Technical Summary:

Jump Point Search Pathfinding in 4-connected Grids

(Summary of Johannes Baum's Paper)

Team 66 - L1

April 8, 2025

Overview

This document summarizes the key ideas and challenges in implementing Johannes Baum's paper, *Jump Point Search Pathfinding in 4-connected Grids*. Our goal is to understand how JPS4 works, why it is needed, and what challenges we might face during its implementation.

1 Problem and Contribution

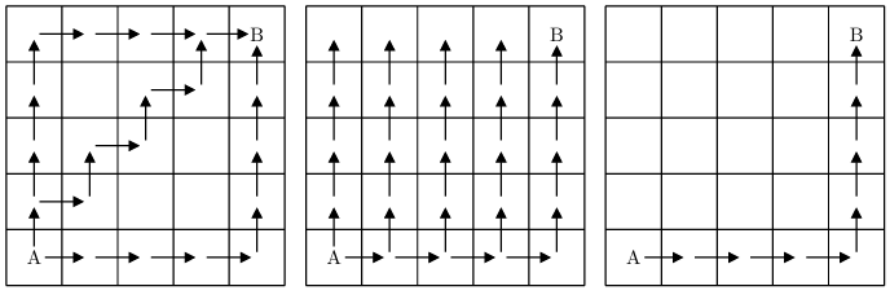
Traditional algorithms like A* often waste time by expanding many unnecessary nodes in dense grid maps. Although JPS8 improves efficiency in 8-connected grids, its techniques do not directly translate to 4-connected grids, where movement is limited to four cardinal directions. This is where **JPS4**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

JPS4 adapts the 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.

Before we dive deeper, Figure 1 from the paper illustrates how multiple optimal paths in an obstacle-free grid are pruned into one unique, canonical path.



(a) Some symmetric paths from A to B

(b) Applying our canonical ordering converts the search space from a graph into a tree by eliminating redundant paths.

(c) The search space that results from applying our canonical ordering (figure 1b) yields only one path from A to B.

Figure 1: Elimination of symmetric paths in obstacle-free maps via canonical ordering.

Figure 1: Canonical ordering in an obstacle-free grid. Dashed lines indicate multiple optimal paths, which are pruned to a single canonical path using a horizontal-first strategy.

2 How the Algorithm Works

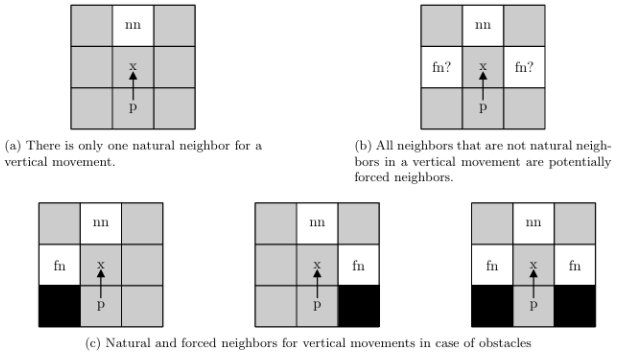
JPS4 builds on a simple but powerful idea: only explore what's necessary.

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 is depicted in Figure 2. 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.

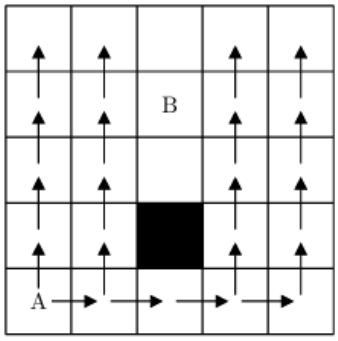
(c) Natural and forced neighbors for vertical movements in case of obstacles

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)

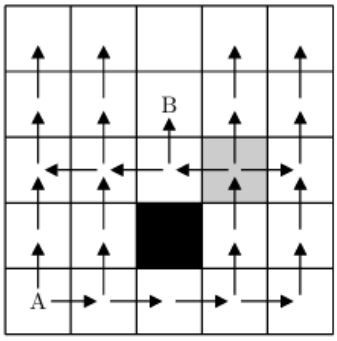
Figure 2: Neighbor pruning in action. The algorithm retains the natural neighbor while adding forced neighbors when obstacles block the natural path.

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 3 shows a typical jump point scenario, helping the search resume efficiently after encountering an obstacle.



(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.

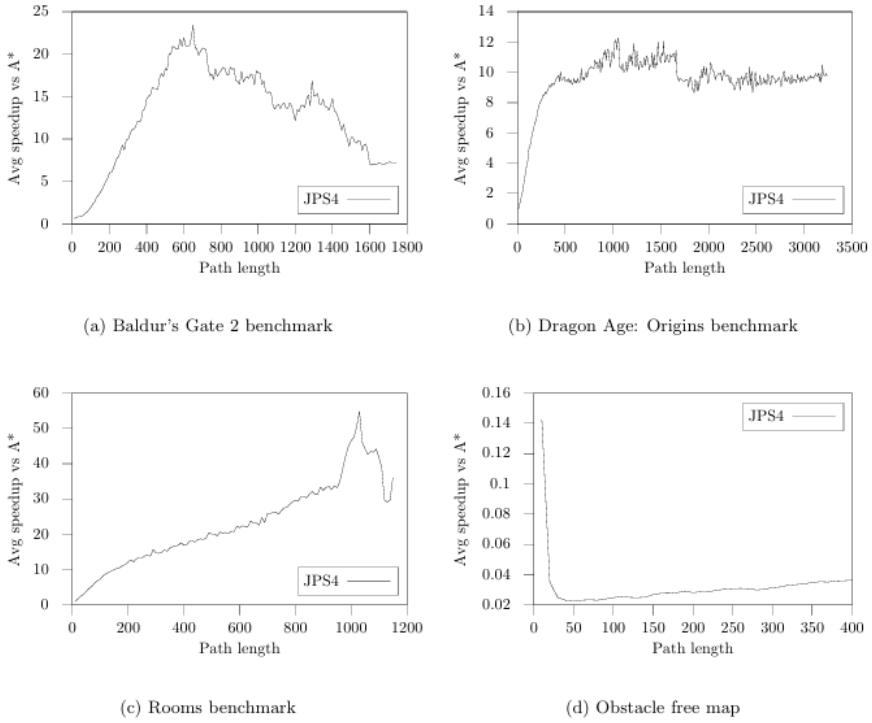
Figure 3: When an obstacle is encountered, a jump point is introduced at its corner, allowing the algorithm to bypass unnecessary nodes and resume the search efficiently.

How It All Comes Together

The `jump()` function is central to the algorithm. It 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 (e.g., from vertical to horizontal).

This mechanism minimizes unnecessary work and leads to optimal paths with far fewer node expansions. Figure 4 provides benchmark results from the paper that demonstrate the efficiency improvements of JPS4 over A* in various map environments.



(a) Baldur's Gate 2 benchmark

(b) Dragon Age: Origins benchmark

(c) Rooms benchmark

(d) Obstacle free map

Figure 4: Average speedup of JPS4 compared to A* grouped by path length.

Figure 4: Benchmark results illustrating the speedup of JPS4 over A* across different map types, particularly in obstacle-dense environments.

3 Comparison with Other Methods

Compared to A*, JPS4 avoids unnecessary node expansions, resulting in a smaller open list and faster performance in cluttered maps. Unlike JPS8—which is designed for 8-connected grids—JPS4 refines these ideas specifically for grids limited to cardinal directions, making it more suitable for certain applications.

4 Data Structures and Techniques

JPS4 leverages several key data structures and techniques:

- **Grid Representation:** A uniform-cost grid where each cell is either free or blocked.
- **Open List:** Managed using a priority queue (e.g., binary heap) to quickly select the lowest-cost node.
- **Pruning Mechanism:** Custom logic based on canonical ordering significantly reduces the search space.

These structures are optimized to work together efficiently and support the selective expansion approach of the algorithm.

5 Implementation Outlook

Although the theoretical design of JPS4 is robust, its practical implementation presents several challenges:

- **Recursion Overhead:** The recursive nature of the `jump()` function might lead to deep call stacks. We could mitigate this by setting a recursion limit or using an iterative approach.
- **Edge Case Handling:** Special attention is needed for narrow corridors and complex obstacle configurations to ensure the algorithm finds optimal paths.
- **Memory Efficiency:** Efficient use of data structures for the grid and the open list is essential, especially when dealing with large maps.

Early strategies include optimizing priority queue operations and employing visualization tools to debug and verify the algorithm's behavior.

Conclusion

JPS4 offers a clever adaptation of jump point search for 4-connected grids by using canonical ordering and selective neighbor expansion. This results in significant performance gains in cluttered environments. While challenges such as recursion depth and edge-case handling remain, the potential improvements in efficiency make JPS4 a promising approach for applications in video games and robotics.