Final Report:

Jump Point Search Pathfinding in 4-connected Grids

Team 66 - L1

April 27, 2025

1 Background and Motivation

Title: Jump Point Search Pathfinding in 4-connected Grids
Author(s): Johannes Baum
Conference: 2025 Research Paper (available on arXiv)

This paper introduces **JPS4**, a novel pathfinding algorithm that has been tailored for 4-connected grid maps. Finding the shortest path between two points is fundamental in fields like video game development, robotics, and navigation systems. Traditionally, the Jump Point Search algorithm (often referred to as JPS8) was designed to work with 8-directional movement. However, JPS4 refines this idea by focusing solely on the four cardinal directions (up, down, left, and right), useful for grid-based scenarios. JPS4 significantly outperform the A* algorithm in environments with high obstacle density—although, in very open spaces, A* may still have an edge.

We found this paper highly relevant due to our interest in algorithm visualization and gaming. The enhancements presented in JPS4 are directly applicable to real-world scenarios. By implementing these ideas into our project, we can gain a deep understanding of path-finding strategies in applications like game level navigation and autonomous robotics. Furthermore, the improvements over A* in terms of reducing computational overhead are compelling to explore further.

The GitHub repository provides a Rust-based implementation of a grid-based pathfinding algorithm. It supports both 4-neighborhood (restricting diagonal moves) and 8-neighborhood grids, making it adaptable for us. The combination of a solid theoretical foundation from the referenced paper and practical example code in the repository ensures that implementing JPS4

is feasible. Beyond implementation, we will benchmark JPS4 against A* to compare their efficiency in different grid-based pathfinding scenarios.

Additionally, we can use Python's libraries like Tkinter to develop the snake game, demonstrating the real-world applications of this algorithm.

2 Algorithm Overview

Problem and Contribution

Traditional algorithms like A* often waste time by expanding many unnecessary nodes in dense grid maps. Although JPS8 improves efficiency in 8-connected grids, its techniques do not directly translate to 4-connected grids, where movement is limited to cardinal directions. This is where **JPS4** comes in.

JPS4 adapts jump point search to 4-connected grids by:

- Using a horizontal-first canonical ordering to eliminate redundant paths.
- Pruning non-essential neighbors and only considering *forced neighbors* when obstacles force a deviation.
- Introducing jump points at key obstacle corners, allowing the search to bypass large sections of nodes.

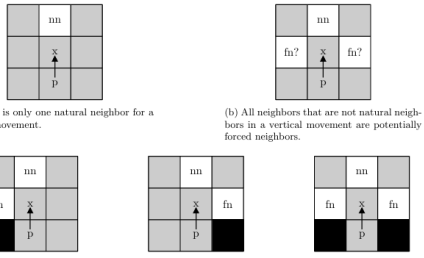
These improvements result in significantly faster pathfinding in cluttered environments such as video game maps and robotics navigation.

Canonical Ordering and Neighbor Pruning

By favoring horizontal moves, the algorithm defines a unique, canonical path. At each node, it considers only:

- The **natural neighbor** (the node directly in the direction of travel).
- **Forced neighbors** that arise when an obstacle forces a deviation.

This selective exploration reduces the search space drastically and is depicted in Figure 1. Notice how the natural neighbor is maintained, while forced neighbors are added only when obstacles are present.



(a) There is only one natural neighbor for a vertical movement.

(b) All neighbors that are not natural neighbors in a vertical movement are potentially forced neighbors.

(c) Natural and forced neighbors for vertical movements in case of obstacles

Figure 2: Natural and forced neighbors for vertical movements (black square: obstacle, p: parent node, x: current node, nn: natural neighbor, fn?: potentially forced neighbor, fn: forced neighbor)

Figure 1: Neighbor pruning in action. The algorithm retains the natural neighbor while adding forced neighbors when obstacles block the natural path.

Jump Points

When an obstacle blocks the natural path, the algorithm creates a **jump point** at the obstacle's corner. This jump point resets the search direction and allows the algorithm to "jump" over multiple nodes instead of expanding each one. Figure 2 shows a typical jump point scenario, helping the search resume efficiently after encountering an obstacle.

How It All Comes Together

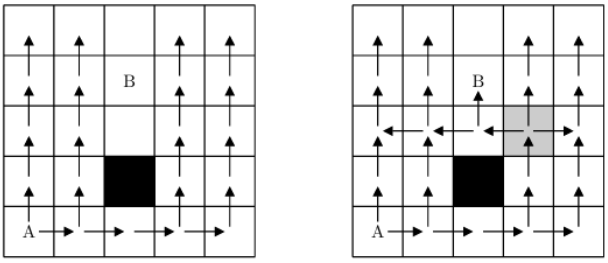
The `jump()` function continues moving in the current direction until one of the following occurs:

- The goal is reached.
- A forced neighbor is encountered.
- The direction of movement changes.

This mechanism minimizes unnecessary work and leads to optimal paths with far fewer node expansions.

3 Implementation Summary

Our implementation of the JPS4 algorithm is functional. It supports 4-connected grids with cardinal movements, featuring canonical ordering, neighbor pruning, jump point identification, and path reconstruction. We also



(a) Our canonical ordering misses the path from A to B because of an obstacle.

(b) The introduction of a jump point that resets the canonical ordering fixes the search space such that target B is now reachable again.

Figure 2: When an obstacle is encountered, a jump point is introduced at its corner, allowing the algorithm bypass unnecessary nodes and resume searching .

implemented a fallback A* algorithm within JPS4 to ensure 100% reliability in cases where JPS4 might fail due to complex grid configurations. No parts of the algorithm were omitted.

We ran multiple tests on grids of different sizes and densities to compare the performance of the A* Algorithm vs JPS4. Densities here refer to how many nodes on any particular grid were blocked off and could not be visited. Furthermore, we implemented a snake game using Tkinter and integrated JPS4 to navigate a `20 x 20` grid, demonstrating real-time pathfinding around obstacles to reach food.

4 Evaluation

4.1 Correctness Testing

We followed the exact implementation given in the paper and validated its correctness via:

1. Manual test cases by printing the grid states with paths (S for start, G for goal, * for path, # for obstacles), as shown in the example usage of both `a_star.py` and `jump_point_search.py`.

2. Randomly generated benchmark datasets with varying grid sizes and densities to compare JPS4 with baseline A*. Full results are in the `tests` folder.

3. In-game validation via `validate_path()`. Paths were visualized using the Python library Tkinter.

4.2 Complexity & Runtime Analysis

Empirical Estimate of Performance:

Using the benchmarking code, we tested on many grids of various sizes (`50 x 50` to `500 x 500`) and compared the JPS4 implementation with A* on the same grids, we notice A* outperforms our implementation of JPS4 in almost all cases.

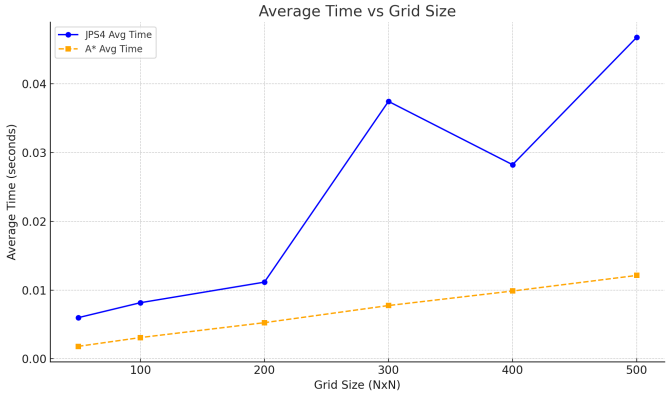


Figure 3: Comparing A* with JPS4

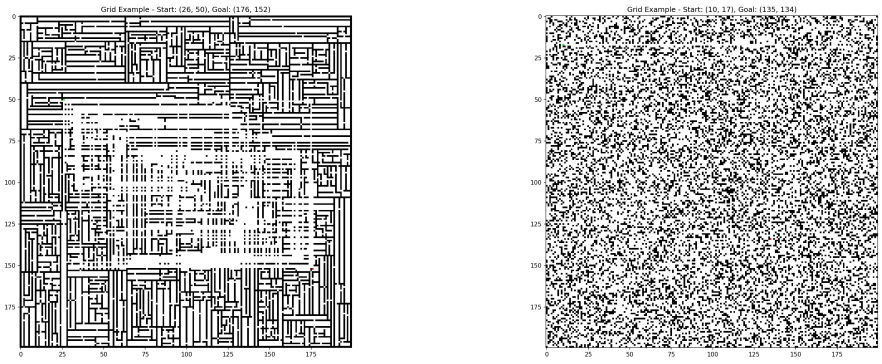
We think this could be for several reasons, like:

1. **Cache Overhead:** Frequent cache checks in our implementation slowed down the algorithm.

2. **Recursive calls** could have been much more costly than we anticipated in functions like `check_perpendicular_jump()`.

4.3 Baseline or Comparative Evaluation

Through comparing JPS4 against our A* implementation, we noticed that both algorithms maintain a 100% success rate across all grid sizes, showing no difference in reliability, but JPS4 has higher execution times than A* in almost all cases. Furthermore, JPS4 expands significantly more nodes than A*, peaking around 900 nodes, while A* expands fewer nodes, reaching around 600 at 500 size.



(a) Example of Maze-like structure (b) Example of Grid-like structure

Figure 4: Examples to illustrate the difference between grids and mazes

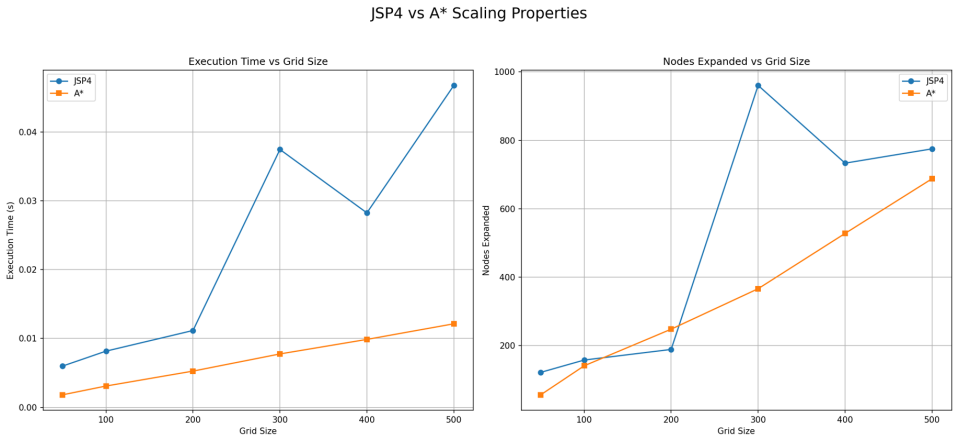


Figure 5: Comparison between JPS4 and A* on large grid size

Overall, A* scales better than JSP4 in terms of execution time and node expansion, though both are equally reliable in success rate.

Similarly, when we compared both algorithms in maze-like environments A* performed much better overall, as is evident in Figure 6. Search Space here refers to the amount of nodes visited.

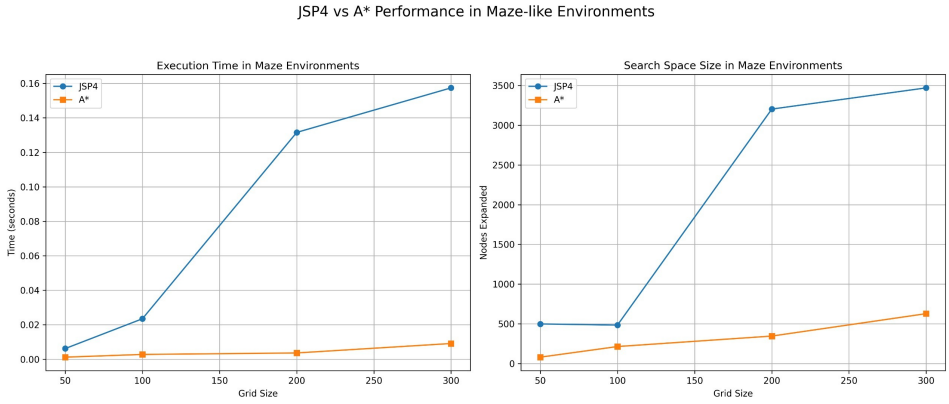


Figure 6: Comparison between A* and JPS4 in Maze Like environments

5 Enhancements

As part of our additional work for our project:

- We tested JPS4 on many different grid benchmarks and maze-specific tests. These detailed reports and visualizations highlighted the performance of our implementation of JPS4 in Python.
- We tested JPS4 in a dynamic `20 x 20` Maze with easy and hard layouts by implementing it in a Snake Game.

6 Conclusion

6.1 Challenges & Solutions

Our primary challenge was converting the JPS4 paper's code into Python. JPS4 performed much worse than the paper's reported speedup. We suspect issues in language-specific optimizations during conversion from Rust to Python, the exact solution remains unclear.

6.2 Learning Outcomes

Through this project, we gained hands-on experience implementing and comparing JPS4 and A* on 4-connected grids, which clarified how algorithmic choices affect performance in open versus cluttered maps. We understood optimization techniques like canonical ordering, neighbor pruning,

forced-neighbor detection, and jump-point identification etc. learned to systematically benchmark runtime and node-expansion across varied grid sizes and obstacle densities.

By observing the disparity between the Rust-reported speedups and our Python results, we identified language-specific overheads (e.g. recursion and cache behavior) as critical factors. This process sharpened our debugging skills in diagnosing bottlenecks and validating correctness via tests and visualizations. We also deepened our understanding of grid data structures and spatial problem-solving, and learned to choose between A* and JPS4 based on environment characteristics, balancing execution time, memory overhead, and implementation complexity.

6.3 Future Work

In terms of future work, we could attempt to optimize jump-point detection by incorporating precomputed caching of obstacle corner information, which should reduce runtime overhead during searches.

To adapt more dynamically to different map types, we could try to implement adaptive pruning thresholds that adjust based on local grid density. For very large-scale grids, we could use parallel processing techniques e.g. splitting the grid into regions processed concurrently which could lead to much better results.

References

References

[1] J. Baum, "Jump Point Search Pathfinding in 4-connected Grids," *arXiv preprint arXiv:2501.14816v1*, Jan. 28, 2025.

[2] T. B. van der Woude, "grid_pathfinding," GitHub, 2025. [Online]. Available: `https://github.com/tbvanderwoude/grid_pathfinding`. [Accessed: Apr. 27, 2025].