Progress Report

Jump Point Search Pathfinding in 4 connected Grids

Team 66 - L1

April 20, 2025

1 Implementation Summary

Our implementation of the JPS4 (Jump Point Search for 4-connected grids) algorithm, based on Johannes Baum's paper, is functional. It supports 4-connected grids with cardinal movements, featuring canonical ordering, neighbor pruning, jump point identification, and path reconstruction. We also 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 path-finding around obstacles to reach food.

We found this paper relevant due to our interest in algorithms in gaming. We gained a deep understanding of path-finding strategies in applications like game level navigation.

2 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.

3 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 got the following results:

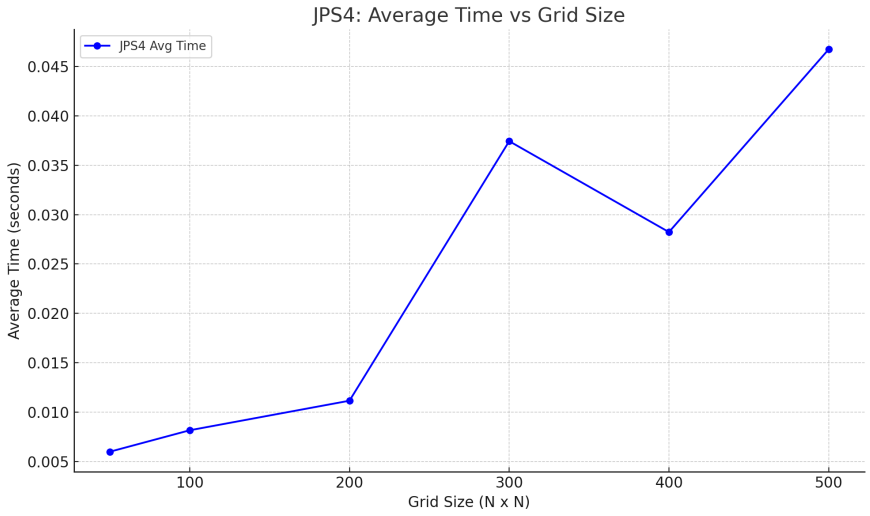

Figure 1: Runtime Analysis of JPS4

Comparing the JPS4 implementation with A* on the same grids, we notice A* outperforms our implementation of JPS4 in almost all cases.

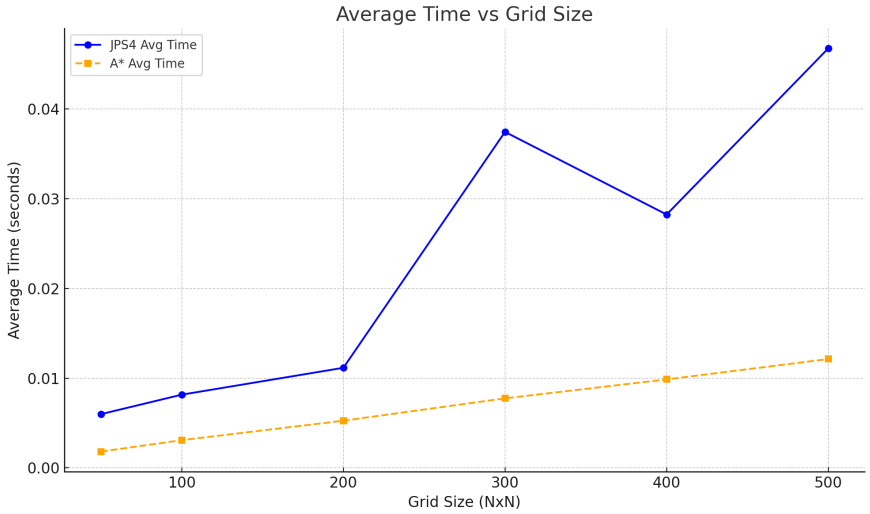

Figure 2: Comparing A* with JPS4

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 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.

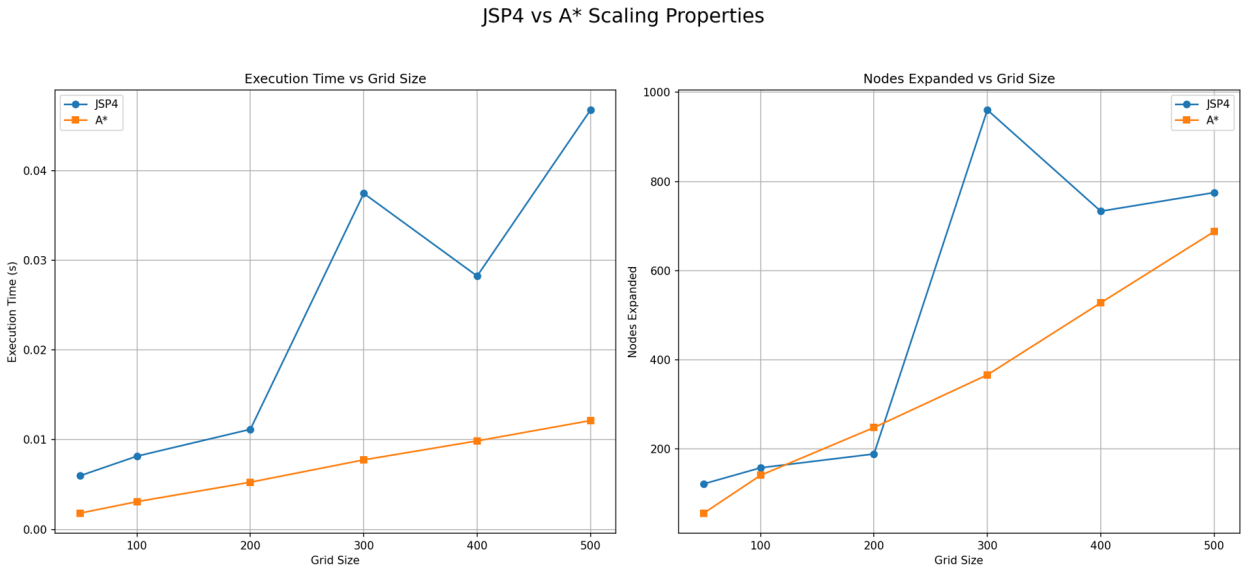

Figure 3: Comparison between JPS4 and A* on large grid size

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

5 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 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` grid with easy and hard layouts by implementing it in a Snake Game.

References

[1] Johannes Baum, *Jump Point Search Pathfinding in 4-connected Grids*, arXiv:2501.14816v1, January 28, 2025.