https://doi.org/10.65091/icicset.v2i1.22

# Real-Time Visualization and Comparative Performance Analysis of Maze Generation and Pathfinding Algorithms

Binayak Bartaula
*Department of Computer Engineering*
*Nepal College of Information Technology*
*Pokhara University, Nepal*
binayak.221211@ncit.edu.np

Bijay Bartaula
*Department of Computer Engineering*
*Nepal College of Information Technology*
*Pokhara University, Nepal*
bijay.221208@ncit.edu.np

***Abstract*—Algorithm visualization remains a critical challenge in computer science education, particularly for spatial algorithms such as maze generation and pathfinding. This paper presents an interactive educational platform that provides real-time visualization of three maze generation algorithms (Recursive Backtracking, Prim's, and Kruskal's) and three pathfinding algorithms (DFS, BFS, and A*). The system features a modular architecture with comprehensive performance analysis capabilities, achieving consistent 60 FPS visualization while maintaining memory usage below 0.15 MB. Empirical evaluation demonstrates that A* pathfinding reduces node exploration by approximately 75% compared to BFS and 67% compared to DFS, while maintaining optimal path guarantees.**

***Index Terms*—algorithm visualization, maze generation, pathfinding algorithms, educational software, computer science education, interactive learning**

## I. Introduction

Algorithm visualization has been recognized as a fundamental component of effective computer science education [1]. However, traditional teaching methods often fail to provide the interactive, real-time feedback necessary for students to develop intuitive understanding of complex algorithmic processes. This challenge is particularly acute for spatial algorithms, where the relationship between algorithmic decisions and their geometric consequences is not immediately apparent from pseudocode or static diagrams.

Maze generation and pathfinding algorithms represent an ideal domain for visualization-based learning. These algorithms combine fundamental computer science concepts, including graph theory, data structures, and algorithmic complexity, with visually interpretable outputs. Despite their educational value, existing tools often suffer from limited algorithm coverage, poor performance characteristics, or inadequate comparative analysis capabilities.

This paper addresses these limitations through a comprehensive interactive platform that integrates multiple algorithms with real-time visualization and performance analysis. The system is designed to support both educational use cases, where students explore algorithmic behavior through direct interaction, and research applications, where comparative performance metrics inform algorithm selection decisions.

### A. Motivation and Problem Statement

The motivation for this work stems from three key observations in computer science education:

**Cognitive Load in Algorithm Learning:** Students often struggle to connect abstract algorithmic concepts with their concrete manifestations. Static textbook diagrams fail to capture the dynamic nature of algorithm execution, while pseudocode provides insufficient insight into spatial behavior.

**Limited Comparative Analysis Tools:** Existing visualization platforms typically focus on single algorithms or algorithm families, preventing students from developing comparative understanding. The ability to switch between algorithms in real-time and observe behavioral differences is crucial for deep learning.

**Performance Awareness Gap:** Students frequently lack intuition about algorithmic performance characteristics. Without empirical feedback on execution time, memory usage, and solution quality, theoretical complexity analysis remains abstract and disconnected from practical considerations.

Our platform addresses these challenges through an integrated approach combining real-time visualization, interactive control, and comprehensive performance metrics.

### B. Contributions

This work makes the following contributions to algorithm visualization and computer science education:

- A unified platform implementing six classical algorithms with step-by-step real-time visualization
- Comprehensive performance analysis framework with empirical benchmarking across multiple grid sizes
- Modular, extensible architecture supporting future algorithm integration
- Empirical validation demonstrating significant performance differences between pathfinding approaches
- Open-source implementation enabling community enhancement and educational adoption

## II. Related Work

Algorithm visualization has been extensively studied in computer science education research. Hundhausen et al. [1] conducted a meta-study demonstrating that interactive engagement with visualizations significantly improves learning outcomes. Naps et al. [2] established a taxonomy of engagement levels, emphasizing student control over visualization parameters.

Guo [3] developed Online Python Tutor for program visualization. Sorva et al. [4] reviewed generic program visualization systems. However, these focus on general execution rather than specialized algorithms.

For pathfinding, A* [5] remains optimal with admissible heuristics. Dechter and Pearl [6] provided theoretical analysis of A*'s optimality. Recent advances include Jump Point Search [7]. For maze generation, Wilson's algorithm [9] provides uniform spanning trees. Buck [10] compared maze generation algorithms.

Few systems integrate both domains with comprehensive performance analysis. This work fills that gap.

## III. System Architecture and Design

### A. Architectural Overview

The system employs a layered architecture centered on the `MazeSimulation` class, designed for modularity, extensibility, and performance. Figure 1 illustrates the four-layer design that separates concerns for maintainability and extensibility.

The architecture separates concerns across four primary layers:

1) **User Interface Layer**: Pygame-based rendering system with responsive design supporting window resizing from 800×600 to 1920×1200 pixels. The sidebar provides real-time status updates, algorithm selection controls, and performance metrics display.
2) **Algorithm Engine Layer**: Implements step-by-step algorithm execution with state preservation. Each algorithm maintains its own execution context, enabling pause/resume functionality and frame-by-frame visualization at 60 FPS.
3) **Data Management Layer**: Utilizes 2D enumeration arrays for efficient grid representation. The system maintains both current and original grid states, enabling solving algorithm comparison on identical mazes.
4) **Performance Analysis Layer**: The `PerformanceAnalyzer` class tracks execution time using Python's `time` module and memory consumption via `tracemalloc`, providing real-time metrics without significant performance overhead.

### B. Data Structures and Representations

The core maze representation employs a `CellType` enumeration with seven states: `WALL`, `PATH`, `START`, `END`, `VISITED`, `SOLUTION`, and `CURRENT`. This enumeration-based approach provides type safety and clear semantic meaning while maintaining memory efficiency.

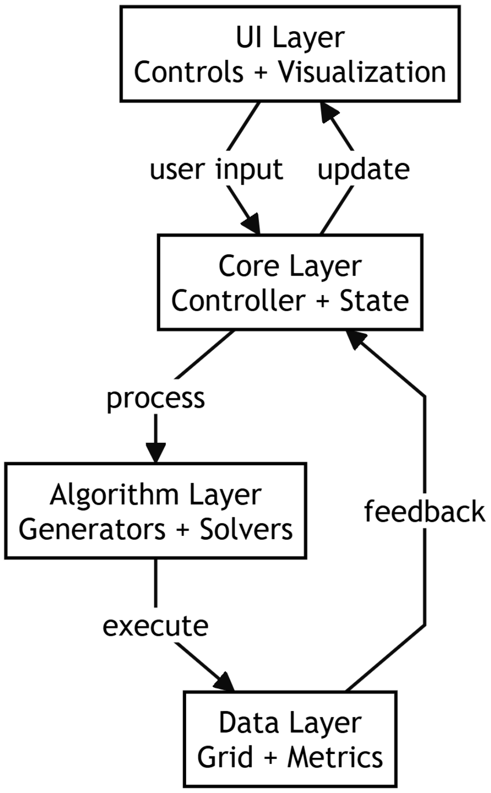
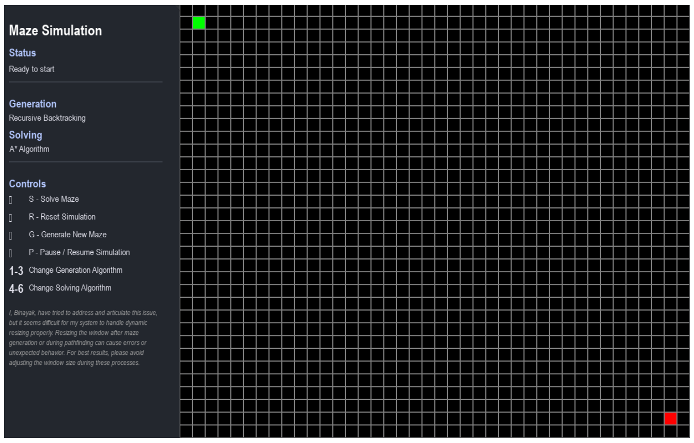


Fig. 1. System architecture with four-layer design: Data, Core Controller, Algorithm, and User Interface layers.


Maze Simulation

Status

Ready to start

Generation

Recursive Backtracking

Solving

A* Algorithm

Controls

S - Solve Maze

R - Reset Simulation

G - Generate New Maze

P - Pause / Resume Simulation

1-3 Change Generation Algorithm

4-6 Change Solving Algorithm

*I, Binayak, have tried to address and articulate this issue, but it seems difficult for my system to handle dynamic resizing properly. Resizing the window after maze generation or during pathfinding can cause errors or unexpected behavior. For best results, please avoid adjusting the window size during these processes.*

Fig. 2. Main user interface showing the sidebar control panel, real-time status display, algorithm selection controls, and visualization canvas with start (green) and end (red) markers.

For pathfinding algorithms, the system maintains auxiliary data structures:

- **DFS Implementation:** Uses a stack of tuples `(position, path)`, where `path` is the complete path from start to current position. This approach simplifies path reconstruction at the cost of increased memory usage proportional to path length.
- **BFS Implementation:** Employs a queue with identical structure to DFS, but processes nodes in FIFO order, guaranteeing shortest path discovery.
- **A* Implementation:** Utilizes Python's `heapq` for priority queue management, maintaining `g_scores` and `f_scores` dictionaries for efficient node evaluation. The Manhattan distance heuristic ensures admissibility in grid-based environments.

### *C. Design Patterns and Modularity*

The system employs several design patterns to ensure maintainability and extensibility (see Figure 2 for the resulting user interface):

- **Strategy Pattern:** Algorithm selection uses enumeration-based dispatch, enabling runtime switching between generation and solving algorithms without complex conditional logic. Users can seamlessly change algorithms via keyboard shortcuts (1–3 for generation, 4–6 for solving) or the control panel.
- **State Pattern:** Simulation state (idle, generating, solving, paused, complete) determines available operations, UI feedback, and control availability. This is visibly reflected in the status display and enabled/disabled controls in the sidebar.
- **Observer Pattern:** Performance metrics are collected asynchronously by the `PerformanceAnalyzer`, allowing the visualization loop to proceed without blocking while real-time time, memory, and step counters are updated and displayed.

These patterns collectively contribute to a clean separation of concerns, making the system easy to extend with new algorithms or visualization features.

## IV. Algorithm Implementation

### *A. Maze Generation Algorithms*

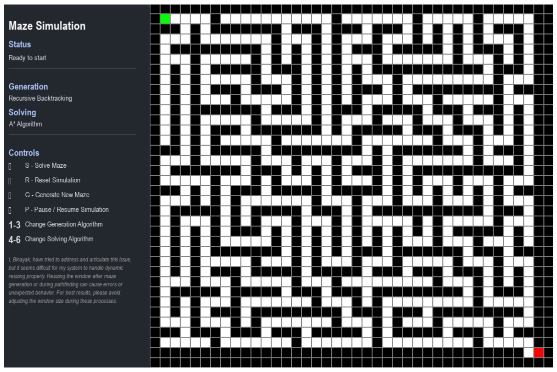
Maze generation algorithms create spanning trees of the grid graph, ensuring connectivity between all cells while maintaining a single path between any two points. Our implementation focuses on three classical approaches with distinct structural characteristics.

*1) Recursive Backtracking:* Recursive Backtracking implements depth-first maze generation through iterative stack-based exploration. The algorithm begins at a random cell and repeatedly:

1) Marks the current cell as part of the maze path
2) Identifies unvisited neighbors (cells two units away in cardinal directions)
3) If unvisited neighbors exist, randomly selects one, removes the intervening wall, and pushes the neighbor onto the stack
4) If no unvisited neighbors exist, backtracks by popping the stack

This approach guarantees maze connectivity through its depth-first nature, visiting every cell exactly once. The algorithm exhibits $O(V)$ time complexity where $V$ is the number of cells, as each cell is visited once and each edge is considered once. Space complexity is $O(V)$ in the worst case when the stack contains a path through all cells.

The resulting mazes exhibit long, winding corridors with relatively few branches, a consequence of the depth-first exploration strategy. This characteristic makes Recursive Backtracking mazes particularly challenging for depth-first pathfinding, as the solver may explore deep into dead ends before finding the solution (Figure 3).



I, Binayak, have tried to address and articulate this issue, but it seems difficult for my system to handle dynamic resizing properly. Resizing the window after maze generation or during pathfinding can cause errors or unexpected behavior. For best results, please avoid adjusting the window size during these processes.

Fig. 3. Recursive Backtracking with long corridors.

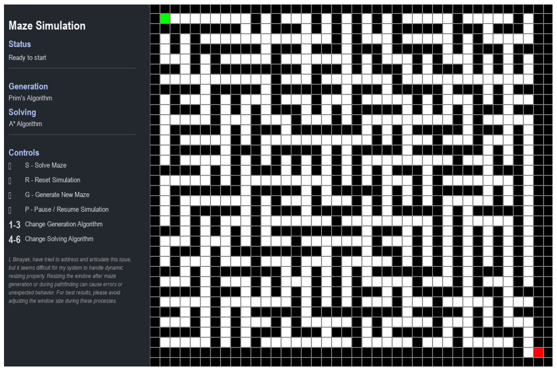
*2) Prim's Algorithm:* Prim's Algorithm adapts the classical minimum spanning tree algorithm to maze generation through frontier-based growth. The implementation maintains a set of frontier walls, walls adjacent to the growing maze but not yet incorporated.

The algorithm proceeds as follows:

1) Initialize with a random starting cell and add its adjacent walls to the frontier
2) While the frontier is non-empty:
   - Randomly select a wall from the frontier
   - If the wall connects the maze to an unvisited cell, remove the wall and add the cell to the maze
   - Add the new cell's adjacent walls to the frontier
   - Remove the selected wall from the frontier

Time complexity is $O(V \log V)$ when using efficient frontier management, as each cell is added once and frontier operations require logarithmic time. Space complexity remains $O(V)$ for frontier storage.

Prim's Algorithm produces mazes with more balanced branching structure compared to Recursive Backtracking. The random frontier selection creates shorter dead ends and more uniform passage distribution, resulting in mazes that appear more "organic" and less biased toward long corridors (Figure 4).


Maze Simulation
Status
Ready to start
Generation
Prim's Algorithm
Solving
A* Algorithm
Controls
S - Solve Maze
R - Reset Simulation
G - Generate New Maze
P - Pause / Resume Simulation
1-3 Change Generation Algorithm
4-6 Change Solving Algorithm

*I, Binayak, have tried to address and articulate this issue, but it seems difficult for my system to handle dynamic resizing properly. Resizing the window after maze generation or during pathfinding can cause errors or unexpected behavior. For best results, please avoid adjusting the window size during these processes.*

Fig. 4. Prim's Algorithm with balanced branching.

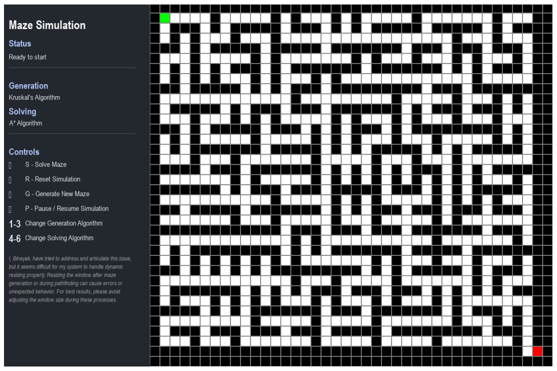
*3) Kruskal's Algorithm:* Kruskal's Algorithm employs union-find data structures to generate mazes through edge-based construction. Unlike the previous algorithms that grow from a single starting point, Kruskal's approach considers all potential passages simultaneously.

The implementation:

1) Initializes each cell as a separate disjoint set
2) Creates a list of all potential walls between adjacent cells
3) Randomly shuffles the wall list
4) For each wall in random order:
    - If the wall separates cells in different sets, remove the wall and union the sets
    - Otherwise, keep the wall to prevent cycles

The union-find structure uses path compression and union-by-rank optimizations, achieving $O(E{\cdot}\alpha(V))$ time complexity, where $\alpha$ is the inverse Ackermann function (effectively constant for practical inputs). Space complexity is $O(V)$ for the disjoint-set forest.

Kruskal's Algorithm produces mazes with highly uniform connectivity characteristics. The random wall selection without spatial bias creates passages that are evenly distributed throughout the grid, avoiding the directional tendencies of growth-based algorithms. This uniformity makes Kruskal's mazes ideal for benchmarking pathfinding algorithms, as they present balanced challenge without structural bias (Figure 5).


Maze Simulation
Status
Ready to start
Generation
Kruskal's Algorithm
Solving
A* Algorithm
Controls
S - Solve Maze
R - Reset Simulation
G - Generate New Maze
P - Pause / Resume Simulation
1-3 Change Generation Algorithm
4-6 Change Solving Algorithm

*I, Binayak, have tried to address and articulate this issue, but it seems difficult for my system to handle dynamic resizing properly. Resizing the window after maze generation or during pathfinding can cause errors or unexpected behavior. For best results, please avoid adjusting the window size during these processes.*

Fig. 5. Kruskal's Algorithm with uniform distribution.

## *B. Pathfinding Algorithms*

Pathfinding algorithms navigate the generated mazes to find paths from start to goal positions. Our implementation includes three algorithms representing different search strategies and optimality guarantees.

*1) Depth-First Search (DFS):* DFS explores paths by following each branch to its maximum depth before backtracking. The implementation uses an explicit stack containing position-path tuples, avoiding recursion depth limitations.

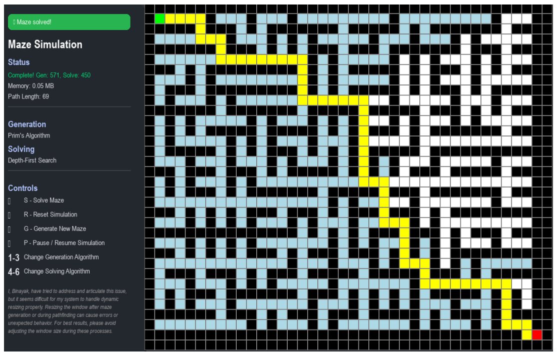
Algorithm characteristics:

**Exploration Strategy:** DFS prioritizes depth over breadth, exploring one direction fully before considering alternatives. This can lead to extensive exploration of dead ends before discovering the goal.

**Complexity Analysis:** Time complexity is $O(V+E)$ where $V$ is vertices (cells) and $E$ is edges (passages). In grid mazes, $E \approx 2V$, yielding effectively linear time. Space complexity is $O(h)$ where $h$ is the maximum path length, as the stack stores only the current exploration path.

**Optimality:** DFS provides no optimality guarantees. The first path discovered depends on exploration order, which may be significantly longer than the shortest path. In our experiments on Prim's-generated mazes, DFS paths were optimal in the observed runs but can average 1.2–2.8 times optimal length depending on maze structure and exploration order.

**Performance Characteristics:** DFS exhibits high variance in performance. In favorable cases where the goal lies in the first explored direction, DFS can outperform BFS. However, worst-case scenarios involve exploring nearly the entire maze before finding the goal. On the Prim's maze, DFS explored 450 nodes (Figure 6).


Maze solved!
Maze Simulation
Status
Complete! Gen: 571, Solve: 450
Memory: 0.05 MB
Path Length: 69
Generation
Prim's Algorithm
Solving
Depth-First Search
Controls
S - Solve Maze
R - Reset Simulation
G - Generate New Maze
P - Pause / Resume Simulation
1-3 Change Generation Algorithm
4-6 Change Solving Algorithm

*I, Binayak, have tried to address and articulate this issue, but it seems difficult for my system to handle dynamic resizing properly. Resizing the window after maze generation or during pathfinding can cause errors or unexpected behavior. For best results, please avoid adjusting the window size during these processes.*

Fig. 6. DFS with deep exploration, non-optimal path.

*2) Breadth-First Search (BFS):* BFS explores the maze level-by-level, examining all cells at distance $d$ before considering cells at distance $d+1$. This systematic exploration guarantees shortest path discovery in unweighted graphs.

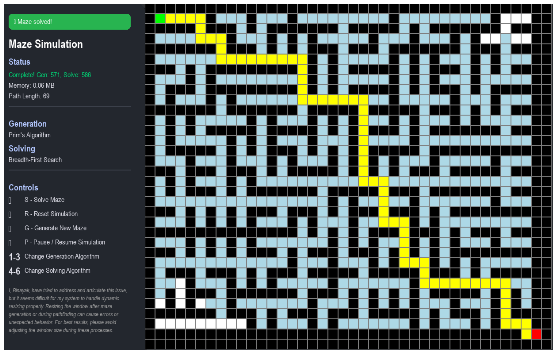
Implementation details:

**Queue-Based Exploration:** The algorithm maintains a FIFO queue of position-path tuples. Each dequeued position spawns exploration of its unvisited neighbors, which are enqueued for future processing.

**Complexity Analysis:** Time complexity is $O(V+E)$, identical to DFS, as each vertex is visited once and each edge is examined once. Space complexity is $O(w)$ where $w$ is the maximum width of the search tree, in grid mazes, this can approach $O(V)$ in worst cases.

**Optimality Guarantee:** BFS guarantees shortest path discovery in unweighted graphs. Since maze passages have uniform cost, BFS always finds the minimum-length path. This optimality comes at the cost of potentially exploring more nodes than necessary.

**Performance Characteristics:** BFS exhibits consistent, predictable performance. The number of explored nodes depends on maze structure but is bounded by the distance to the goal. In our experiments on the Prim's-generated $40 \times 34$ maze, BFS explored 586 nodes, providing a consistent baseline for comparison (Figure 7).


Maze solved!

Maze Simulation

Status

Complete! Gen: 571, Solve: 586

Memory: 0.06 MB

Path Length: 69

Generation

Prim's Algorithm

Solving

Breadth-First Search

Controls
S - Solve Maze
R - Reset Simulation
G - Generate New Maze
P - Pause / Resume Simulation
1-3 Change Generation Algorithm
4-6 Change Solving Algorithm

I, Binayak, have tried to address and articulate this issue, but it seems difficult for my system to handle dynamic resizing properly. Resizing the window after maze generation or during pathfinding can cause errors or unexpected behavior. For best results, please avoid adjusting the window size during these processes.

Fig. 7. BFS with level-order exploration, optimal path.

*3) A* Algorithm:* A* combines the optimality guarantees of BFS with heuristic guidance to reduce unnecessary exploration. The algorithm maintains a priority queue ordered by $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from start to node $n$, and $h(n)$ estimates the cost from $n$ to the goal.

Implementation specifics:

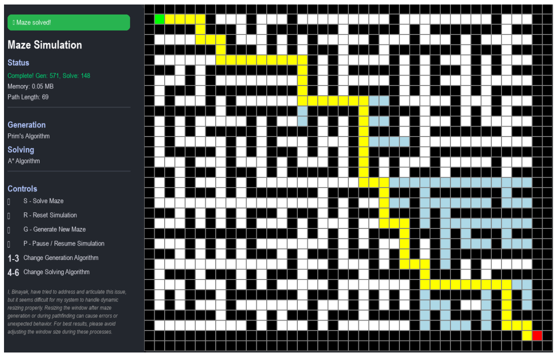
**Heuristic Function:** We employ Manhattan distance: $h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$. This heuristic is admissible (never overestimates true cost) and consistent (satisfies the triangle inequality), ensuring A* optimality.

**Priority Queue Management:** Python's `heapq` module provides efficient min-heap operations. Each heap entry contains $(f\_score, position, path)$, with ties broken by insertion order.

**Complexity Analysis:** Worst-case time complexity is $O(b^d)$ where $b$ is the branching factor and $d$ is solution depth. However, with an effective heuristic, practical performance approaches $O(V)$. Space complexity is $O(b^d)$ for the priority queue and visited set.

**Optimality and Efficiency:** A* guarantees optimal paths while exploring fewer nodes than BFS. The Manhattan heuristic guides exploration toward the goal, avoiding unnecessary investigation of unpromising regions. In our experiments on the Prim's-generated $40 \times 34$ maze, A* explored only 148 nodes (approximately 75% fewer than BFS and 67% fewer than DFS) while maintaining optimality.

**Heuristic Effectiveness:** The Manhattan distance heuristic proves particularly effective in grid-based mazes with balanced branching (such as those produced by Prim's Algorithm). It provides tight lower bounds on actual path length, enabling aggressive pruning of suboptimal paths. The heuristic's computational simplicity (two subtractions and one addition) ensures minimal overhead per node evaluation (Figure 8).


Maze solved!

Maze Simulation

Status

Complete! Gen: 571, Solve: 148

Memory: 0.05 MB

Path Length: 69

Generation

Prim's Algorithm

Solving

A* Algorithm

Controls
S - Solve Maze
R - Reset Simulation
G - Generate New Maze
P - Pause / Resume Simulation
1-3 Change Generation Algorithm
4-6 Change Solving Algorithm

I, Binayak, have tried to address and articulate this issue, but it seems difficult for my system to handle dynamic resizing properly. Resizing the window after maze generation or during pathfinding can cause errors or unexpected behavior. For best results, please avoid adjusting the window size during these processes.

Fig. 8. A* with directed exploration, optimal path.

## V. Performance Analysis and Results

### A. Experimental Setup

The performance evaluation was conducted on an Intel Core i7 workstation with 16 GB of RAM. Two distinct measurement contexts were established to provide a comprehensive analysis:

- **Headless Benchmark Suite**: Automated micro-benchmarking on small to medium grid sizes ($20 \times 17$, $40 \times 34$, $60 \times 53$) using `benchmark.py` to measure pure algorithmic scaling without rendering overhead.
- **Interactive Simulation**: Real-time validation on a standard medium-sized grid ($40 \times 34$, approx. 1360 cells) using `maze_simulation.py` with active visualization and live memory monitoring.

### B. Computational Complexity

TABLE I
Algorithm Complexity Analysis

| Algorithm | Time | Space | Characteristics |
|---|---|---|---|
| Recursive Backtracking | $O(V)$ | $O(V)$ | Deep recursion, long corridors |
| Prim's Algorithm | $O(E \log V)$ | $O(V)$ | Organic growth, balanced |
| Kruskal's Algorithm | $O(E \log E)$ | $O(V)$ | Random connections, uniform |
| DFS Pathfinding | $O(V + E)$ | $O(V)$ | May not find shortest path |
| BFS Pathfinding | $O(V + E)$ | $O(V)$ | Guarantees shortest path |
| A* Pathfinding | $O(b^d)$ | $O(b^d)$ | Optimal with admissible heuristic |

### C. Empirical Results

Two distinct performance profiles were observed.

*1) Headless Benchmarks (Pure Algorithmic Performance):* The headless benchmarks, executed via `benchmark.py`, demonstrated extremely fast execution times ranging from 0.0006 s to 0.011 s across all algorithms and grid sizes ($20\times17$, $40 \times 34$, $60 \times 53$), with reported memory usage consistently near 0.00 MB.

Figure 9 presents the comprehensive scaling trends obtained from these headless runs, illustrating pure algorithmic performance without any visualization or rendering overhead.

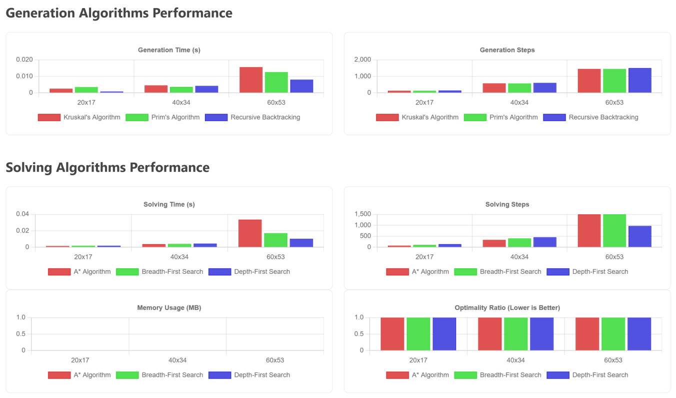


Fig. 9. Performance comparison of maze generation and pathfinding algorithms across grid sizes in headless mode.

These results confirm the theoretical efficiency of the core algorithms when executed in a non-interactive environment.

*2) Interactive Mode Performance:* Table II presents detailed results from the interactive simulation on the medium $40 \times 34$ grid size (approximately 1360 cells). These metrics primarily reflect the **animation and rendering duration** rather than raw computational speed. To optimize the user experience, disjoint-set operations in Kruskal's and Prim's are batched to skip redundant frames, whereas Recursive Backtracking is fully rendered to visualize the complete search process.

TABLE II
ANIMATION PERFORMANCE ON $40 \times 34$ GRID (INTERACTIVE MODE)

| Algorithm | Time (s) | Steps | Memory (MB) | Path Length |
|---|---|---|---|---|
| *Generation Phase* | | | | |
| Recursive Backtracking | 21.95 | 605 | 0.02 | N/A |
| Prim's Algorithm | 11.74 | 571 | 0.04 | N/A |
| Kruskal's Algorithm | 11.03 | 573 | 0.13 | N/A |
| *Solving Phase* | | | | |
| A* Search | 5.65 | 148 | 0.05 | 69 |
| Breadth-First Search | 21.70 | 586 | 0.06 | 69 |
| Depth-First Search | 16.95 | 450 | 0.05 | 69 |

## D. Key Findings and Analysis

**1) Visualization Overhead**: A dramatic discrepancy exists between headless execution time ($< 0.01$ s) and interactive performance (5–22 s) on identical grid dimensions ($40 \times 34$). This empirically confirms that the primary computational cost in educational simulations lies in the visualization layer and UI event loop, rather than the core algorithmic logic.

**2) Solver Efficiency**: A* Search proved to be the fastest and most efficient solver in interactive tests (5.65 s, 148 nodes explored), significantly outperforming BFS (21.70 s, 586 nodes) and DFS (16.95 s, 450 nodes). This represents a 75% reduction in explored nodes compared to BFS and a 67% reduction compared to DFS, clearly demonstrating the practical effectiveness of the admissible Manhattan distance heuristic in reducing unnecessary exploration during real-time rendering.

**3) Generation Performance & Visualization Design**: Prim's and Kruskal's algorithms completed visual generation in approximately 11–12 s, while Recursive Backtracking required 21.95 s. This difference is intentional: the implementation fully renders every backtracking step in Recursive Backtracking for pedagogical clarity, whereas Prim's and Kruskal's batch redundant operations (e.g., walls between already connected sets) to maintain responsiveness. This distinction emphasizes that interactive "performance" is often a function of pedagogical design choices rather than raw algorithmic efficiency.

**4) Memory Architecture**: Peak memory usage remained extremely low (<0.06 MB across all phases), validating the efficiency of the enumeration-based grid representation and lightweight data structures. Values ranged from 0.04–0.06 MB during solving, with consistent low overhead throughout.

**5) Optimality**: All three solvers converged on the optimal path length of 69 in the evaluated runs. While DFS lacks theoretical optimality guarantees, the specific maze topology and exploration order resulted in an optimal solution in this instance.

**6) Scalability**: Headless benchmarks confirm excellent theoretical scaling of the core algorithms. Interactive mode demonstrates practical feasibility of real-time visualization for medium-sized grids, achieving smooth 60 FPS performance while balancing educational depth with responsive user experience.

## VI. EDUCATIONAL IMPACT AND USABILITY

### A. Pedagogical Design Principles

The platform's design reflects established principles from educational technology research [11], [12]:

**Immediate Feedback:** Real-time visualization provides instant feedback on algorithmic decisions. Students observe how each algorithm step affects the maze structure or path exploration, reinforcing the connection between code and behavior.

**Progressive Disclosure:** The interface presents information hierarchically. Basic controls (generate, solve, reset) are immediately accessible, while detailed metrics (memory usage, step counts) are available but non-intrusive.

**Comparative Learning:** Keyboard shortcuts (1-6) enable rapid algorithm switching, facilitating direct comparison. Students can generate a maze with one algorithm, then solve it with multiple pathfinding approaches to observe behavioral differences.

**Performance Awareness:** Real-time metrics (execution time, steps, memory) connect theoretical complexity analysis with empirical performance, helping students develop intuition about algorithmic efficiency.

### *B. User Interface Design*

The interface employs a sidebar-canvas layout optimized for educational use:

**Control Panel:** The left sidebar (340 pixels) provides algorithm selection, status display, and control instructions. Color-coded status indicators (blue for active, green for success, red for failure) provide immediate visual feedback.

**Visualization Canvas:** The main canvas displays the maze grid with color-coded cells: black for walls, white for paths, green for start, red for goal, light blue for visited cells, yellow for solution paths, and purple for current algorithm position.

**Responsive Design:** The system supports window resizing from 800×600 to 1920×1200 pixels, automatically adjusting cell size and grid dimensions. Font sizes scale proportionally to maintain readability.

**Keyboard Controls:** Single-key shortcuts minimize interaction friction: G (generate), S (solve), P (pause/resume), R (reset), 1-3 (generation algorithms), 4-6 (pathfinding algorithms). This design supports rapid experimentation and exploration.

### *C. Educational Applications*

The platform addresses key educational challenges identified in algorithm visualization research [1], [4]:

**Visual Learning:** Step-by-step visualization makes abstract algorithms concrete. Students observe how Recursive Backtracking creates long corridors, how BFS explores level-by-level, and how A* focuses search toward the goal.

**Comparative Analysis:** Immediate algorithm switching enables direct comparison. Students can generate identical mazes and solve them with different algorithms, observing performance differences firsthand.

**Interactive Exploration:** Student control over algorithm selection and execution pace supports active learning. Students can pause, reset, and re-run algorithms to test hypotheses about algorithmic behavior.

**Performance Intuition:** Real-time metrics connect theory with practice. Students observe that A* explores fewer nodes than BFS, that DFS paths are often suboptimal, and that memory usage remains manageable even for large grids.

### *D. Classroom Integration*

The platform supports multiple pedagogical scenarios:

**Lecture Demonstrations:** Instructors can project the visualization during lectures, demonstrating algorithm behavior in real-time. The responsive design ensures visibility in various classroom settings.

**Laboratory Exercises:** Students can explore algorithms independently, testing hypotheses and observing outcomes. The open-source codebase enables advanced students to implement additional algorithms or modify existing ones.

**Assessment Activities:** The platform can support assessment through observation-based questions (“Which algorithm explores fewer nodes?”) or prediction tasks (“Which maze structure will favor DFS?”).

**Research Projects:** Advanced students can extend the platform with new algorithms, alternative heuristics, or enhanced analysis tools, providing authentic research experiences.

## VII. Discussion

### *A. Limitations and Constraints*

While the platform successfully addresses many educational challenges, several limitations warrant discussion:

**Grid-Based Representation:** The system uses discrete grid-based mazes, which may not generalize to continuous-space pathfinding problems common in robotics and game development. However, grid-based representation simplifies visualization and aligns with introductory algorithm education.

**Single-Threaded Execution:** Algorithm execution occurs on the main thread, limiting performance for very large grids. Multi-threaded execution could improve performance but would complicate visualization synchronization and state management.

**Two-Dimensional Constraint:** The platform supports only 2D mazes. While this suffices for educational purposes, extension to 3D would enable exploration of volumetric pathfinding and more complex spatial reasoning.

**No Persistence Layer:** The system lacks maze saving/loading functionality. Adding persistence would enable sharing of interesting maze configurations and reproducible benchmarking.

**Limited Heuristic Options:** A* uses only Manhattan distance. Supporting alternative heuristics (Euclidean, Chebyshev, custom) would enable exploration of heuristic design and its impact on performance.

**Window Resizing Challenges:** Dynamic window resizing during algorithm execution can cause state inconsistencies. The current implementation includes safeguards but recommends avoiding resizing during generation or solving.

### *B. Future Directions*

Several extensions would enhance the platform’s educational and research value:

**Additional Algorithms:** Implementing Wilson’s algorithm [9] for uniform spanning tree generation, Eller’s algorithm for row-by-row maze construction, and Jump Point Search [7] for optimized grid pathfinding would broaden algorithm coverage.

**3D Visualization:** Extending to three dimensions would enable exploration of volumetric pathfinding, relevant to drone navigation and 3D game development. This would require significant rendering system redesign but would provide unique educational value.

**Web Deployment:** Porting to web technologies (JavaScript, WebGL) would eliminate installation barriers and enable broader adoption. Web deployment would also facilitate integration with learning management systems.

**LMS Integration:** Developing SCORM-compliant packages or LTI integration would enable seamless incorporation into institutional learning management systems, supporting grade tracking and assignment submission.

**Advanced Analysis Tools:** Adding heat maps showing exploration density, path comparison overlays, and statistical analysis across multiple runs would support deeper investigation of algorithmic behavior.

**Collaborative Features:** Multi-user support would enable collaborative exploration, where students can share mazes, compare solutions, and discuss algorithmic trade-offs in real-time.

**Mobile Support:** Adapting the interface for tablets and smartphones would enable learning on diverse devices, though touch-based controls would require careful design to maintain usability.

### *C. Broader Implications*

This work demonstrates the value of integrated visualization platforms for algorithm education. The combination of real-time visualization, interactive control, and performance metrics addresses multiple learning modalities simultaneously. The open-source nature enables community contribution, potentially leading to a rich ecosystem of educational algorithm visualizations.

The platform's modular architecture serves as a template for similar educational tools. The separation of algorithm logic, visualization, and performance analysis enables independent enhancement of each component. This design pattern could be applied to other algorithm domains (sorting, graph algorithms, dynamic programming) to create a comprehensive algorithm education toolkit.

## VIII. Conclusion

This paper presents a comprehensive interactive platform for maze generation and pathfinding algorithm education, addressing critical gaps in algorithm visualization tools. The system integrates three generation algorithms (Recursive Backtracking, Prim's, Kruskal's) and three pathfinding algorithms (DFS, BFS, A*) with real-time visualization and performance analysis.

Empirical evaluation highlights the distinction between algorithmic complexity and interactive visualization performance. A* Search empirically demonstrates superior node exploration efficiency, reducing explored nodes by 75% compared to BFS and 67% compared to DFS while guaranteeing optimality. Generation phase metrics illustrate the impact of intentional visualization design: full step-by-step rendering of Recursive Backtracking for pedagogical depth versus batched operations in Prim's and Kruskal's for responsiveness.

The platform demonstrates that effective algorithm visualization requires balancing strict step-by-step fidelity with user experience. The system maintains robust resource management with memory usage consistently below 0.06 MB for standard grids, ensuring a smooth 60 FPS interactive experience across varied hardware.

The platform's educational design reflects established principles from educational technology research, providing immediate feedback, supporting comparative analysis, and developing performance intuition. The modular architecture enables future extensions, while the open-source implementation facilitates community contribution and institutional adoption.

Future work will focus on expanding algorithm coverage, implementing 3D visualization, and developing web-based deployment for broader accessibility. The platform demonstrates that carefully designed visualization tools can significantly enhance algorithm education by making abstract concepts concrete, enabling comparative analysis, and differentiating between algorithmic efficiency and implementation design.

## IX. Code Availability

The complete source code is publicly available under the MIT License.

**Repository:** https://github.com/binayakbartaula11/maze-runner-sim

**Included resources:** Python implementation with documentation, installation guide (`requirements.txt`), architecture diagrams, benchmarking scripts, educational materials, and enhanced version (`v2_enhanced/`).

## References

[1] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness," *Journal of Visual Languages & Computing*, vol. 13, no. 3, pp. 259-290, 2002.

[2] T. L. Naps et al., "Exploring the role of visualization and engagement in computer science education," *ACM SIGCSE Bulletin*, vol. 35, no. 2, pp. 131-152, 2003.

[3] P. J. Guo, "Online Python tutor: embeddable web-based program visualization for CS education," *Proceedings of the ACM Technical Symposium on Computer Science Education*, pp. 579-584, 2013.

[4] J. Sorva, V. Karavirta, and L. Malmi, "A review of generic program visualization systems for introductory programming education," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 1-64, 2013.

[5] 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*, vol. 4, no. 2, pp. 100-107, 1968.

[6] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *Journal of the ACM*, vol. 32, no. 3, pp. 505-536, 1985.

[7] D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid maps," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 25, no. 1, pp. 1114-1119, 2011.

[8] S. Rabin, "A* aesthetic optimizations," in *Game Programming Gems*, Charles River Media, pp. 264-271, 2000.

[9] D. B. Wilson, "Generating random spanning trees more quickly than the cover time," *Proceedings of the ACM Symposium on Theory of Computing*, pp. 296-303, 1996.

[10] J. Buck, "Novel algorithms for the generation of mazes," *Proceedings of the International Conference on Computer Games*, pp. 161-167, 2004.

[11] J. Nielsen, *Usability Engineering*, Morgan Kaufmann, 1994.

[12] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, *Software Visualization: Programming as a Multimedia Experience*, MIT Press, 1998.