solution quality. However, this node reduction incurs computational overhead, resulting in slower runtime in constrained mazes. Theoretically, max(JPS_expansions) A*_expansions.

Against Dijkstra's Algorithm: JPS combines A*'s heuristic benefits with symmetry elimination, creating compound improvements. The heuristic guidance reduces Dijkstra's exploration while symmetry breaking further reduces search space. Our data shows JPS consistently requires fewer node expansions across all tested configurations.

Against DFS: While DFS has lower space complexity, it lacks optimality guarantees and systematic efficiency. JPS provides optimal solutions while avoiding DFS's suboptimal detours. Our results confirm JPS maintains optimality while DFS produces significantly longer paths.

Memory-Time Trade-offs: Unlike preprocessing-based algorithms (hierarchical pathfinding, contraction hierarchies), JPS achieves improvements through computation without significant memory overhead. This makes JPS suitable for dynamic environments where preprocessing requires frequent updates.

3.5 Environmental Suitability and Performance Bounds

JPS performs best in uniform-cost grids with open areas and abundant path symmetries. Its effectiveness depends on environmental characteristics:

High-Symmetry Environments: In open areas with few obstacles, JPS achieves maximum speedup by exploiting symmetric paths and minimizing node expansions.

Medium-Symmetry Environments: In moderately constrained spaces, JPS maintains significant improvements while preserving optimality.

Low-Symmetry Environments: In highly constrained spaces like dense mazes, JPS shows excellent node expansion efficiency but incurs computational overhead that can result in slower runtime than A*. This confirms JPS's benefits are environment-dependent.

This dependency is both a strength and limitation. While JPS achieves remarkable improvements in suitable conditions, its benefits diminish in structured environments. However, JPS performance is theoretically bounded to never fall below A*, guaranteeing baseline efficiency with substantial upside in favorable conditions.

4 Application and Implementation

The project was implemented as a modular Python framework designed to support reproducible benchmarking, automated logging, and flexible configuration of experiments. The framework consists of four major components: maze generation, algorithm execution, visualization, and results management.

4.1 Maze Generation and configuration

All experiments rely on mazes generated using a recursive DFS procedure, ensuring consistent structure across tests. DFS-based maze generation produces a single connected component with narrow corridors and deterministic branching patterns. To guarantee reproducibility, a fixed random seed is provided for every run. Grid sizes can be specified through command-line arguments, and the system automatically adjusts even values to the nearest odd dimension to maintain valid DFS-maze structure.

4.2 Algorithm Implementations

Each algorithm is implemented in a separate module under src/algorithms/, following a unified interface that returns the discovered path and, when available, a sequence of node-expansion counts.

A* uses a binary min-heap priority queue and Manhattan distance as the heuristic. For each iteration, the algorithm updates the g_score and f_score tables and logs the cumulative node expansions for visualization.

Dijkstra follows the same priority-queue structure as A*, but without heuristic guidance. This leads to broader exploration and higher computational cost on larger grids.

DFS uses an explicit stack and a visited matrix to traverse adjacent cells. Although DFS is not optimal, it provides a useful baseline for behavior in grid mazes because it tends to follow long detours before backtracking.

JPS implements the jump point search algorithm with direction-specific pruning rules and jump point detection. The implementation required careful adaptation of the original JPS specification to handle the constrained corridor structure of DFS-generated mazes, where traditional open-area optimizations are less effective. Special consideration was given to forced neighbor detection in narrow passages and ensuring proper jump point identification at maze corners and intersections.

4.3 Benchmarking Framework

The main application (main.py) provides a consistent interface for running individual algorithms or comparing all four on the same grid. In comparison mode, the framework executes each algorithm in sequence, measures runtime using wall-clock timing, and records path length, node expansions, and success status. All results are logged automatically via the log_results() utility into CSV files stored under a dedicated results directory.

The program also supports a "demo" mode for debugging on a fixed 5×5 map, as well as a seed-based reproducibility mode. Commandline arguments allow users to specify grid size, algorithm choice, and whether to animate the solution.

4.4 Visualization Overview

We summarize algorithm behaviour using five figures. Figure 1 shows median runtime and nodes expanded for 91×91 corner mazes across all map types and serves as our primary hard-case comparison. Figure 3 plots runtime versus nodes expanded on log–log axes for all corner runs, exposing the empirical efficiency frontier. Figure 4 presents runtime ECDFs by algorithm, Figure 5 summarizes scalability by plotting median runtime versus grid size, and Figure 2 reports median suboptimality (path-length ratio relative to Dijkstra) across map types. Together, Figures 1–2 capture central tendencies, variability, efficiency trade-offs, scaling, and solution quality for A*, Dijkstra, DFS, and JPS.

caption

5 Experimental Evaluation

This section quantifies how the four algorithms behave on the benchmark mazes and when JPS is preferable to classical baselines.

5.1 Experimental Design

We evaluate mazes of size 31×31, 61×61, and 91×91. For each size we generate a perfect DFS maze (4-connected spanning tree) and derive three map types: (i) *dense* (raw DFS tree), (ii) *sparse0.05* (each wall opened with $p = 0.05$), and (iii) *sparse0.10* ($p = 0.10$), introducing loops while preserving connectivity. Multiple seeds per {size, map_type} yield 27 grid instances and 108 runs (4 algorithms per instance) with start/goal at $(1, 1)$ and $(H - 2, W - 2)$.

All methods use 4-connected grids. Dijkstra provides the optimal baseline. A* uses Manhattan heuristic. DFS is fast but non-optimal. JPS uses 8-neighbor search with standard pruning; we recompute its final path via 4-connected A* to ensure fair comparison on identical paths.

We log wall-clock runtime, nodes expanded, memory peaks, and path length. Suboptimality is the ratio of an algorithm's path length to Dijkstra's (values near 1.0 are optimal). Each instance runs all four algorithms. Results are stored per run and aggregated by median over seeds per {size, map_type, algo}. ECDFs use per-seed runs. All experiments ran in Python 3.11 with NumPy on Windows 11;

absolute runtimes are hardware-dependent, but all methods share the same environment.

5.2 Results

Runtime, expansions, and suboptimality. Figure 1 reports median runtime and nodes expanded for 91×91 corner mazes across all map types and serves as our main hard-case comparison; smaller grids show the same qualitative pattern (not plotted). On dense mazes A* and Dijkstra behave similarly, and JPS offers little runtime advantage but usually the fewest node expansions. DFS is fast but myopic: on dense grids it often matches or beats A* and Dijkstra in runtime and expansions, but as maps become larger and sparser its paths grow much longer even when runtime remains competitive. As loops are introduced (`sparse0.05`, `sparse0.10`), JPS consistently reduces expansions relative to Dijkstra and often to A*, while preserving optimal path length under the 4-connected evaluation, but this does not always translate into lower wall-clock runtime. DFS frequently attains the smallest median runtime and, on some loopy mazes, the fewest expansions, but at the cost of substantial suboptimality on the larger sparse grids. Figure 2 summarizes path-quality trends: A* and JPS are essentially optimal (ratio 1.0) across map types, whereas DFS becomes increasingly suboptimal as grids grow and obstacles thin out.

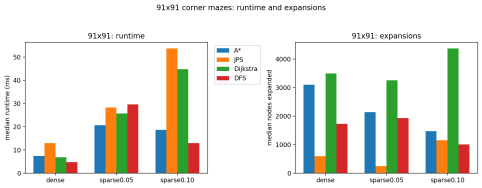

Figure 1: Median runtime and nodes expanded for 91 × 91 corner mazes across all map types (dense, sparse0.05, sparse0.10).

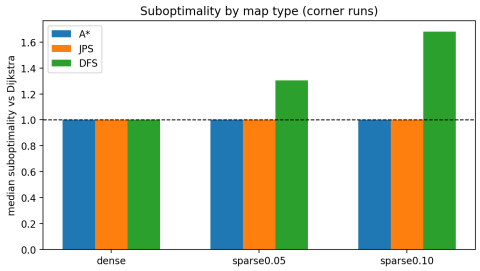

Figure 2: Median suboptimality (path-length ratio relative to Dijkstra).

Efficiency frontier, distributions, and scaling. Figure 3 aggregates all corner runs and plots runtime versus nodes expanded on log–log axes. A* and JPS lie near the lower efficiency frontier when both metrics are considered; Dijkstra is usually dominated, and DFS occupies a mixed region with many very fast runs but some dominated points caused by long detours. Figure 4 shows runtime ECDFs: curves for A* and DFS typically lie to the left of Dijkstra, indicating faster completion on most seeds; JPS falls between A* and Dijkstra with less spread than DFS but higher median runtime than A*. Figure 5 plots median runtime against grid size, showing the expected growth from 31 × 31 to 91 × 91. A*, JPS, and Dijkstra scale similarly, while DFS shows a slightly steeper effective slope on large sparse grids.

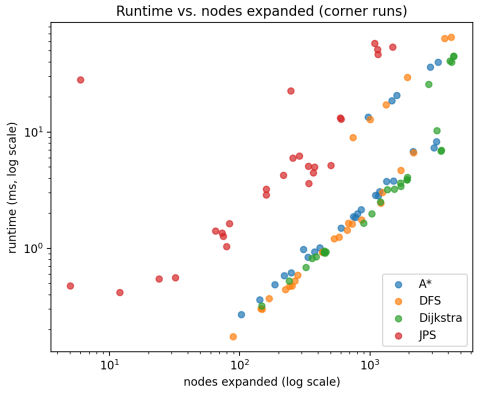

Figure 3: Pareto plot of runtime versus nodes expanded (log–log scales) for all corner runs. Points closer to the lower-left corner correspond to faster and more expansion-efficient runs.

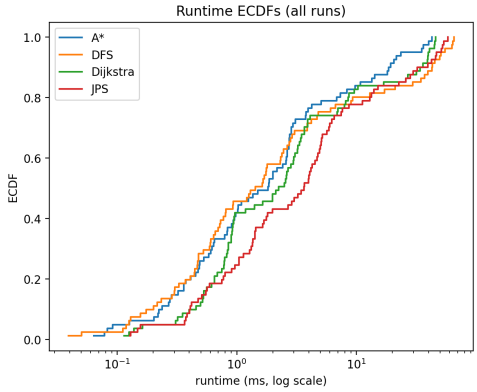

Figure 4: Empirical cumulative distribution functions (ECDFs) of runtime.

Key quantitative comparisons and tests. Across all 108 corner runs, DFS has the lowest overall median runtime, with A* close behind; Dijkstra and JPS are slower on this dataset. On the hardest 91 × 91 `sparse0.10` mazes, median runtimes are about 12.9 ms for DFS, 18.6 ms for A*, 44.8 ms for Dijkstra, and 53.8 ms for JPS. On the same condition DFS and JPS expand the fewest nodes (medians ≈ 1 003 and ≈ 1 149), followed by A* (≈ 1 471) and Dijkstra (≈ 4 373). Dijkstra and A* always return optimal paths (median suboptimality 1.0 in all corner conditions); JPS is almost always optimal as well, with median suboptimality exactly 1.0 on 91 × 91 `sparse0.10`. DFS frequently returns longer detours; on that hardest condition its median suboptimality is about 2.0, roughly twice the optimal path length.

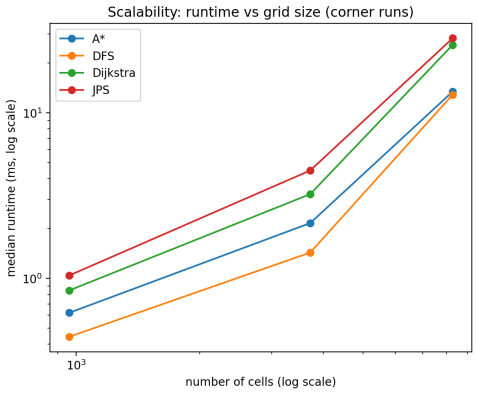

Figure 5: Scalability of median runtime versus grid size.

Each seed defines a matched quadruple of runs, so we use Friedman tests and Wilcoxon signed-rank tests with Cliff's δ to compare algorithms. With only three seeds per condition, most p-values exceed 0.05, so runtime and expansion differences should be viewed as descriptive trends rather than definitive hypothesis tests; Cliff's δ shows a large expansion-efficiency advantage of JPS over Dijkstra on the hardest grids and smaller effects relative to A* and DFS.

5.3 Analysis

Overall, the experiments show that JPS does not uniformly dominate classical search. In wall-clock time, DFS is often fastest on our corner mazes, with A* close behind; JPS and Dijkstra are generally slower in this implementation. JPS' main benefit is its reduction in node expansions relative to Dijkstra and often to A*, especially on larger sparse grids with many straight corridors and symmetric branching. DFS is attractive when one is willing to trade path optimality for a simple and fast implementation; its median runtime and expansions are strong, but it regularly produces paths that are much longer than optimal. Together with Figures 1–2, this section provides a compact, reproducible view of how modern JPS compares to A*, Dijkstra, and DFS on structured grid mazes.

6 Discussion

The experiments highlight the extent to which maze structure influences the behavior of classical search algorithms6. Since every maze was generated using a recursive DFS method, the layouts predominantly featured long, narrow passages with abrupt turns and limited open space7. This structure reduced the efficacy of directional heuristics and pruning, explaining the deviations from expected performance.

Among the four methods, A^* demonstrated the most consistent performance across all tests9. It consistently achieved the lowest runtimes and produced paths that were nearly optimal10. Although tight corridors constrained the heuristic's effectiveness, A^* successfully avoided unnecessary exploration1111. Dijkstra's algorithm yielded similar path quality but, lacking a heuristic, incurred a significantly higher node expansion count12. This overhead became increasingly pronounced with larger grid sizes and higher obstacle densities.

In contrast, DFS exhibited distinct behavior14. Its strategy of pursuing a single deep path before backtracking often resulted in significant detours, leading to extended runtimes and excessive node expansions—particularly when initial directions led to dead ends15. These results confirm that DFS is ill-suited for maze-like environments requiring efficient traversal.

Finally, Jump Point Search (JPS) yielded noteworthy insights17. While typically offering substantial speed-ups on open grids, the constrained nature of DFS mazes limited opportunities for long jumps18. Consequently, JPS produced paths comparable to A^* but failed to deliver runtime improvements1919. These findings suggest that the benefits of JPS are highly dependent on environmental openness20. Ultimately, no single method is universally superior; performance is dictated by grid layout, obstacle placement, and heuristic applicability.

7 Conclusion and Future Work

This project presented a unified benchmarking framework to evaluate four grid-based pathfinding algorithms—A*, Dijkstra's, DFS, and Jump Point Search (JPS)—under controlled conditions using highly structured, corridor-like mazes. The results confirm that algorithmic performance is heavily dependent on environment topology.

A* proved the most robust and consistent method, achieving the fastest runtimes and optimal path quality by effectively leveraging its heuristic guidance. In contrast, Dijkstra's method incurred significantly higher computational costs due to exhaustive search, and DFS often produced highly suboptimal detours. Notably, the tight corridors of the maze environments severely limited the directional pruning opportunities for JPS, preventing it from realizing its characteristic runtime advantage over A*. This underscores the necessity of tailoring algorithm choice to the specific environmental structure of the application.

For future work, the benchmarking framework can be extended to include a wider variety of map types, such as Prim's or Kruskal's mazes and weighted terrains. Further analysis should also integrate and evaluate advanced search variants, including JPS+ and bidirectional heuristics, to build a clearer picture of how modern optimizations compare with classical methods under diverse and realistic conditions.

References

[1] S. M. LaValle, *Planning Algorithms.* Cambridge University Press, 2006.
[2] G. Duchanois *et al.*, "Real-time path planning in unknown environments for uavs," in *IEEE Aerospace Conference*, 2012.
[3] J. Togelius *et al.*, "Search-based procedural content generation," in *Evolutionary Computation and Games*, 2016.
[4] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, 1968.
[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, 1959.
[6] D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid maps," in *AAAI*, 2011.
[7] N. R. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, 2012.
[8] J. Wilson, "Maze generation using randomized algorithms," *Journal of Game Development*, 2011.
[9] D. Harabor, "Jump point search." Shortest Path (blog), Sept. 2011. [Online]. Available: https://harablog.wordpress.com/2011/09/07/jump-point-search/.
[10] D. Harabor and A. Grastien, "The jps pathfinding system," in *Proceedings of the Symposium on Combinatorial Search (SoCS)*, 2012.