DTU Compute
Department of Applied Mathematics and Computer Science

Localized Conflict-Resolution in Multi-Agent Pathfinding

An Enhancement to Conflict-Based Search

Master Thesis

Localized Conflict-Resolution in Multi-Agent Pathfinding
An Enhancement to Conflict-Based Search

Master Thesis
Oct, 2024

By
Taiquan Sui

Supervisor
Thomas Bolander

Cover photo: Vibeke Hempler, 2012
Published by: DTU, Department of Applied Mathematics and Computer Science, Building 324, 2800 Kgs. Lyngby Denmark
www.compute.dtu.dk
ISSN: [0000-0000] (electronic version)
ISBN: [000-00-0000-000-0] (electronic version)

ISSN: [0000-0000] (printed version)
ISBN: [000-00-0000-000-0] (printed version)

Approval

This thesis has been prepared over a period of five and a half months at the Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU), in partial fulfillment of the requirements for the Master of Science in Engineering (MSc Eng.) degree.

It is assumed that the reader has a basic knowledge in the areas of statistics.

Taiquan Sui - s223131

.....

Signature

.....

Date

Abstract

Multi-Agent Pathfinding (MAPF) is a fundamental problem in robotics and artificial intelligence, where multiple agents must navigate through a shared environment without colliding. Conflict-Based Search (CBS) is a state-of-the-art approach for solving MAPF optimally, but it suffers from scalability issues as the number of agents and the complexity of the environment increase, largely due to the exponential growth of the constraint tree (CT) used to resolve conflicts. This thesis proposes a novel enhancement to the CBS framework that mitigates this issue by adopting a localized conflict resolution strategy. Inspired by the efficiency of Jump Point Search (JPS), the proposed method resolves conflicts locally, within a limited region around the conflict point, rather than requiring global replanning. The proposed method is implemented and evaluated on established MAPF benchmarks, and its performance is compared against basic CBS with bypass improvement. The results demonstrate that the localized conflict resolution strategy can enhance the scalability and performance of CBS, offering a promising alternative for complex, real-world multi-agent pathfinding scenarios.

Acknowledgements

I would like to express my gratitude to my parents for their support throughout my master's degree.

I would also like to thank my supervisor, Thomas Bolander, for allowing me the freedom to choose my research topic and explore the problems that interest me.

Additionally, I would like to acknowledge the creators of the pathfinding visualization website https://qiao.github.io/PathFinding.js/visual/ for their valuable tool.

Contents

Preface ii
Abstract iii
Acknowledgements iv

1 Introduction **1**

2 Problem definition and terminology **4**
2.1 Problem input 4
2.2 Action 4
2.3 A Sequence of Actions 4
2.4 Constraint 5
2.5 Conflict 5
2.6 Solution 5
2.7 Cost Function 5
2.8 Distributed vs. Centralized 6

3 Background and Related Work **7**
3.1 A* 7
3.2 Space-Time A* 7
3.3 Safe Interval Path Planning (SIPP) 8
3.4 Jump point search 9
3.5 Conflict based search(CBS) 12
3.6 ICBS 14
3.7 ICBS-H 17
3.8 CBSH2 18
3.9 Disjoint Splitting (DS) 18
3.10 ECBS, BCBS, GCBS 19

4 Continuous-Search Jump Point Search **21**
4.1 Definitions 21
4.2 Continuous Search 21
4.3 JPSPath - Jump Points 22
4.4 JPSPath - Possible Interval 22
4.5 Theoretical Analysis of Continuous Search in JPS 24
4.6 Different Choice for Possible Intervals 25

5 JPSCBS **28**
5.1 Definitions 28
5.2 High Level 28
5.3 Finding Bypass 30
5.4 Resolve a conflict 31
5.5 Low Level - JPS 31
5.6 Low Level - A* variants 32
5.7 JPSCBS - Suboptimal 32

6 Implementation **34**
6.1 Possible Interval 34

6.2 Tie Breaking for A* Node 35

7 Experimental Setup **37**
7.1 Maps and Scenarios 37
7.2 Benchmark Setup 37
7.3 Evaluation Methodology 38

8 Result **39**
8.1 Success Rate 39
8.2 Expanded Nodes 40
8.3 Run Time 41

9 Conclusion **42**

10 Future Work **43**

Bibliography **45**

A Appendix **46**

1 Introduction

Path planning plays a crucial role in various aspects of daily life, including mobile navigation, autonomous driving, and character movement in video games. The goal of path planning is to determine an optimal route from a start position to a destination in a complex environment for one or more agents, such as robots, vehicles, or game characters. Traditionally, when only a single agent is involved, the problem is referred to as **single-agent pathfinding**. However, when multiple agents navigate the same environment simultaneously, the problem becomes **multi-agent pathfinding (MAPF)**.

Single-Agent Pathfinding

In single-agent pathfinding, the objective is to find the shortest or most optimal path from the start to the goal position. The A* algorithm, introduced by Hart et al., is one of the most widely used algorithms for this task. A* efficiently finds optimal paths by combining the actual cost from the start node to the current node with a heuristic estimate of the cost from the current node to the goal. However, in grid-based environments, where numerous symmetric paths with equal costs exist, A* can suffer from inefficiencies, especially in large-scale or open areas.

To address this limitation, Harabor and Grastien proposed Jump Point Search (JPS), which exploits the structural properties of grids. JPS employs pruning and jump-based expansion strategies, selectively extending only ”turning points” that are essential for the optimal path. This significantly reduces the number of expanded nodes and improves search efficiency while preserving A*’s optimality. Due to these advantages, JPS has been widely adopted in video games and robotic navigation applications.

Multi-Agent Pathfinding

In multi-agent scenarios, each agent must not only compute an optimal path but also ensure that its path does not conflict with other agents’ movements. One of the most effective and optimal approaches in MAPF is Conflict-Based Search (CBS). CBS decomposes the multi-agent problem into a series of single-agent pathfinding problems. Initially, it computes individual paths independently and then iteratively detects and resolves conflicts between agents to ensure a feasible, collision-free solution[1]. This two-level search framework allows CBS to guarantee optimality while effectively handling large-scale MAPF instances.

Several enhancements to CBS have been proposed to improve its efficiency. Notable examples include Improved CBS (ICBS) [2], which refines conflict resolution strategies, and CBS with Heuristics (CBSH) [3], which integrates heuristic functions to guide high-level search. While these improvements enhance CBS’s computational efficiency and robustness, the algorithm remains computationally expensive, particularly as the number of agents increases or the environment becomes more complex.

To address the scalability challenges of optimal MAPF solvers, researchers have developed suboptimal approaches that trade off optimality for efficiency. One such method is Enhanced CBS (ECBS) [4], which introduces bounded suboptimality to accelerate computation. Additionally, machine learning-based distributed planning approaches, such as the PRIMAL series [5], leverage reinforcement learning to optimize path planning in large-scale dynamic environments. These advancements highlight the ongoing efforts

to balance optimality, computational efficiency, and practical applicability in multi-agent pathfinding.

Main Contributions and Innovations

This thesis builds upon the efficiency of Jump Point Search (JPS) in single-agent path planning and the conflict resolution mechanisms of Conflict-Based Search (CBS). We propose a local conflict resolution strategy, which leverages JPS's pruning advantages in local regions to resolve conflicts efficiently without requiring global re-planning. This approach aims to enhance the overall efficiency of CBS.

The proposed method offers the following key advantages:

- **Efficiency**: By focusing on localized conflict resolution instead of global replanning, our approach significantly reduces the number of expanded nodes, thereby accelerating the overall planning process.
- **Scalability**: The local conflict resolution mechanism can be seamlessly integrated into existing CBS frameworks, ensuring optimal or near-optimal solutions while significantly improving planning speed. Moreover, various CBS enhancements can be adapted with minor modifications to work with our method.
- **Theoretical and Practical Contributions**: We provide a theoretical analysis of the proposed approach, demonstrating that it maintains optimality under certain conditions. Additionally, extensive experiments on standard grid-based maps and real-world scenarios validate its effectiveness in solving large-scale MAPF problems.

Methodology Overview and Implementation

In this thesis, we first conduct an in-depth study of single-agent path planning techniques, such as A* and its optimized variant JPS, as well as conflict detection and resolution mechanisms in multi-agent path planning. By integrating the advantages of both approaches, our proposed local conflict resolution strategy consists of the following key steps:

- **Localized Path Adjustment**: Utilizing the pruning and jump mechanisms of JPS, we generate alternative paths within the conflict region without requiring full replanning for all agents.
- **Global Consistency Assurance**: While resolving conflicts locally, the generated paths are validated to ensure that the overall solution remains conflict-free and either optimal or near-optimal.

This methodology enables efficient computation by focusing only on conflict-prone areas, reducing computational overhead and improving search efficiency, especially in complex or large-scale environments with obstacles.

Thesis Structure

To provide a clear and structured presentation of the research background, methodology, and experimental results, this thesis is organized as follows:

- **Chapter 2: Problem Definition and Terminology**
  This chapter defines the fundamental concepts of multi-agent pathfinding (MAPF), introduces commonly used terminologies, and categorizes different types of conflicts (e.g., vertex conflicts, edge conflicts), establishing a foundation for subsequent discussions.

- **Chapter 3: Background and Related Work**
  A comprehensive review of classical single-agent pathfinding algorithms, such as A* and Jump Point Search (JPS), is provided. Additionally, this chapter discusses widely used conflict resolution strategies in MAPF, including Conflict-Based Search (CBS) and its improved variants.
- **Chapter 4: Continuous-Search Jump Point Search**
  In this chapter, we extend the JPS algorithm to support continued search and modify the return information of its path search results, enabling it to be integrated with CBS for local conflict resolution.
- **Chapter 5: Local Conflict Resolution Strategy**
  This chapter presents the core contribution of this thesis— a novel local conflict resolution strategy. It details the underlying principles, key algorithms, and implementation techniques, demonstrating how path adjustments can be efficiently performed within local regions to resolve conflicts while ensuring optimal or near-optimal global solutions.
- **Chapter 6 - 8: Implementation and Experiments**
  The system implementation is described, followed by a comprehensive evaluation of the proposed method on standard grid-based benchmarks and real-world scenarios. The experimental results indicate that, compared to conventional CBS, the local conflict resolution approach significantly increases success rate, reduces computation time in most cases while maintaining high solution quality.
- **Chapter 9 - 10: Conclusion and Future Work**
  This chapter summarizes the key findings of the research, discusses the advantages and limitations of the proposed approach, and outlines potential directions for future improvements.

In summary, this research builds upon the foundations of JPS and CBS by integrating a local conflict resolution mechanism, effectively addressing efficiency bottlenecks in multi-agent pathfinding. The proposed method contributes to both theoretical advancements in path planning and practical applications in multi-robot coordination under complex environments.

2 Problem definition and terminology

There exist many variants of the Multi-Agent Pathfinding (MAPF) problem. In this work, we define the problem and subsequently describe algorithms within the framework of a widely-used general variant of the problem [6, 7, 1]. Similar to Conflict-Based Search (CBS), this variant is designed to be as general as possible, ensuring the broad applicability of the algorithm, while encompassing various sub-variants. For clarity and consistency, several definitions and terms previously established in other works will be adopted here for convenience [8].

2.1 Problem input

The input to the multi-agent pathfinding problem (MAPF) is:

- $G = (V, E)$, where G is an undirected graph with a set of vertices V and a set of edges E. The vertices are possible locations for the agents, and the edges are the possible transitions between locations.
- $A = \{a_1, a_2, \ldots, a_k\}$, where A is a set of k agents, each uniquely labeled by a_i.
- $s : A \to V$, where $s(a_i)$ maps each agent $a_i \in A$ to a starting vertex $s(a_i) \in V$.
- $t : A \to V$, where $t(a_i)$ maps each agent $a_i \in A$ to a target vertex $t(a_i) \in V$.

Time is assumed to be discretized. At time point t_0, agent a_i is located in location $s(a_i)$. Each agent is situated in one of the graph vertices in every time step and can perform a single action.

2.2 Action

An action is formally defined as a function $a : V \to V$, such that $a(v) = v'$ indicates that if an agent is located at vertex v and executes action a, it will be at vertex v' in the subsequent time step. There are two distinct types of actions: **Wait** and **Move**.

1. *Wait*: The agent remains at its current vertex, that is, $v' = v$.
2. *Move*: The agent transitions from its current vertex v to an adjacent vertex v', meaning $v' \neq v$ and $(v, v') \in E$.

At each discrete time step, an agent can either perform a *move* action to a neighboring vertex or a *wait* action to remain at its current vertex.

2.3 A Sequence of Actions

For a sequence of actions $\pi_i = (a_1, \ldots, a_n)$ of agent i, the expression $\pi_i[x]$ denotes the location of agent i after executing first x actions of plan π_i, starting from its initial location $s(i)$. Formally, this can be written as:

$$\pi_i[x] = a_x(a_{x-1}(\ldots a_1(s(i))\ldots)).$$

A sequence of actions π_i is defined as **a single-agent plan** for agent i iff executing the entire sequence π_i from the initial position $s(i)$ results in the agent reaching its target vertex $t(i)$. Formally, this condition is satisfied if and only if $\pi_i[|\pi|] = t(i)$.

2.4 Constraint

In multi-agent pathfinding, constraints play a critical role in ensuring that the paths computed for agents avoid conflicts such as collisions, both in terms of space and time. A simple example of constrain is that when there are static or dynamic obstacles, constraints are introduced to ensure that agents do not attempt to move into an obstacle space. Another example can be precedence constraints, which are used to ensure that one agent completes its task or movement before another agent begins its task. A conflict is a case where a constraint is violated.

2.5 Conflict

The goal of MAPF solvers is to find solutions without violating constraints. A MAPF solution is valid if and only if there are no conflicts between any two agents' plans. Let π_i and π_j be two single-agent plans. In this thesis, we consider the following types of conflicts:

- **Vertex conflict**: Occurs if both agents occupy the same vertex at the same time step. Formally, a vertex conflict exists iff $\exists x$ such that $\pi_i[x] = \pi_j[x]$.
- **Following conflict**: Occurs if one agent occupies a vertex that another agent occupied in the previous time step. Formally, this conflict exists iff $\exists x$ such that $\pi_i[x+1] = \pi_j[x]$.
- **Swapping conflict**: Occurs if two agents swap locations in a single time step. Formally, this conflict exists iff $\exists x$ such that $\pi_i[x+1] = \pi_j[x]$ and $\pi_j[x+1] = \pi_i[x]$, also referred to as an edge conflict in MAPF literature.

There are other types of common conflicts:

- **Edge conflict**: Occurs if both agents traverse the same edge in the same direction at the same time. Formally, this conflict exists iff $\exists x$ such that $\pi_i[x] = \pi_j[x]$ and $\pi_i[x+1] = \pi_j[x+1]$.
- **Cycle conflict**: Occurs when multiple agents cyclically swap locations in the same time step. Formally, this conflict exists iff $\exists x$ such that $\pi_i(x+1) = \pi_{i+1}(x), \ldots, \pi_j(x+1) = \pi_i(x)$.

However, our algorithm does not explicitly consider these conflicts, as vertex conflicts imply edge conflicts, and following conflicts imply cycle conflicts.

2.6 Solution

A **solution** to the multi-agent pathfinding problem consists of a set of k single-agent plans without violating any constraint, one for each agent.

2.7 Cost Function

There may be multiple solutions to a MAPF problem. To evaluate if a solution is better than another, cost functions are introduced. The two most common functions in classical MAPF are makespan and sum of costs:

- **Makespan**: The number of time steps required for all agents to reach their targets. For a MAPF solution $\pi = \{\pi_1, \ldots, \pi_k\}$, the makespan is defined as $\max_{1 \leq i \leq k} |\pi_i|$.
- **Sum of costs**: The total time steps taken by all agents to reach their targets. The sum of costs is defined as $\sum_{1 \leq i \leq k} |\pi_i|$ and is also known as flowtime.

2.8 Distributed vs. Centralized

MAPF problems can be categorized into two groups: distributed and centralized.

Centralized MAPF

In a centralized approach to MAPF, a single entity (usually referred to as the central planner) has complete knowledge of all agents, their goals, and the environment. This central planner computes paths for all agents simultaneously, taking into account the possible conflicts (such as collisions) that might arise as agents move through the environment.

Advantages:

- **Global Optimization**: The centralized controller considers the entire state space, enabling globally optimal pathfinding.
- **Coordination**: All agents' paths are planned together, ensuring systematic conflict resolution and collision-free solutions.

Challenges:

- **Scalability**: As the number of agents increases, the state space grows exponentially, leading to high computational overhead.
- **Single point control**: Single point of failure—if the central planner fails, the entire system is affected.

Distributed MAPF

Distributed approaches decentralize pathfinding, with each agent planning its own path while coordinating with others to avoid conflicts. Agents often have only partial knowledge of the environment and resolve conflicts locally.

Advantages:

- **Scalability**: These approaches tend to scale better, as the computational workload is spread across agents, avoiding a single point of control.
- **Resilience**: Distributed systems are more resilient to individual failures, as one agent's failure does not necessarily affect the entire system.

Challenges:

- **Suboptimal Solutions**: Agents operating with partial knowledge may fail to find globally optimal paths, making coordination essential for avoiding conflicts.
- **Coordination Overhead**: Communication between agents to resolve conflicts can lead to increased overhead, particularly in systems with limited communication capabilities.

The scope of this thesis is limited to centralized approaches since it are based on conflicted based search. But the idea of solving conflict locally also can be used in distributed MAPF.

3 Background and Related Work

A key challenge in solving MAPF optimally is the combinatorial explosion of possible configurations as the number of agents increases. Traditional approaches, such as A* and its variants, perform well for single-agent pathfinding but struggle to scale for multi-agent scenarios [9]. To address these scalability issues, specialized MAPF algorithms have been developed. Among them, Conflict-Based Search (CBS) [1] is recognized as one of the most efficient optimal MAPF solvers.

CBS divides the MAPF problem into two levels. At the low level, individual agent paths are computed using single-agent search algorithms like A*. The high-level search detects conflicts between agents, such as two agents attempting to occupy the same location simultaneously, and resolves these conflicts by introducing constraints. While this method guarantees optimality, it can experience significant slowdowns in environments with high agent density, as numerous conflicts lead to an exponentially growing constraint tree.

This thesis aims to optimize the speed of CBS in grid map by resolving conflicts locally at the low level, avoiding the need for repeating re-searching with single-agent algorithms. Therefore, this section introduces relevant work on single-agent path finding and the conflicted based search.

3.1 A*

The single-agent pathfinding problem involves finding a path between two vertices in a graph. A* is one of the most widely used algorithms for this. It evaluates the total cost function at each node, defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost from the start node to n, and $h(n)$ is the heuristic estimate of the cost from n to the goal. The algorithm expands by selecting the node with the smallest f(n) to ensure that it can find a more optimized path. The search efficiency and path quality of A* depend heavily on the design of the heuristic function h(n). When h(n) is an accurate estimate of the true cost, A can quickly find the optimal solution. A* also guarantees the completeness and optimality of the solution as long as the heuristic function is consistent or admissible-it never overestimates the true cost [10]. However, A* may become inefficient in large grids due to exhaustive node exploration, especially in sparse or structured environments.

3.2 Space-Time A*

Cooperative Pathfinding is a method for coordinating multi-agent path planning to avoid collisions and conflicts that arise in traditional A* searches applied to multi-agent scenarios. In 2005, David Silver proposed *Space-Time A** as a solution[9], introducing the *temporal dimension* to improve coordination in multi-agent pathfinding.

3.2.1 Challenges of Traditional A* in Multi-Agent Environments

Standard A* is primarily designed for single-agent path planning and does not account for the paths of other moving agents, leading to dynamic conflicts:

- When multiple agents move simultaneously, A* may generate *overlapping paths* or require frequent *replanning*, resulting in inefficiencies or deadlocks (e.g., agents indefinitely avoiding each other in a narrow passage).
- Traditional A* often treats other agents as static obstacles rather than dynamically moving entities, leading to unnecessary detours or congestion.

3.2.2 Space-Time A* (STA)

*Space-Time A** extends pathfinding into a three-dimensional space (x, y, t), known as the *Space-Time Map*, which explicitly models the temporal dimension:

State Representation:

Each pathfinding state consists of not only spatial coordinates (x, y) but also a time step t, forming the state tuple (x, y, t).

Path Search:

- A* is applied in the three-dimensional (x, y, t) space to find the optimal path while ensuring no temporal conflicts with other agents.
- The goal state is defined as reaching the destination at *any feasible time step* t_{goal}, rather than at a fixed time.

3.2.3 Reservation Table for Coordination

To ensure coordinated path planning, *Space-Time A** employs a *Reservation Table*, which records planned paths to prevent conflicts:

- Each cell in the table stores whether it has been reserved by another agent at a specific time step t.
- When planning a path, A* avoids occupied cells, ensuring no time conflicts occur.
- *Head-to-Head Collision Avoidance*: Agents prevent direct head-on collisions by reserving both t and $t+1$ time steps, ensuring that two agents do not attempt to occupy the same space at different but conflicting time steps.

3.3 Safe Interval Path Planning (SIPP)

Safe Interval Path Planning (SIPP)[11] is an optimized time-dependent path search algorithm designed for navigation in dynamic environments. Compared to traditional A* search, SIPP leverages *safe intervals* to significantly reduce the search space while maintaining both optimality and completeness.

3.3.1 Background

In dynamic environments, path planning must consider not only spatial coordinates (x, y) but also the temporal dimension t, as obstacles may change over time. Traditional approaches often discretize time into fixed intervals, leading to an exponential increase in computational cost. SIPP addresses this issue by introducing *safe intervals*, which efficiently manage the temporal dimension.

3.3.2 Core Improvements

Safe Intervals:

- Instead of considering individual time steps, SIPP stores *continuous, collision-free time intervals* for each location, reducing the search space.
- Each state is defined as a tuple $(location, safe\ interval)$ rather than $(location, time\ step)$, allowing for the merging of redundant time states and decreasing computational complexity.

Search Optimization:

- During search, SIPP only expands paths that lead to the next available safe interval, eliminating the need to evaluate all possible time steps.

- The algorithm employs a *wait-and-move* strategy, allowing the agent to wait at the start of a safe interval until it can safely proceed, thereby avoiding unnecessary search expansions.

3.4 Jump point search

Jump Point Search (JPS) was introduced to improve the efficiency of A* by reducing the number of explored nodes during the search [12]. Rather than visiting every neighboring node, JPS prunes the search space by ”jumping” between critical nodes, or jump points, where the path direction changes. These points are identified using symmetry-breaking rules, allowing the algorithm to skip redundant nodes while maintaining A*’s optimality. This localized search mechanism, which focuses on critical regions, inspires the approach in this thesis to resolve conflicts in multi-agent pathfinding locally rather than globally. Since some definitions and ideas of JPS will be repeatedly used and discussed throughout this thesis, we will provide a detailed citation of some content from the original papers to comprehensively introduce and explain the definitions and concepts of JPS.

3.4.1 Notation and Terminology

- JPS works in undirected uniform-cost grid maps, where each node has at most 8 neighbors and is either traversable or not. Each straight (i.e., horizontal or vertical) move from a traversable node to one of its neighbors has a cost of 1, while diagonal moves cost $\sqrt{2}$. Moves involving non-traversable (obstacle) nodes are disallowed.
- The notation $\tilde{d}$ refers to one of the eight allowable movement directions (up, down, left, right, etc.). We write $y = x + k\tilde{d}$ when node y can be reached by taking k unit moves from node x in direction $\tilde{d}$. When $\tilde{d}$ represents a diagonal move, we denote the two straight moves at $45°$ to $\tilde{d}$ as $\tilde{d}_1$ and $\tilde{d}_2$.
- A path $\pi = \langle n_0, n_1, \ldots, n_k \rangle$ is a cycle-free ordered walk starting at node n_0 and ending at n_k. We will sometimes use the setminus operator in the context of a path: for example, $\pi \setminus x$. This means that the subtracted node x does not appear on (i.e., is not mentioned by) the path.
- The function `len` refers to the length (or cost) of a path and the function `dist` refers to the distance between two nodes on the grid, e.g., `len(`π`)` or `dist(`n_0, n_k`)` respectively.

3.4.2 Neighbour Pruning Rules

JPS reduces the search space by applying pruning rules. The objective is to identify, from each set of such neighbors (i.e., `neighbours(x)`), any nodes n that do not need to be evaluated in order to reach the goal optimally. This is achieved by comparing the length of two paths: π, which begins at node $p(x)$, visits x, and ends at n, and another path π' which also begins at $p(x)$ and ends at n without visiting x. Note that if x is the start node, $p(x)$ is null, and nothing is pruned.

There are two cases to consider, depending on whether the transition to x from its parent $p(x)$ involves a straight move or a diagonal move. Note that if x is the start node, then $p(x)$ is `null`, and nothing is pruned.

Straight Moves

For a node x, $n \in$ neighbours(x) is pruned if the following dominance condition is satisfied:

$$\text{len}\left(\langle p(x), \ldots, n\rangle \setminus x\right) \leq \text{len}\left(\langle p(x), x, n\rangle\right) \tag{1}$$

Figure3.1(a) shows an example where $p(x) = 4$ and all neighbours except $n = 5$ are pruned.

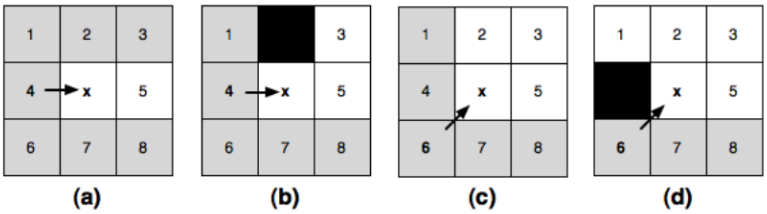


Figure 3.1: Several cases where a node x is reached from its parent p(x) by either a straight or diagonal move. When x is expanded we can prune from consideration all nodes marked grey.

Diagonal Moves

This case is similar to the pruning rules for straight moves, but the path that excludes x must be strictly dominant:

$$\text{len}\left(\langle p(x), \ldots, n\rangle \setminus x\right) < \text{len}\left(\langle p(x), x, n\rangle\right) \tag{2}$$

Figure3.1(c) illustrates this, where $p(x) = 6$ and all neighbours except $n = 2$, $n = 3$, and $n = 5$ are pruned.

Natural and Forced Neighbours

Assuming neighbours(x) contains no obstacles, the nodes that remain after applying straight or diagonal pruning (as appropriate) are referred to as the *natural neighbours* of x. These correspond to the non-gray nodes in Figure3.1(a) and Figure3.1(c). When neighbours(x) contains an obstacle, not all non-natural neighbours can be pruned. In this case, we say that the evaluation of such neighbours is *forced*.

Definition 1. A node $n \in$ neighbours(x) is *forced* if:

1. n is not a natural neighbour of x, and
2. $\text{len}\left(\langle p(x), x, n\rangle\right) < \text{len}\left(\langle p(x), \ldots, n\rangle \setminus x\right)$

Figure3.1(b) provides an example of a forced evaluation involving a straight move where $n = 3$. Figure3.1(d) shows a similar example for a diagonal move, where the evaluation of $n = 1$ is forced.

3.4.3 Algorithm description

We begin by formally defining the concept of a *jump point*.

Definition 2. Node y is the jump point from node x, heading in direction $\tilde{d}$, if y minimizes the value k such that $y = x + k\tilde{d}$ and one of the following conditions holds:

1. Node y is the goal node.
2. Node y has at least one neighbor whose evaluation is forced, according to Definition 1.
3. $\tilde{d}$ is a diagonal move, and there exists a node $z = y + k_i\tilde{d}_i$, which lies $k_i \in \mathbb{N}$ steps in direction $\tilde{d}_i \in \{\tilde{d}_1, \tilde{d}_2\}$, such that z is a jump point from y by condition 1 or condition 2.

Figure3.1(b) shows an example of a jump point, identified via condition 3. Starting at x and traveling diagonally, we encounter node y. From y, node z can be reached with $k_i = 2$

horizontal moves. Thus, z is a jump point successor of y (by condition 2), which in turn identifies y as a jump point successor of x.

Algorithm 3.4.1: Identify Successors

Input: x: current node, s: start node, g: goal node
Output: Successors of node x

```
1 successors(x) ← ∅ ;
2 neighbours(x) ← prune(x, neighbours(x)) ;
3 foreach n ∈ neighbours(x) do
4     n ← jump(x, direction(x, n), s, g) ;
5     add n to successors(x) ;
6 return successors(x) ;
```

The process for identifying individual jump point successors is outlined in Algorithm3.4.1. We begin by pruning the set of neighbors immediately adjacent to the current node x (line 2). Instead of adding each neighbor n to the set of successors for x, we attempt to "jump" to a node further away but in the same direction as n (lines 3–5). If we find such a node, we add it to the set of successors instead of n. If no jump point is found, nothing is added. The process continues until all neighbors have been exhausted, and the set of successors for x is returned (line 6).

Function 3.4.2: jump

Input: x: initial node, $\tilde{d}$: direction, s: start, g: goal
Output: A jump point or null if no jump point is found

```
1  n ← step(x, d̃) ;
2  if n is an obstacle or is outside the grid then
3      return null ;
4  if n = g then
5      return n ;
6  if ∃n′ ∈ neighbours(n) such that n′ is forced then
7      return n ;
8  if d̃ is diagonal then
9      foreach i ∈ {1, 2} do
10         if jump(n, d̃_i, s, g) is not null then
11             return n ;
12 return jump(n, d̃, s, g) ;
```

In order to identify individual jump point successors we will apply Function3.4.2. It requires an initial node x, a direction of travel $\vec{d}$, and the identities of the start node s and the goal node g. The algorithm attempts to establish whether x has any jump point successors by stepping in the direction $\tilde{d}$ (line 1) and testing if the node n at that location satisfies Definition 2. If so, n is designated a jump point and returned (lines 5, 7, and 11). If n is not a jump point, the algorithm recurses and steps again in the direction $\tilde{d}$, with n as the new initial node (line 12). The recursion terminates when an obstacle is encountered, preventing further steps (line 3). Before each diagonal step, the algorithm must first fail to detect any straight jump points (lines 9–11), corresponding to the third condition in Definition 2, which is essential for preserving optimality.

The algorithm proceeds as follows: Manage an open list using a priority queue to keep track of the path cost and parent node for each expanded node, initialized with the start node.

1. At each iteration, pop the node as current node with the lowest f-value from the open list.
2. Identify successors of current nodes.
3. Add the identified successors to the open list, updating their path cost and parent information.
4. Repeat the above steps until the goal node is found.

3.5 Conflict based search(CBS)

3.5.1 Definitions for CBS

- A constraint is a tuple (a_i, v, t) where agent a_i is prohibited from occupying vertex v at time step t. Agents are associated with constraints during the algorithm. A consistent path for a_i satisfies all its constraints, and a consistent solution is a set of paths where each agent's path is consistent with its constraints.
- A conflict is a tuple (a_i, a_j, v, t), where agents a_i and a_j both occupy vertex v at time t. A solution is valid if no conflicts exist, though a consistent solution may still be invalid if paths conflict despite satisfying individual constraints.

3.5.2 High Level

At the high level, CBS searches a binary tree called the constraint tree (CT)[13]. Each node N in the CT consists of:

- **N.constraints**: A set of constraints for individual agents. The root contains no constraints. Each child inherits the parent's constraints and adds one new constraint for one agent.
- **N.solution**: A set of k paths, one for each agent, consistent with their respective constraints.
- **N.cost**: The total cost of the solution, summed over all agents' path costs (referred to as the f-value).

A CT node is a goal node if its solution is valid, meaning no conflicts exist among the paths. CBS performs a best-first search on the CT, where nodes are ordered by cost. For each node, the low-level search finds consistent paths for all agents, which are then validated by checking for conflicts. If no conflicts are found, the node is a goal node, and its solution is returned. If a conflict $C = (a_i, a_j, v, t)$ is found, the node is non-goal.

3.5.3 Processing a Node in the CT

Given the list of constraints for a node N in the Constraint Tree (CT), the low-level search is invoked. The low-level search (described in detail below) returns a shortest path for each agent a_i that is consistent with all the constraints associated with a_i in node N.

Once a consistent path has been found for each agent (with respect to its own constraints), these paths are validated against each other. The validation is performed by iterating through all time steps and checking for conflicts in the locations occupied by different agents. If no two agents plan to occupy the same location at the same time, the CT node N is declared a goal node, and the current solution ($N.solution$)—which consists of the computed paths—is returned.

However, if a conflict $C = (a_i, a_j, v, t)$ is detected between two or more agents a_i and a_j at location v at time t, the validation process halts, and the node is declared a non-goal node.

3.5.4 Resolving a Conflict

For a non-goal node N with conflict $C_n = (a_i, a_j, v, t)$, at most one conflicting agent can occupy vertex v at time t. To resolve this, CBS splits N into two child nodes, adding constraint (a_i, v, t) to one and (a_j, v, t) to the other. Each child inherits all constraints from N. To maintain optimality, both possibilities are explored.

Instead of storing all cumulative constraints in each node, only the latest constraint is saved, and earlier constraints are retrieved by traversing the tree from the node to the root. The low-level search is then performed only for the agent associated with the newly added constraint, as the paths of other agents remain unchanged.

For conflicts involving more than two agents ($k > 2$), CBS offers two methods to handle such k-agent conflicts:

1. Generate k children, where each child adds a constraint to $k - 1$ agents, allowing only one agent to occupy the conflicting vertex v at time t.
2. Alternatively, focus only on the first two agents detected in the conflict and branch based solely on their conflict.

In all the places where the CBS algorithm and the JPSCBS algorithm are discussed in this thesis, we use the second option and will not go into details later.

3.5.5 Low Level

The low-level search in CBS focuses on finding an optimal path for an agent a_i that satisfies its constraints, while ignoring other agents. The search space includes both spatial and time dimensions. Any single-agent pathfinding algorithm can be used. During the search, when a state (v, t) (where v is a location and t is a time step) is generated, it is discarded if a constraint (a_i, v, t) exists in the current CT node.

The heuristic used is the shortest path in the spatial dimension, ignoring other agents and constraints. For tie-breaking, Standley's conflict avoidance table (CAT) is used where states with fewer conflicts with other agents are preferred. For example, if two states $s_1 = (v_1, t_1)$ and $s_2 = (v_2, t_2)$ have the same f-value, but v_1 is occupied by two other agents at t_1 while v_2 is free, s_2 will be expanded first. This tie-breaking policy improves runtime by a factor of 2 compared to arbitrary tie-breaking.

Duplicate detection (DD) further speeds up the low-level search. Since the search space includes both position and time, two states are duplicates if the position and time for a_i are identical in both.

3.5.6 Meta-Agent Conflict-Based Search (MA-CBS)

Meta-Agent Conflict-Based Search (MA-CBS) is an extension of the traditional Conflict-Based Search (CBS) algorithm, designed to alleviate CBS's performance bottlenecks in highly conflict-prone environments. CBS employs a two-level structure: at the high level, it constructs a Constraint Tree (CT) to perform conflict decomposition, while at the low level, it conducts single-agent path searches. However, when numerous conflicts exist between multiple agents, CBS may generate an excessively deep CT, leading to reduced computational efficiency.

MA-CBS mitigates this issue by dynamically merging conflicting agents under certain conditions, treating them as a single *Meta-Agent*, and jointly solving their optimal path at the

low level. This approach reduces the number of independent conflicts that CBS needs to handle, effectively decreasing the complexity of the search space.

Specifically, MA-CBS defines a conflict threshold B. When the number of conflicts between a pair of agents exceeds B, these agents are merged into a new Meta-Agent, whose path is then computed collectively using a low-level solver such as A*. By appropriately choosing the value of B, MA-CBS achieves a trade-off between CBS and the Independence Detection (ID) framework:

- When $B = \infty$, MA-CBS degenerates into standard CBS, where no agents are merged.
- When $B = 0$, MA-CBS behaves like Independence Detection (ID), merging agents immediately upon their first conflict.

3.6 ICBS

Several variants of CBS have been developed to improve its performance. In this section, we introduce several improvements in ICBS. Below is the pseudo code(Algorithm3.6.1) for CBS integrated with the following enhancements: MA-CBS (Meta-Agent CBS), Prioritizing Conflicts (PC), Merge and Restart (MR) and Bypassing (BP).

3.6.1 Merge and restart

In the Meta-Agent CBS (MA-CBS) framework, merging agents into a *Meta-Agent* reduces the size of the Constraint Tree (CT) but increases the computational cost of low-level pathfinding. The *Merge and Restart* (MR) strategy is designed to minimize redundant computations and improve search efficiency.

Problem Analysis

In MA-CBS, two agents, a_1 and a_2, may be merged after experiencing B conflict resolutions. However, by the time this merging occurs, the CT's OPEN list may already contain $B+1$ relevant nodes, each performing the same merging operation. This results in redundant computations of the same Meta-Agent search, leading to unnecessary computational overhead.

MR Solution

The MR strategy optimizes computation by **preemptively merging agents and restarting the search**:

- When a_1 and a_2 need to be merged, MR discards the current CT and restarts the search from a new root node, where a_1 and a_2 are already treated as a Meta-Agent.
- This approach **eliminates redundant low-level Meta-Agent searches** across multiple CT nodes, improving computational efficiency.

Potential Trade-offs

The MR strategy may lead to some information loss. For instance, if a third agent a_3 had previously conflicted with a_1, then merging a_1 and a_2 may require rediscovering constraints that were originally imposed on a_3. However, in most cases, the computational savings from MR outweigh the potential overhead introduced by such rediscoveries.

3.6.2 Prioritizing conflicts

The core idea behind the PC (Prioritized Conflicts) improvement is to prioritize conflicts based on their impact on the solution cost, focusing first on those conflicts that will directly increase the solution cost. Specifically, conflicts are classified into three types:

Algorithm 3.6.1: High-level of ICBS

```
Initialize R with low-level paths for the individual agents;
Insert R into OPEN;
while OPEN is not empty do
    N ← best node from OPEN // lowest solution cost;
    Simulate the paths in N and find all conflicts.;
    if N has no conflict then
        return N.solution // N is goal;
    C ← find-cardinal/semi-cardinal-conflict(N) // (PC);
    if C is not cardinal then
        if Find-bypass(N, C) then
            Continue;// (BP)
    if should-merge(a_i, a_j) then
        // Optional, MA-CBS: a_ij ← merge(a_i, a_j);
        if MR active then
            Restart search; // (MR)
        else
            Update N.constraints;
            Update N.solution by invoking low-level search for a_ij;
            Insert N back into OPEN;
            Continue; // go back to the while statement
    foreach agent a_i in C do
        A ← GenerateChild(N, (a_i, v, t));
        Insert A into OPEN;
Function GenerateChild(Node N, Constraint C = (a_i, v, t)):
    A.constraints ← N.constraints + (a_i, v, t);
    A.solution ← N.solution;
    Update A.solution by invoking low-level search for a_i;
    A.cost ← SIC(A.solution);
    return A;
```

- **Cardinal Conflicts**: A conflict C is considered cardinal if adding constraints to any of the agents involved in the conflict will increase the cost of their respective paths. In other words, the optimal paths of all agents must pass through the conflict at the same location and time. The characteristic of cardinal conflicts is that once such a conflict is chosen for splitting, the solution cost is guaranteed to increase.

- **Semi-Cardinal Conflicts**: A conflict is considered semi-cardinal if adding a constraint to one agent increases its path cost, but adding a constraint to the other agent does not affect its cost.

- **Non-Cardinal Conflicts**: A conflict is non-cardinal if adding constraints to either agent involved does not increase their path cost.

PC Workflow

When the CBS algorithm selects a node for expansion, PC first inspects all conflicts within the node and classifies them:

1. If a cardinal conflict is detected, it is immediately chosen for splitting. This ensures that the cost of the newly generated child nodes will exceed that of the current node, thereby reducing the search tree size.
2. If no cardinal conflict is found, the algorithm selects a semi-cardinal conflict, which will increase the cost of at least one child node.
3. If neither cardinal nor semi-cardinal conflicts are present, the algorithm randomly selects a non-cardinal conflict for splitting.

By prioritizing cardinal conflicts, PC avoids generating a large number of subtrees with identical costs, thus significantly reducing the number of nodes in the search tree and improving search efficiency.

3.6.3 Bypass

Bypass is an optimization technique designed to enhance the computational efficiency of Conflict-Based Search (CBS) by reducing the growth of the Constraint Tree (CT) and minimizing redundant computations.

Definitions

- For each CT node N, let $N.N_C$ denote the total number of conflicts of the form (a_i, a_j, v, t) between the paths in $N.solution$. Calculating $N.N_C$ is trivial.
- A path P_i' is a valid bypass to path P_i for agent a_i with respect to a conflict $C = \langle a_i, a_j, v, t\rangle$ and a CT node N, if the following conditions are satisfied:
  - P_i' does not include conflict C.
  - cost$(P_i') =$ cost(P_i).
  - Both P_i and P_i' are consistent with $N.constraints$.
- Replacing a path P_i at a CT node N means substituting P_i with a valid bypass P_i' for agent a_i. When this happens, we say that P_i' was adopted by N. Adopting a valid bypass may introduce more conflicts and potentially worsen the overall runtime. Therefore, we only allow the adoption of bypasses that reduce $N.N_C$. These are called helpful bypasses.
- A valid bypass P_i' is a helpful bypass to P_i if $N'.N_C < N.N_C$, where N' is the CT node that adopted P_i'. We use $<$ (not $\leq$) to avoid infinite loops of alternating conflicts.

Bypass1: Peek at the Child

The first method, denoted as Bypass1 (BP1), peeks at either of the immediate children in the CT and tries to adopt their paths. If a child path is a helpful bypass, it is adopted by the parent node without adding a new constraint. The right child will not be generated, and the CT size is not increased. If neither peeking operation finds a helpful bypass, the children are added to OPEN as in basic CBS.

BP1 does not incur extra overhead compared to CBS, as it only avoids adding new nodes to OPEN if a helpful bypass is found. In the worst case, when both peeks fail, BP1 behaves identically to CBS, generating and adding both children to OPEN. Adopting a path from a child can save significant search effort by reducing the size of the CT.

Bypass2: Deep Search for Bypasses

Bypass2 (BP2) generalizes Bypass1 (BP1). Define $ST(N)$ as the subtree below N (including node N itself), containing only nodes with the same cost as $N.cost$. BP2 searches, in best-first order (according to $N.N_C$), through the entire set of nodes in $ST(N)$ to find a helpful descendant, i.e., a descendant $N' \in ST(N)$ such that $N'.N_C < N.N_C$.

Algorithm 3.6.2: BP2. (BP1 changes line 1)

Input: Node N

```
1 foreach l ∈ ST(N) in a best-first order do
2     C ← first conflict (a_i, a_j, v, t) in l;
3     foreach agent a_i in C do
4         A ← generate child(l, (a_i, s, t));
5         if A.cost = P.cost and A.N_C < P.N_C then
6             N.solution ← A.solution;
7             Insert N to OPEN;
8             return true;
9 return false;
```

In line 4, BP2 calls the function 'generate-child' (shown in Algorithm3.6.2), which invokes the low-level search. If the child is a helpful descendant (i.e., it has the same cost but fewer conflicts, line 5), this low-level path is returned and adopted by N (lines 6 and 7). BP1 uses the same pseudocode except that in line 1, it only allows the two immediate children.

It's important to note that in practical implementation, when BP2 fails to find a helpful descendant (line 9), all the frontier nodes from this search are available and can be passed to Algorithm3.6.2, which will directly add them to OPEN without regenerating them (or invoking the low-level search again for these nodes). Additionally, node N, after adopting the solution of one of its descendants, will now be the best node in OPEN and will be chosen for expansion next. Thus, if it again has a helpful descendant, the bypass process will repeat. In fact, the next real split will only occur when a bypass call fails.

3.7 ICBS-H

Improved Conflict-Based Search with Heuristics (ICBS-h)[3] is an optimization of Improved CBS (ICBS), designed to reduce the search space in Multi-Agent Pathfinding (MAPF) by heuristically estimating future search effort.

3.7.1 Problem Analysis

Traditional CBS relies solely on the current path cost of Constraint Tree (CT) nodes—denoted as the g-value—to guide the search. However, it lacks an effective mechanism to estimate future search costs, leading to inefficient expansions. ICBS-h addresses this limitation by computing admissible heuristic costs—denoted as the h-value—providing a more informed high-level search and reducing unnecessary expansions.

3.7.2 Core Methods

Conflict Classification: ICBS-h employs a *Cardinal Conflict Graph* to analyze critical conflicts (Cardinal Conflicts) within CT nodes. Based on this conflict graph, it computes heuristic costs.

Admissible Heuristic Cost Calculation:

- **Disjoint Cardinal Conflicts:** If a node N has x mutually independent cardinal conflicts, then its heuristic estimate is given by $h(N) = x$, which is admissible.
- **Minimum Vertex Cover (MVC):** The heuristic quality is further improved by leveraging the Minimum Vertex Cover method to estimate the minimal number of conflicts

that must be resolved.

Search Optimization: During high-level search, ICBS-h integrates both g-values and h-values, similar to how A* improves upon Dijkstra's algorithm. This allows the search to prioritize promising solutions more efficiently.

3.8 CBSH2

CBSH2 (Conflict-Based Search with Heuristics 2)[14]is an improvement over CBSH, designed to optimize high-level search in CBS by providing more accurate heuristic estimates. This enhancement reduces the size of the search tree and improves computational efficiency.

3.8.1 Background

CBSH introduced a heuristic method for CBS high-level search using the *Cardinal Conflict Graph* (CG). By computing cardinal conflicts within Constraint Tree (CT) nodes, CBSH provides an admissible heuristic estimate for each node. However, CG-based heuristics only account for conflicts in the current solution and do not predict future conflicts, which can still lead to a large search space in certain cases.

CBSH2 improves upon CBSH by incorporating two more advanced heuristic methods:

- **Dependency Graph Heuristic (DG)**: Considers potential future conflicts and infers inter-agent dependencies at the child node level.
- **Weighted Dependency Graph Heuristic (WDG)**: Extends DG by not only identifying agent dependencies but also quantifying the additional path cost required to resolve these conflicts.

3.8.2 Key Improvements

Dependency Graph (DG) Heuristic DG constructs a dependency graph between agent pairs to anticipate future conflicts that may arise in CT nodes. This proactive estimation increases the heuristic value h, enhancing the efficiency of CBS high-level search.

For instance, in certain cases, even if a conflict is classified as a *non-cardinal conflict*, DG can still detect that all potential optimal paths will eventually lead to new conflicts. By preemptively estimating cost increases, DG enables more informed search decisions.

Weighted Dependency Graph (WDG) Heuristic WDG further refines DG by not only recording potential conflicts between agents but also calculating the minimal cost required to resolve them.

By leveraging the *Edge-Weighted Minimum Vertex Cover* (EWMVC), WDG provides a more precise prediction of the minimum cost increase for each CT node, thereby optimizing the search order.

3.9 Disjoint Splitting (DS)

Disjoint Splitting (DS) [15] is a key optimization for Conflict-Based Search (CBS) designed to reduce redundant computations in high-level search. In standard CBS, each conflict resolution generates two subproblems. However, since these subproblems still share partially overlapping solution paths, redundant exploration of the same solution space occurs, reducing efficiency. Disjoint Splitting introduces *positive constraints* to partition the problem into completely disjoint subproblems, thereby avoiding duplicate computations.

3.9.1 Core Improvements

Issues in Standard CBS Splitting Standard CBS employs *negative constraints* to resolve conflicts, meaning that when two agents a_1 and a_2 conflict at a location v, CBS generates two subproblems:

- One where a_1 is prohibited from entering v.
- One where a_2 is prohibited from entering v.

However, these two cases are not entirely independent—under certain circumstances, the subproblems may still share partially identical solutions, leading to redundant searches.

Disjoint Splitting Approach To address this, DS introduces *positive constraints* alongside negative constraints during conflict resolution:

- One subproblem enforces that one agent *must* pass through v.
- The other subproblem prohibits the other agent from entering v.

This ensures that each subproblem generates completely **disjoint** solution sets, eliminating solution path redundancy between subproblems and improving search efficiency.

3.9.2 Performance Improvements

Theoretically, Disjoint Splitting guarantees a complete partitioning of the search space, preventing redundant computations caused by overlapping solution paths in traditional CBS.

3.10 ECBS, BCBS, GCBS

Conflict-Based Search (CBS) is a widely used optimal algorithm for the Multi-Agent Pathfinding (MAPF) problem. However, due to the need to exhaustively search the Constraint Tree (CT), CBS has high computational complexity, making it difficult to scale to large problems. To address this limitation, Barer proposed suboptimal variants of CBS[4], which trade optimality for improved computational efficiency.

3.10.1 Greedy CBS (GCBS)

GCBS accelerates the search by employing a **non-optimal conflict selection strategy** at the high level, prioritizing the expansion of CT nodes that are more likely to quickly yield a feasible solution:

- **Conflict heuristics:**
  - Compute the number of conflicts in each CT node and prioritize nodes with fewer conflicts.
  - Assess the number of agents involved in each conflict and prioritize resolving conflicts with fewer affected agents.
  - Construct a **conflict graph** and use **minimum vertex cover (MVC)** to estimate the minimal number of conflicts that must be resolved.
- GCBS sacrifices optimality in exchange for faster search, making it suitable for applications where solution quality is less critical but quick results are required.

3.10.2 Bounded-Suboptimal CBS (BCBS)

BCBS introduces the **Focal Search** mechanism at both the high and low levels of CBS, allowing it to return a suboptimal solution while guaranteeing that the solution cost does not exceed w times the optimal cost:

- **High-level Focal Search:**
  - Expand CT nodes whose costs do not exceed $w\times$ the cost of the current best solution, reducing the number of expanded nodes.
- **Low-level Focal Search:**
  - Employ **Weighted A* (WA*)**, ensuring that single-agent paths do not exceed w times their optimal cost.
- BCBS(w_H, w_L) allows users to set different weight parameters for the high and low levels, offering a trade-off between computation time and solution quality.

3.10.3 Enhanced CBS (ECBS)

ECBS improves upon BCBS by optimizing the selection strategy for the **Focal List**, making the search more flexible:

- **Utilizes a dynamic cost bound LB in low-level search** as the selection criterion for the Focal List, improving search efficiency.
- **Compared to BCBS, ECBS finds better solutions under the same suboptimality constraints**.

3.10.4 Summary

ECBS, BCBS, and GCBS significantly improve the efficiency of CBS by relaxing its optimality constraints, making them suitable for large-scale MAPF problems. Among them, ECBS provides the best balance between suboptimality guarantees and computational efficiency, making it the preferred suboptimal CBS method.

4 Continuous-Search Jump Point Search

To accommodate our algorithm, we extended JPS to support continuous search and modified its output. Specifically, in addition to returning the path, our continuous-search JPS also outputs the set of specific jump points and possible intervals, as described below. This additional information enables localized conflict resolution between jump points.

4.1 Definitions

Definition 1. A *Trimmed Symmetric Path* refers to a path that is pruned during the execution of Jump Point Search (JPS) due to the pruning strategy. These paths are typically equivalent to already expanded paths in heuristic search, meaning they can reach the same target node with no higher cost than the selected path. To enhance search efficiency, JPS ignores such symmetric paths and retains only the essential *jump points*, thereby reducing unnecessary state expansions.

Definition 2. A *Possible Interval* is a specific interval in a path found by Jump Point Search (JPS), where an *Trimmed Symmetric Path* with the same cost can be found for the path within the interval.

Definition 3. A *JPSPath* is path with extra information. It contains the path for a agent, specific jump points along the path and possible intervals.

4.2 Continuous Search

We extend Jump Point Search (JPS) to enable it to search for suboptimal paths. We maintain a *state* variable to store the state of each agent, which includes the agent-specific open list. This modification allows JPS to continue searching for suboptimal paths. The pseudo code (Algorithm 4.2.1) is shown below.

4.2.1 State Management

The state is maintained as follow.

- Each agent maintains an open list within its state.
- The state is passed to JPS during each invocation.

After the initial JPS search, when a new path is required, JPS uses the stored open list within its state to resume the search from previously unexpanded nodes. During each search, a close list is maintained for explored nodes. However, If a node is expanded again during later jumping operations, it is neither discarded nor updated immediately. Instead, the node is added to a temporary set for use in the future continuous JPS search for this agent. When the current JPS search is finished, all nodes in the temporary collection will be added to the open list for the next search. (line 16-17)

4.2.2 Initialization of State

We initialize the state as follow:

- The open list for each agent is initialized with its start node.
- JPS is then invoked with this initial state, ensuring that the first optimal path is found.
- Once a path is found, all remaining unexpanded nodes in the open list are retained in the agent state for future searches.

By maintaining an open list for each agent across different stages of the search, JPSCBS efficiently explores suboptimal paths without re-executing JPS from the start.

Algorithm 4.2.1: Jump Point Search (JPS)

Input: $start$, $goal$, $grid$, $state$
Output: $path$

```
1  Initialize closed as empty list;
2  Initialize temp_nodes as empty list;
3  while state.open is not empty do
4      current ← node pops from state.open with lowest f-value;
5      Add current to closed;
6      if current.pos == goal then
7          Move all nodes in temp_nodes to state.open;
8          Move all nodes in closed to state.closed;
9          return ReconstructPath(current);
10     successors ← PruneNeighbours(current, grid);
11     foreach successor ∈ successors do
12         cost ← getMoveCost(current.pos, successor);
13         g ← current.g + cost;
14         h ← heuristic(successor, goal);
15         Create new node with (successor, g, h, current);
16         if successor in closed then
17             Add node to temp_nodes;
18         else
19             Add node to state.open;
20 return empty path;
```

4.2.3 Reconstruct Path

The ReconstructPath function reconstructs the path from the goal node back to the start node. It backtracks using the parent links stored in each node, collecting the sequence of actions, the jump points and the possible intervals along the way.

4.3 JPSPath - Jump Points

To better handle conflicts between jump points, we keep only specific jump points along the path by JPS search when constructing JPSPath result.

In JPSPath, *jump points* refer exclusively to the turning points among those expanded by JPS in the final path. These points are stored in the order in which they are expanded during the search.

Furthermore, all intermediate turning points within possible intervals are removed, preserving only the start and end vertices of each interval.

4.4 JPSPath - Possible Interval

In Figure4.1, there are many equally valid best paths through the rectangular area, each of these paths costs the same. The only difference is in what order we choose to move diagonally or horizontally. Among them, only one representative path is selected as the final result, while the others are disregarded. In Jump Point Search (JPS), due to the

diagonal-first rule, the agent will jump from the start vertex (green) to the intermediate jump point – gray vertex, and subsequently to the goal vertex (red) as the final result.

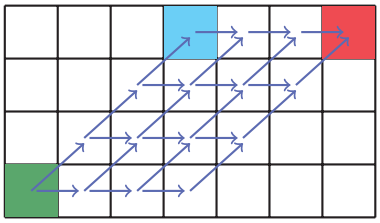


Figure 4.1: Symmetric paths

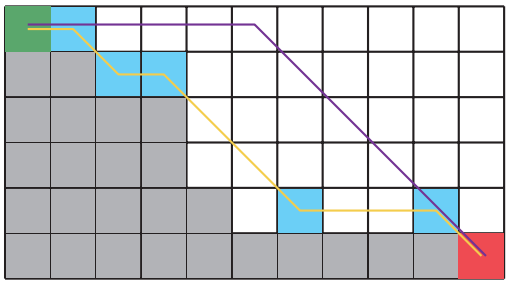


Figure 4.2: Example of symmetric paths

Figure 4.2 shows a path (yellow line) and its jump point (cyan vertex) found by JPS search. Let's review the pruning strategy of JPS. For a jump point comes from straight move, the neighbours that can be accessed by parent node with less or equal cost is pruned. For a jump point comes from diagonal move, the neighbours that can be accessed by parent node with less cost is pruned. Since the jump points that before and after which doesn't have a change of direction exists because there is a need to jump in other directions, we will only look into turning points.

There are three possible types of turns at a turning point: (a) Diagonal-to-Diagonal, as shown in Figure 4.3, (b) Straight-to-Diagonal, as shown in Figure 4.4, and (c) Diagonal-to-Straight, as shown in Figure 4.5. Other types of turning points, such as Straight-to-Straight, are trivially suboptimal and are not considered here, following the pruning rules introduced earlier in Chapter 3, JPS Section 3.4.2.

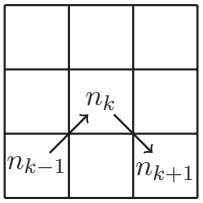


Figure 4.3: Case(a)

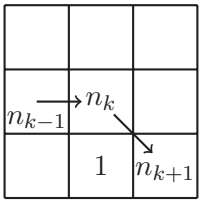


Figure 4.4: Case(b)

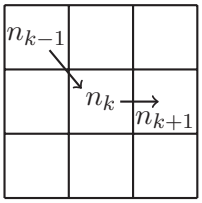


Figure 4.5: Case(c)

4.4.1 Turning Point - Case (a) and Case (b)

If a turning point is either a Diagonal-to-Diagonal or a Straight-to-Diagonal turn, the pruning rules dictate that:

- In Case (a), node n_k resulting from a straight jump will not consider diagonal neighbors. (see Figure3.1 a)
- In Case (b), node n_k resulting from a diagonal jump will not consider diagonal neighbors in other directions. (see Figure3.1 c)

This implies that there must exist a forced neighbor. According to the forced neighbor rule, the next jump point n_{k+1} cannot be reached from n_{k-1} with a cost less than or equal to that of the current path unless n_k appears on the path. Consequently, n_k is a necessary turning point to ensure the path cost does not increase.

4.4.2 Turning Point - Case (c)

If a turning point is a Diagonal-to-Straight turn, forced neighbors are not necessarily present. According to the pruning rules, n_k from a diagonal jump considers neighbors that cannot be reached from n_{k-1} with a lower cost than any path that does not contain n_k. A Diagonal-to-Straight turning point may introduce a path segment where an alternative path exists with the same cost but without n_k. In such cases, n_k is unnecessary for the path to retain cost.

To better understand why multiple paths can have the same cost, we analyze the impact of pruning rules. A diagonal move prunes neighbors that cannot be reached from the parent node with any path that excludes itself at a lower cost. A straight move prunes neighbors that cannot be reached from the parent node with any path that excludes itself at an equal or lower cost. As a result, if a path is not applying any forced neighbor rule, a vertex that can be reached through a combination of diagonal and straight moves will favor the diagonal-first approach. This implies that the same path cost can be achieved by rearranging the sequence of diagonal and straight moves in an alternative permutation.

Therefore, to resolve conflicts between jump points, it is unnecessary to retain the turning point of Case (c) in the set of jump points. This is because the cost of the current path can be preserved without traversing this jump point and the other equivalent paths may be useful for resolving conflicts. To address this, we remove the turning point from the jump points in JPSPath. Instead, we designate its preceding jump point, n_{k-1}, as the starting point of the interval and its succeeding jump point, n_{k+1}, as the endpoint of the interval. We then define (n_{k-1}, n_k, n_{k+1}) as a possible interval and store it in the set of possible intervals within JPSPath.

4.5 Theoretical Analysis of Continuous Search in JPS

In this section, we formally demonstrate that the continuous search in JPS does not proceed indefinitely and provide an estimation of the total number of paths that JPS may explore.

4.5.1 Finiteness of Paths Generated by Continuous Search

We begin by establishing the following assumptions:

- The search space S is finite (e.g., a grid map of size $N \times M$).
- All edge weights are non-negative.
- Waiting actions are not considered.
- The heuristic function $h(n)$ is admissible and consistent.

Under these conditions, in an A* search (such as octile search without waiting actions), the algorithm is guaranteed to find the optimal path. Once A* has identified the optimal path, any continued search will return the next best path from the remaining nodes in the open list. While this process may continue, it can theoretically generate only a finite number of paths from the start to the goal.

This conclusion can be justified as follows:

Properties of Paths:

- In a finite state space S, any valid path must be a **simple path** (i.e., it does not contain cycles).
- Each state can be visited at most once in any given path.

- The maximum possible path length is bounded by $|S|$.

Upper Bound on the Number of Paths:

- Given a finite state space of size $|S|$, the number of distinct simple paths from the start to the goal is combinatorially bounded.
- This upper bound can be expressed in terms of permutations, with a theoretical maximum of $|S|!$.

Therefore, the total number of possible paths is finite, and continuous search cannot generate an infinite number of distinct paths.

4.5.2 Estimated number of paths from Continuous Search

Jump Point Search (JPS) is a pruning-based optimization technique built upon A* that accelerates path planning by reducing redundant node expansions. A jump point is only generated when its parent node cannot reach it through a shorter path. In other words, regardless of how many valid paths with the same cost exist between two jump points, JPS consolidates them into an equivalent path, thereby significantly reducing the search space.

Moreover, in the process of handling jump points and identifying valid intervals, only two turning points are left in jump points that can influence the final path cost and remain in the search. These two jump points exists because of forced neighbors. JPS seeks the most efficient way around obstacles. Therefore, the total number of distinct paths JPS can explore through continuous search is primarily determined by the shape, number, and placement of obstacles—that is, how obstacles disrupt path symmetry.

However, in scenarios where conflicts are not dense, the continued search in JPS typically does not require exploring a large number of paths. This is because, in many cases, resolving conflicts at the jump points of the first few JPS searched paths is sufficient to identify the optimal single agent plan for the final solution.

4.6 Different Choice for Possible Intervals

In the previous section on possible intervals, we identified the jump points preceding and succeeding the turning point in Case (c), along with the turning point itself, forming the interval (n_{k-1}, n_k, n_{k+1}) in which conflicts can be resolved. However, as illustrated in Figure 4.2, equivalent paths that maintain the same cost are not necessarily confined to this predefined interval.

This occurs because, after removing n_k, the jump point n_{k+1} is no longer a straight-to-diagonal turning point. Consequently, the forced neighbor rule in JPS no longer applies, making it invalid in this context. Despite this, we can still employ the cost evaluation method used in JPS to compare the path cost when passing through this point versus not traversing it. This allows us to determine in what kind of interval that we could find alternative path with same cost.

Case 1: We first analyze the case where the next turning point after n_k follows a diagonal direction different from the direction from n_{k-1} to n_k. Since n_k has been discarded, we need to determine whether n_{k+2} can be reached without passing through n_{k+1} while maintaining the same or a lower cost.

Before discarding n_k, n_{k+1} was a turning point from a straight-line move to a diagonal move. According to the forced neighbor rule, position 1 in the figure must be occupied by an obstacle. Consequently, two alternative paths exist:

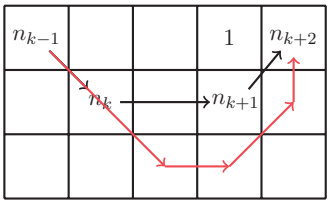


Figure 4.6: Case 1

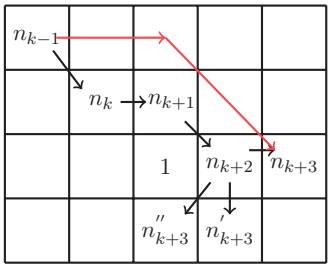


Figure 4.7: Case 2

- n_{k-1} finds a path from the north of obstacle 1 to n_{k+2}.
- n_{k-1} finds a path from the south of obstacle 1 to n_{k+2}.

From our previous theoretical analysis, the continued search in A* will explore all paths from the start to the goal, whereas JPS will ignore all equivalent or more costly paths. Therefore, as long as the jump points in the two paths differ, they will be considered separate paths in the continued search. If n_{k-1} finds a path from the north of obstacle 1 to n_{k+2}, it will, at the very least, avoid passing through n_k. Since the two paths are distinct, there is no need to consider bypassing from the north, as the corresponding path will already be explored separately.

If we choose to bypass from the south, we mark n_{k+1} as unpassable and invoke A*. The search then identifies the red path as the closest feasible path from n_{k-1} to n_{k+2}, but its cost exceeds that of the original path. In this scenario, to maintain the original path cost, n_{k+1} must not be discarded.

Case 2: We apply the same analytical method. Suppose the next jump point n_{k+2} after n_{k+1} is an identical turning point to n_k—i.e., it is also a diagonal-to-straight turning point, with both diagonal and straight moves following the same directions. In this case, there may exist an alternative path with the same cost that replaces the segment from n_{k-1} to n_{k+3}, rendering n_{k+2} unnecessary.

Next, we examine the cases of n'_{k+3} and n''_{k+3}. If we mark n_{k+2} as unpassable, then from the north side of obstacle 1, it becomes impossible to find a path to n'_{k+3} or n''_{k+3} with an equal or lower cost. Conversely, if approaching from the south, another JPS path will be explored, corresponding to this scenario.

Conclusion: In summary, only when the next jump point after a diagonal-to-straight turning point (e.g., n_k) is also a diagonal-to-straight turning point with identical diagonal and straight move directions, can the two possible intervals be merged into a single interval. Within this merged interval $(n_{k-1}, n_k, n_{k+1}, n_{k+2}, n_{k+3})$, an alternative path may exist with the same cost that bypasses both n_k and n_{k+2}.

4.6.1 Choice of Possible Intervals:

Based on the analysis above, we can define the possible intervals using the following approaches:

- Maintain the form (n_{k-1}, n_k, n_{k+1}) as described in Section 4.4, meaning that each possible interval contains only a single diagonal-to-straight turning point.
- Collect all successive, identical diagonal-to-straight patterns and include all turning points within the possible interval, forming $(n_{k-1}, n_k, n_{k+1}, \ldots, n_{k+i})$.

- Collect a fixed number of successive, identical diagonal-to-straight patterns, say k, and include the corresponding turning points within the possible interval. For example, if each possible interval includes jump points from at most two successive, identical diagonal-to-straight patterns, then each possible interval contains five jump points: $(n_{k-1}, n_k, n_{k+1}, n_{k+2}, n_{k+3})$.

The performance of each option depends on the size of the specific map, and the size, shape, and distribution of obstacles.

5 JPSCBS

In Conflict-Based Search (CBS), all agents involved in a conflict are assigned a constraint at the conflicting time step respectivly in generated child node, and A* search is used at the low level to find new paths for each agent. However, due to the existence of many symmetric paths in the grid map, we do not have to search the path from the start to end every time, but can try to take other symmetric paths with the same cost within the local conflict area to avoid the conflict. When a conflict cannot be avoided, as in standard CBS, we generate two children of the current node and attempt to resolve the conflict but we do it locally between certain jump points.

5.1 Definitions

Definition 1. A *constraint info* is a tuple (c, jp_1, jp_2), where c represents the constraint, and jp_1 and jp_2 denote the two jump points along the path that we choose to re-plan the path segment between.

At the high level of basic CBS, the constraint tree is processed by validating paths to detect conflicts. When a conflict (a_i, a_j, v, t) is identified, it is resolved by imposing constraints on the relevant agents. In JPSCBS, we directly generate the corresponding constraints for the affected agents upon detecting a conflict. Consequently, in the relevant pseudocode and discussions, we use the precomputed constraints or constraint info directly to handle conflicts efficiently.

5.2 High Level

In the following section, we describe the high-level process of JPSCBS, which is a modified version of CBS.

5.2.1 The constraint tree

At the high level, the overall structure is almost the same as that of CBS which searches a tree called the constraint tree (CT). However, we make some changes to make it suitable for jump point search. The solution of a constraint tree is changed from a set of paths to a set of priority queue which store the $path$ from low-level search, one for each agent, peek the path with the highest priority as the single agent plan of corresponding agent, consistent with their respective constraints. This structure enables efficient access to the best path while retaining alternative paths as backup solutions.

Node N in the CT is a goal node when the set of paths fetched from priority queue of N.solution for all agents is valid, i.e., has no conflicts. The high level performs a best-first search on the CT where nodes are ordered by their costs.[1] In our implementation, ties are broken in favor of CT nodes whose associated solution contains fewer conflicts. Further ties were broken in a First-In-First-Out (FIFO) manner.

5.2.2 Tie-Breaking Rules for the Solution Priority Queue

For each node N in CT, the highest-priority path from each agent's priority queue in N.solution is selected to form a solution. However, since there may be multiple paths per agent in the priority queue of N.solution, it is possible that several paths have the same cost for a single agent. In such cases, we break the tie by choosing the path that is in the combination of paths that results in the least number of conflicts among all combinations. Further ties are resolved in a First-In-First-Out (FIFO) manner.

For example, consider a k - agent scenario ($k > 2$) where two agents - say a and b - both have two minimum-cost paths while the remaining $k-2$ agents each have a unique minimum-cost path. This scenario generates four possible solutions:

$$\{\{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\}, \{a_2, b_2\}\} \cup \{\text{paths of remaining } k-2 \text{ agents}\}$$

The algorithm evaluates the number of conflicts for each combination and selects the one that minimizes conflicts. Empirical analysis of MAPF instances reveals that the occurrence of multiple minimum-cost paths for a single agent is rare, and the probability of multiple agents simultaneously having multiple minimum-cost paths is even lower. As a result, the computational overhead of explicitly evaluating conflicts across all path combinations is negligible and does not significantly impact runtime compared to a simple first-in-first-out (FIFO) tie-breaking strategy.

Algorithm 5.2.1: High-level

```
1  Initialize R.solution and backups with jump point search;
2  Insert R into OPEN;
3  while OPEN is not empty do
4      N ← best node from OPEN
5      Simulate paths in N to detect conflicts;
6      if N has no conflict then
7          return N.solution;
8      Cs ← GenerateConstraintInfos(N);
9      if FindBypass(N, Cs) then
10         continue;
11     foreach ConstraintInfo c_i in Cs do
12         A.constraints ← N.constraints + c_i.constraint;
13         A.solution ← ResolveConflictLocally(N, c_i);
14         A.cost ← SIC(A.solution);
15         UpdateSolutions(A);
16         ValidateAndRepairNode(A);
17         Insert A into OPEN;
```

5.2.3 Processing a node in the CT

The JPSCBS high level pseudo code is shown above. The root node is initialized by generating individual paths for all agents with jump point search(JPS). These initial paths are also stored the map of path list $backups$. The $backups$ stores all the paths for each agent searched by JPS which will be used by any node that needs a better solution compared to the best path in the priority queue of the node solution later. (line 1-2)

At each iteration, the node N with the lowest solution cost is selected from OPEN. Then the validation is performed by iterating through all time steps and checking the locations reserved by all agents. If no two agents plan to be at the same location at the same time, this CT node N is declared as the goal node, and the current solution (N.solution) that contains this set of paths is returned. If, however, while performing the validation, a set of constraints info $Cs = (c, jp_1, jp_2)$ is found for two agents a_i and a_j , the validation halts and the node is declared a non-goal node. (line 5-8)

Once a goal is declared a non-goal node with found constraints info, we first find bypass which will try to find a path that has the same cost for the agent as ICBS do.(line 9-10) If

it fails, we resolving the constraints by splitting the node N into two children in the same way as CBS do. However, we don't invoke low-level search to find a shortest new path for each agent. Instead, we invoke low-level search between the two jump points jp_1, jp_2 in Cs to find a new local path to solve the conflict, which will be described in detail later.

5.2.4 Path Update Strategy

After the conflict is resolved, the path cost of the affected agent increases. It is possible that the path that JPS continues to search has lower cost than the path after resolving the conflict, so we need to check whether the path in backups needs to be updated. When the cost of the best path of certain agent exceeds or equals the known worst path in backups, the algorithm attempts to find a new path by JPS. If a new path is found, it is added to the global backups list. All newly discovered paths with lower costs than the best path of certain agent are added to the priority queue.

Once the update of priority queue of solution is done. The new paths added might against some constraints. We resolve these violated constraints just as we did in processing constraint tree node, first looking for bypasses, and then trying to resolve the conflict locally.

5.3 Finding Bypass

The bypass improvement used in our algorithm is not fundamentally different from the bypass in ICBS. The key distinction lies in the scope of the bypass search: instead of applying it to the entire path from the agent's start position to the goal, our approach confines the search to the path segments between jump points in the constraint info.

Algorithm 5.3.1: BP2. (BP1 changes line 2)

```
1  Function FindBypass(Node N, ConstraintInfo[ ] Cs):
2      foreach l ∈ ST(N) in a best-first order do
3          foreach ConstraintInfo ci in Cs do
4              A ← N;
5              path_segment ← FindPath(A, ci);
6              Replace the path_segment at A;
7              if A.cost = P.cost and A.N_C < P.N_C then
8                  N.solution ← A.solution;
9                  Insert N to OPEN;
10                 return true;
11     return false;
12 Function FindPath(Node N, ConstraintInfo ci):
13     temp_constraints ← N.constraints + ci.constraint;
14     path ← Astar(start, goal, temp_constraints, time);
15     if has_better_solution(new_path, ci, current_path) then
16         return FindPath(N, ConstraintInfo(c, jp1, next_jp));
```

The definition and concept of bypass have already been introduced and explained in Chapter 3, CBS Section 3.6.3, and will not be reiterated here. However, we introduce a modification to the definition of path replacement.

Definition (Path Replacement): Replacing a path segment P_i at a CT node N in the context of constraint c_i refers to substituting the path segment P_i with a valid bypass P_i'

for agent a_i. The path segment P_i is extracted from the single-agent plan of agent a_i and is defined as the segment between the first jump point in the constraint information (serving as the start point) and the second jump point in the constraint information (serving as the end point). When this replacement occurs, we say that P_i' has been *adopted* by N.

We first select the best `JPSPath` for each agent from the solution of the current best CT node. We then check for conflicts at time t. Upon detecting the first conflict, we generate constraint information (c, jp_1, jp_2) for the two involved agents. Specifically, we generate constraint c, determine the two jump points jp_1 and jp_2 in the `JPSPath` between which the constraint occurs, and store these jump points.

For each agent, we attempt to find a bypass for the path by searching for an alternative path segment between jp_1 and jp_2 that maintains the same cost but does not violate constraint c.

If the newly found path segment satisfies the conditions for discarding the interval's endpoint which will be described in later section, then jp_2 is discarded. The next jump point, `next_jp`, following jp_2 in `jump points` of `JPSPath` is then taken as the new endpoint of path segment. Consequently, the alternative path from jp_1 to `next_jp` is now searched. This process is applied recursively until the endpoint of the current searched path segment either no longer needs to be discarded or already corresponds to the goal of the path.

5.4 Resolve a conflict

In terms of conflict resolution, JPSCBS remains similar to CBS. The key difference lies in how the low-level search is utilized: instead of re-searching the entire path for the agent, JPSCBS re-searches only the path segment that contains the constraint in the best path of the current solution.

The conditions that require a discard point are the same as those in the bypass-finding process. Consequently, when resolving conflicts, it is unnecessary to re-search the path segments for both agents involved in the conflict. If the attempt to find a bypass fails, we retain the path found during the bypass search and directly use the saved new path segment to replace the corresponding path segment when resolving conflicts.

Function 5.4.1: ResolveConflictLocally

Input: Node N, ConstraintInfo ci
Output: Node A.solution

1 $A \leftarrow N$;
2 `path_segment` $\leftarrow$ `FindPath`(A,ci);
3 Replace the `path_segment` at A;
4 **return** A.*solution*;

5.5 Low Level - JPS

In the high-level process of JPSCBS, we maintain a global variable to store the state of each agent, which includes the agent-specific open list. When processing of a CT node in high level, if a new path is required by path update strategy, JPS utilizes the stored open list to continue searching.

5.6 Low Level - A* variants

When resolving conflicts between jump points, the basic JPS algorithm is unable to handle dynamic obstacles. Therefore, an additional low-level search is required to find valid paths. In our approach, we employ Space Time A*. Various other algorithms can also be used, including A* variants such as EPEA* and SIPP. Any MAPF solver that possesses the following three attributes may be used at the low level[1]:

1. **Completeness** – The solver must return a solution if one exists; otherwise, it must return false.
2. **Constraint Handling** – The solver must never return a solution that violates a constraint.
3. **Optimality** – The solver must return the optimal solution.

However, a key question remains: when a conflict occurs, between which two jump points should we apply Space-Time A* to resolve it? Namely, we need to define the jp_1 and jp_2 in the constraint info. We propose JPSCBS-S, which provides near-optimal solutions.

5.7 JPSCBS - Suboptimal

We propose the suboptimal version of JPSCBS. When a conflict occurs, we select the preceding jump point along the path closest to the conflicting vertex v as jp_1 in the constraint info and the next jump point along the path closest to the conflicting vertex v as jp_2. The path segment between jp_1 and jp_2 is then re-planned.

5.7.1 Discard Jump Point in JPSCBS-S

When resolving conflicts between jump points, it is sometimes necessary to discard certain jump points to maintain the quality of solution.

For example, consider the scenario illustrated in Figure 5.1. Suppose a conflict arises in the path from the preceding jump point to n_k. In response, we invoke A* to find an alternative path, which results in the red path shown in the figure. Starting from point 2, we observe that the green path has a lower cost compared to the segment from point 2 along the red path to n_k, then continuing to n_{k+1}, and subsequently to n_{k+2}. This suggests that discarding certain jump points may lead to a path with less cost.

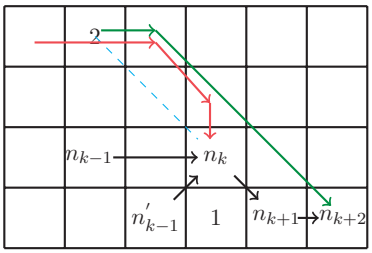


Figure 5.1: Case 1

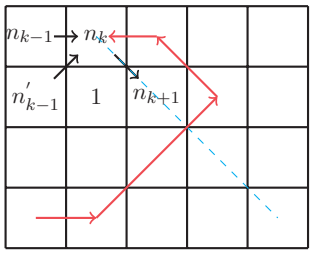


Figure 5.2: Case 2

We first discuss Case 1, where n_{k-1} finds a path from the north of obstacle 1 to n_k. We draw a ray from n_k in the direction of n_{k+1} to n_k. If every vertex in the found path segment lies on the same side of this ray as n_{k-1} (excluding vertices on the ray itself), then n_k remains a necessary jump point to maintain the path cost. However, if every vertex in the found path segment lies on the opposite side of the ray as n_{k-1} (including vertices on the ray), then n_k should be discarded, as there may exist a more cost-efficient path. A detailed analysis of this scenario is similar to the discussion in 4.6 for Case 2.

Case 2 considers the situation where n_{k-1} finds a path from the south of obstacle 1 to n_k. This implies that bypassing obstacle 1 from the north is no longer optimal. However, the case of bypassing obstacle 1 from the south will be handled by another path found through the JPS search, following the same reasoning as in 4.6 for Case 1. Therefore, in this scenario, the current path should be discarded; otherwise, two identical paths may exist simultaneously in the solution of the current node, leading to redundant processing of these paths in further conflict resolving.

6 Implementation

We have implemented the JPSCBS-S algorithm, and the complete source code is available in the GitHub repository: https://github.com/TaiquanSui/JPSCBS. The implementation was developed entirely from scratch, without referencing any publicly available source code or implementations from existing research papers.

Since most of the details have been thoroughly explained in the theoretical sections, we do not repeat them here. Instead, this section focuses on additional implementation aspects that were not explicitly covered in the theoretical discussion.

6.1 Possible Interval

Algorithm 6.1.1: PossibleIntervals

Input: `s`: Start vertex, `g`: Goal vertex, `jumps`: List of jump points
Output: List of intervals containing jump points

```
1  intervals ← [], i ← 0;
2  while i < length(jumps) −1 do
3      if i +2 < length(jumps) then
4          current ← jumps[i];
5          next ← jumps[i + 1];
6          next2 ← jumps[i + 2];
7          dir1 ← CalculateDirection(current, next);
8          dir2 ← CalculateDirection(next, next2);
9          if IsDiagonal(dir1) and IsStraight(dir2) then
10             interval_points ← [current, next, next2];
11             j ← i + 2;
12             while j +2 < length(jumps) do
13                 pattern_start ← jumps[j];
14                 pattern_mid ← jumps[j + 1];
15                 pattern_end ← jumps[j + 2];
16                 pattern_dir1 ← CalculateDirection(pattern_start,
                    pattern_mid);
17                 pattern_dir2 ← CalculateDirection(pattern_mid, pattern_end);
18                 if IsDiagonal(pattern_dir1) and IsStraight(pattern_dir2) and
                    pattern_dir1 = dir1 and pattern_dir2 = dir2 then
19                     interval_points.append(pattern_mid);
20                     interval_points.append(pattern_end);
21                     j ← j + 2;
22                 else
23                     break;
24             intervals.append(new Interval(interval_points));
25             i ← j-1;
26             continue;
27     i ← i + 1;
28 return intervals;
```

Regarding the choice of Different Choice for Possible Intervals in the specific algorithm4.6, we adopt the second option for selecting possible intervals, specifically: Collect all successive, identical diagonal-to-straight patterns and include all turning points within the possible interval. The implementation is shown in the pseudo code above.

6.2 Tie Breaking for A* Node

Algorithm 6.2.1: A* Node Comparison Strategy (Tie-breaking Rules)

```
Function CompareNodes(node_a, node_b):
    if |f(node_a) − f(node_b)| > ε then
        return f(node_a) > f(node_b);
    end
    if |h(node_a)| < ε then
        return false;                    // node_a is goal node, prioritize it
    end
    if |h(node_b)| < ε then
        return true;                     // node_b is goal node, prioritize it
    end
    if conflicts(node_a) ≠ conflicts(node_b) then
        return conflicts(node_a) > conflicts(node_b);
    end
    if |g(node_a) − g(node_b)| > ε then
        return g(node_a) < g(node_b);
    end
    return timestamp(node_a) > timestamp(node_b);
```

In multi-agent path planning, the node comparison strategy of the A* algorithm significantly impacts both search efficiency and solution quality. We have designed a multi-level tie-breaking strategy that prioritizes nodes with similar values in the following order:

1. **Comparison of $f(n)$ values:** The primary criterion is the evaluation function $f(n) = g(n) + h(n)$. Nodes with smaller $f(n)$ values have higher priority. A small tolerance value $\epsilon = 10^{-6}$ is used to account for floating-point precision errors.

2. **Goal node priority:** If two nodes have the same $f(n)$ value, priority is given to the node that has already reached the goal (i.e., $h(n) \approx 0$). This helps expedite search termination once a feasible solution is found. The same tolerance ϵ is used to determine whether $h(n)$ is effectively zero. This tie breaking rule prevents the situation where the end point is extended to the last due to the conflict count in Conflict Avoidance Table(CAT).

3. **Conflict-aware prioritization:** For nodes with equal $f(n)$ values that have not reached the goal, the node with fewer conflicts with other agents is prioritized. Conflict count are from CAT which calculates the number of times the current node is occupied by other agents in the solution at the current time. In our case, CAT is used in the way described in CBS paper[1]. This optimization is specifically designed for multi-agent scenarios, helping to reduce future conflict resolution overhead.

4. **Preference for higher $g(n)$ values:** When conflict counts are identical, the node with the larger $g(n)$ value is selected. This implicitly favors paths that have progressed further toward the goal. In scenarios with multiple similar paths, this criterion helps prioritize those that are closer to completion. Furthermore, in the `Discard`

`Jump Point in JPSCBS-S` section5.7.1, we previously discussed the rule for discarding jump points. To minimize unnecessary interactions with this rule, we leverage the preference for higher $g(n)$ values. Since diagonal movements result in a greater increase in $g(n)$ compared to straight-line movements when expanding the same point, our tie-breaking strategy inherently prioritizes diagonal paths. As a result, the path expansion process naturally avoids performing a final diagonal move that would trigger the ”ray condition” in the jump point discarding rule. Thus, by favoring higher $g(n)$ values, our method not only improves path selection efficiency but also indirectly reduces unnecessary jump point discards, ensuring a more consistent and efficient search process.

5. **Timestamp-based tie-breaking:** As the final criterion, priority is given to nodes with earlier timestamps. This ensures search determinism and promotes the expansion of nodes that were added to the open list earlier, thereby maintaining stability and predictability in the search process.

Experimental results demonstrate that this tie-breaking strategy effectively reduces the number of expanded nodes and prevents unnecessary discard of jump points, improving search efficiency. However, due to space limitations and the fact that this aspect is not the primary focus of our algorithm performance, we do not include a performance comparison before and after implementing this part in the experimental section.

7 Experimental Setup

A MAPF problem is defined by a graph along with a set of source and target vertices (referred to as maps and scenarios in the following discussion), as introduced in Chapter 2, Section 2.1. Consequently, a MAPF benchmark consists of a collection of graphs, each paired with multiple sets of source and target vertices. In this section, we describe the evaluation process of the algorithm.

7.1 Maps and Scenarios

The maps and scenarios used in our benchmark are sourced from movingai.com [16]. The detailed description of these maps and scenarios can be found in the paper [8]. There are several types of maps:

- **City maps.** These maps are derived from real-world urban layouts, including Berlin, Boston, and Paris. They contain structured roads and pathways simulating real city environments.
- **DAO (Dragon Age Origins) maps.** These are grids taken from the game Dragon Age Origin and are relatively large and open, with some maps as large as 1000×1000.
- **Open $N \times N$ grids:** These are uniform $N \times N$ grids, commonly with values of $N = 8, 16, 32$. Such grids facilitate experiments with high agent density, where the number of empty vertices is minimal.
- **Open $N \times N$ grids with random obstacles:** These are $N \times N$ grids in which a subset of cells is randomly selected as impassable obstacles [6].
- **Maze-like grids:** These grids feature complex interconnected corridors.
- **Room-like grids:** These grids simulate indoor spaces with rooms and corridors.
- **Warehouse grids:** Inspired by real-world autonomous warehouse applications, these maps are structured to simulate inventory shelving and robot navigation.

There are 33 maps and 25(x2) benchmark sets available for every map. Each benchmark file consists of a list of start/goal locations. The standard evaluation approach involves incrementally adding agents until an algorithm fails to solve the problem within a specified time or memory limit. There are two types of benchmark sets. One set of benchmarks is random scenarios that has problems that are generated purely randomly, capped at 1000 problems per file. The individual problems for a given agent will all tend to be longer. The other set of benchmarks is even scenarios that has problem that are evenly distributed in buckets of 10 problems with the same (length/4). These problems will have an even mix of short and long problems.

7.2 Benchmark Setup

The benchmark setup follows the specifications outlined in paper [8] and the webpage [17]. Our setting considers a MAPF problem on an 8-neighbor grid with the following conditions:

1. Edge, vertex, and swapping conflicts are forbidden.
2. Following and cycle conflicts are allowed.

3. The cost of diagonal movement is $\sqrt{2}$, the cost of straight-line movement is 1, and the cost of waiting action is 1.
4. The objective function is the sum of costs.
5. The agent behavior at target is disappear at target.
6. The runtime limit is set to 30 seconds. Exceeding the time or memory limit is considered a failure.
7. All experiments are conducted on an AMD Ryzen 7 5800H (single-threaded) with 16GB RAM.

We collect benchmark data based on the number of agents, including the success status, runtime, number of nodes expanded when a solution is successfully found, and solution cost. After gathering results for each scenario, we compute a statistical summary. Evaluation metrics include success rate, runtime, number of expanded nodes, and solution cost.

7.3 Evaluation Methodology

To assess the effectiveness of our proposed method, we compare it against Conflict-Based Search (CBS) with the *bypass* improvement. CBS is a highly efficient optimal MAPF solver and serves as a strong baseline for our evaluation.

Since the original CBS paper did not provide publicly available source code, we implemented CBS and the bypass improvement ourselves. All implementation details strictly adhere to the specifications outlined in the original paper, ensuring an accurate reproduction of the algorithm. For the bypass optimization, we employed *Bypass1*.

We select CBS with bypass optimization as the comparison algorithm for the following reasons:

1. JPSCBS already incorporates optimizations similar to bypass.
2. Other existing CBS improvements and variants, such as MA-CBS, Prioritizing conflicts, CBSH, CBSH2, and Disjoint Splitting, as discussed in Chapter 3, also can be adapted to optimize JPSCBS with minor modifications. Therefore, at this stage, we limit our comparison to the bypass-optimized version of CBS.

8 Result

In this chapter, we will analyze the benchmark results from three perspectives: success rate, the number of expanded nodes and runtime when the MAPF problem is successfully solved. This analysis aims to compare the performance differences between CBS and JPSCBS-S.

8.1 Success Rate

Due to space constraints and to maintain the flow of the main text, all benchmark result figures of success rate have been placed in Appendix A.

8.1.1 City Maps

In the tests conducted on three city maps—Berlin_1_256 (A.1), Boston_0_256 (A.2), and Paris_1_256 (A.3)—each map included 25 test scenarios for both the ”even” and ”random” distributions, totaling 50 scenarios per map. Across all test scenarios, JPSCBS consistently produced optimal results. Furthermore, as shown in the success rate comparison figure, JPSCBS achieved a significantly higher success rate than CBS while maintaining optimality.

8.1.2 Open N x N Grids

In Open $N \times N$ grids, where there are no obstacles, no intermediate nodes are stored in the JPSPath for localized conflict resolution. As a result, the efficiency of JPSCBS is theoretically comparable to that of CBS, with JPSCBS incurring only a minor additional computational overhead. Therefore, we selected only one map, `empty-16-16` (A.7), for testing, and the results confirmed this conclusion.

8.1.3 Open N x N Grids with Random Obstacles

In maps with randomly placed obstacles — (A.19,A.20,A.21,A.22), the performance of JPSCBS is comparable to that of CBS. We hypothesize that this is due to two primary reasons: (1) the relatively small map size, which limits the benefits of localized conflict resolution compared to longer paths, and (2) in scattered random obstacle maps, Jump Point Search tends to expand a large number of jump points, leading to reduced search efficiency. However, due to time constraints, we did not conduct additional tests to determine which specific part of JPSCBS is more time-consuming in random maps.

In 32×32 random maps, across 100 test scenarios on two different maps, more than half of the solutions obtained within 30 seconds were non-optimal. However, the cost difference between these solutions and the optimal ones did not exceed 0.5% in any case.

8.1.4 Maze-like Grids

In maze grids — (A.15,A.16,A.12,A.14,A.13). JPSCBS demonstrates a significant advantage over CBS, with a substantial improvement in success rates across all tested maps. This advantage can be attributed to the structured nature of maze environments, where the JPS-generated paths tend to produce more jump points along the path, which can be effectively utilized for localized conflict resolution. Except in small maps, where the advantage is less significant because the total path length is short, and the computational overhead of CBS calculating the entire path is not much higher than JPSCBS computing only segmented paths.

8.1.5 Room-like Grids

In room-like grids — (A.24),A.23), the structure of room grids is also highly suitable for JPSCBS. Due to the room-based layout, JPSCBS can generate jump points more evenly for local conflict resolution, leading to a significant improvement in success rates.

8.1.6 Warehouse Grids

In warehouse grids with narrow passages — (A.26,A.27, A.28, A.29), JPSCBS does not exhibit a significant advantage. This is likely due to two reasons: first, the relatively small map size, and second, the limited expandable space caused by the narrow corridors. Since CBS computes the entire path in such constrained environments, it quickly completes the search, reducing the comparative benefit of JPSCBS.

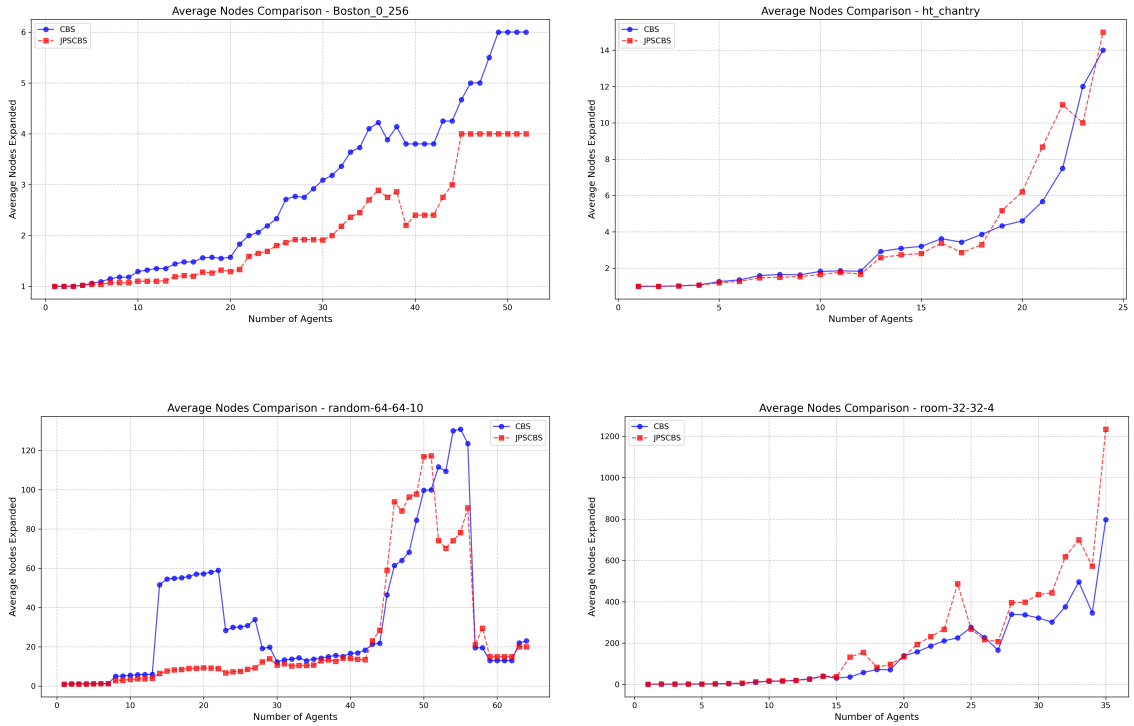
8.1.7 DAO (Dragon Age Origins) Maps

In the maps of games DAO and DAO2 - all other maps in Appendix, JPS-CBS demonstrates a significant advantage, which is hypothesized to be due to the structured nature of the maps and the clustering of obstacles. On larger maps, such as `orz900d` (A.17), `brc202d` (A.4), and `w_woundedcoast` (A.25), this advantage becomes even more pronounced. It is speculated that the disparity between the local conflict resolution path segments and the entire path is more substantial, thus contributing to JPS-CBS's superior performance on these maps.

8.2 Expanded Nodes

Since CBS fails too quickly with a small number of agents in many maps, the number of statistically meaningful samples for expanded nodes—where both methods successfully solve the same problem—is too low. Therefore, we selected only four maps for statistical comparison.

The data shows that CBS and JPSCBS have almost the the number of expanded nodes at almost the same level when they successfully solve the same problem.



8.3 Run Time

The statistics on runtime are similarly difficult to obtain due to the insufficient number of successful CBS samples in many maps. Additionally, the runtime of JPS-CBS is significantly influenced by factors such as the length of the specific path segments where conflicts occur, the surrounding obstacle configuration, and whether the next jump point needs to be abandoned. In contrast, CBS calculates the entire path each time, leading to relatively smaller differences. As a result, there can be substantial variations in runtime between JPS-CBS and CBS across different maps and scenarios. For example, on maze maps, JPS-CBS can be several hundred times faster in many cases. On DAO and city maps, it is typically 5 to several dozen times faster. On random maps, however, the difference in runtime is minimal.

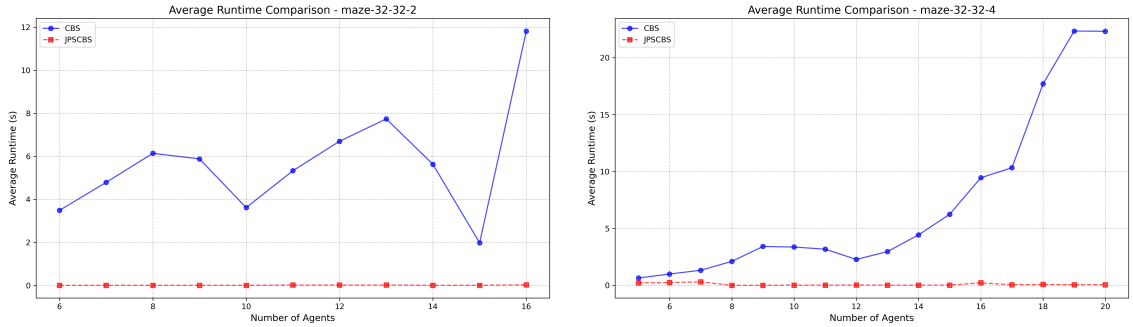


Figure 8.3: Maze-like Grids

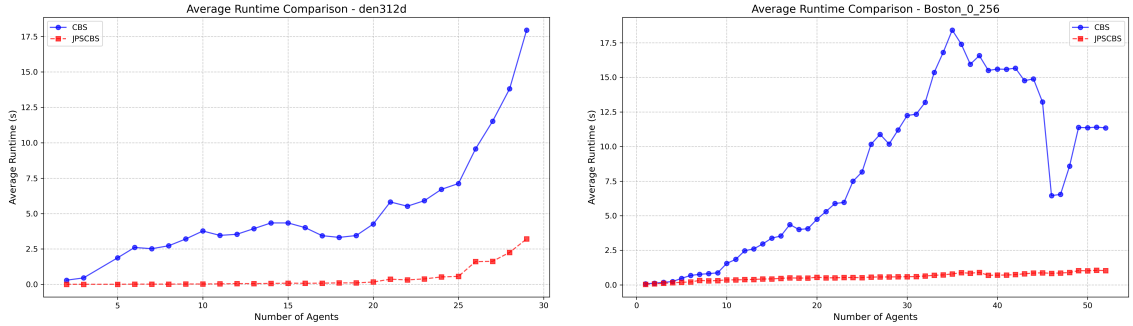


Figure 8.4: DAO and City Maps

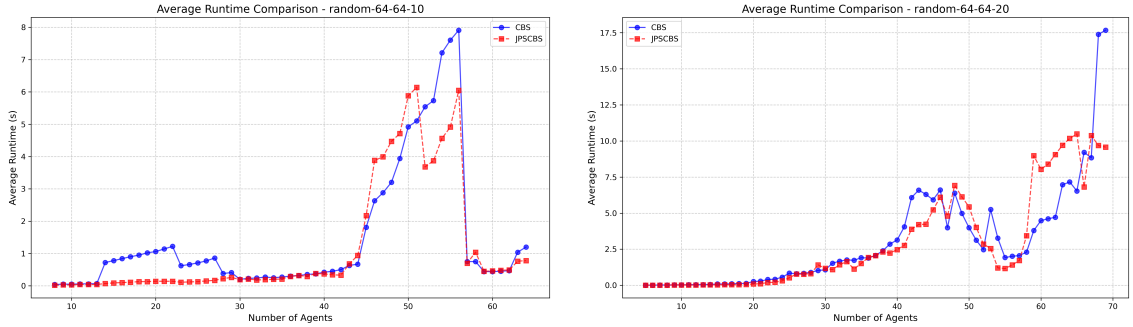


Figure 8.5: Open N x N Grids with Random Obstacles

9 Conclusion

In this thesis, we explored an enhanced approach to multi-agent pathfinding (MAPF) by integrating Jump Point Search (JPS) with Conflict-Based Search (CBS) to develop the JPSCBS algorithm. The proposed method aims to improve computational efficiency by leveraging localized conflict resolution rather than global path recalculations.

Our experimental results demonstrate that JPSCBS-S significantly enhances success rates and reduces computation time in most benchmark scenarios compared to standard CBS, at the cost of losing less than 0.5% of the cost in a low-conflict or middle-level-conflict environment. The localized conflict resolution mechanism effectively minimizes redundant searches while maximizing solution quality. Notably, JPSCBS shows substantial performance improvements in structured environments, such as maze-like and room-based grids, where jump points play a crucial role in path optimization. In large maps with regularly clustered obstacles, such as certain DAO maps, JPSCBS outperforms CBS due to its efficient local conflict resolution.

Beyond computational efficiency, the theoretical contributions of JPSCBS lie in its novel approach to handling conflicts. By utilizing JPS's pruning techniques within local conflict regions, JPSCBS ensures that conflict resolution remains constrained to affected areas, reducing unnecessary path modifications for agents not involved in conflicts. This feature makes JPSCBS a promising candidate for scaling MAPF to larger instances with high agent density.

Despite these improvements, certain challenges remain. In open grids with random obstacles, JPSCBS exhibits performance comparable to CBS, indicating that its benefits are most pronounced in structured environments. Additionally, the current approach immediately discards certain jump points, which may introduce inefficiencies in specific scenarios.

Overall, this work contributes to both theoretical advancements and practical applications in multi-agent pathfinding. By combining JPS's efficiency with CBS's optimal conflict resolution framework, JPSCBS provides a promising direction for improving MAPF solvers. The results of this research suggest that localized conflict resolution is a viable strategy for balancing optimality and efficiency in MAPF.

10 Future Work

This chapter outlines several promising directions for future research based on the findings and limitations of this study.

Completeness of JPSCBS-S

The completeness of JPSCBS-S has not yet been proven, and we aim to prove its completeness in future work.

JPSCBS-O

In our experiments, we observed that the non-optimality of JPSCBS-S arises because, during local conflict resolution, there are cases where choosing a later time to reach the start of the conflict path segment can result in a lower-cost solution. For example, if an agent encounters a constraint, the best local solution might be to wait, but waiting incurs a cost of 1. However, if the agent delays its arrival by a time t, it may avoid the constraint altogether. In this case, by modifying the preceding path where has a diagonal move to switch to two straight moves, the agent can reach the position later, with only a cost increase of $2 - \sqrt{2}$. Therefore, it may be possible to modify the algorithm to address this situation, potentially resulting in the optimal version of JPSCBS, which we refer to as JPSCBS-O.

Don't Discard Jump Points Immediately

Currently, JPSCBS-S discards certain jump points to ensure optimality. However, this strategy may lead to increased search overhead in some scenarios. A promising direction for future research is to investigate a variant of JPSCBS where jump points are not immediately discarded when required but instead retained temporarily to improve search efficiency. This approach may still generate near-optimal solutions while reducing search overhead. Future studies should analyze how different retention strategies impact performance and compare the trade-offs between solution quality and computational efficiency.

Integration with Other CBS Improvements

Several CBS variants, such as ICBS, CBSH2, and Disjoint Splitting, introduce heuristics and optimization techniques in constraint tree that reduce the search space and improve performance. Since JPSCBS still maintains the basic framework of the CBS constraint tree, exploring how JPSCBS can be combined with these improvements could lead to further advancements in JPSCBS efficiency. For instance, integrating JPSCBS with the dependency graph heuristic from CBSH2 could enhance high-level search prioritization, while combining it with Disjoint Splitting might further reduce redundant searches.

Performance Analysis and Improvement

Experimental results indicate that the performance of JPSCBS is comparable to CBS in maps with randomly placed obstacles. We hypothesize that this is due to two primary reasons: (1) the relatively small map size, which limits the benefits of localized conflict resolution compared to longer paths, and (2) in scattered random obstacle maps, Jump Point Search tends to expand a large number of jump points, leading to reduced search efficiency. Future work should involve a more detailed analysis of JPSCBS in large open

grids with randomly placed obstacles to identify the exact bottlenecks and potential optimizations.

By pursuing these research directions, future work can further refine and expand the capabilities of JPSCBS, making it more adaptable to a broader range of multi-agent pathfinding scenarios.

Bibliography

[1] Guni Sharon et al. “Conflict-Based Search for Optimal Multi-Agent Path Finding”. In: *Artificial Intelligence* (July 2012), pp. 563–569. DOI: 10.1016/j.artint.2014.11.006.

[2] Eli Boyarski et al. “ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding”. In: (July 2015), pp. 740–746.

[3] Ariel Felner et al. “Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding.” In: *International Conference on Automated Planning and Scheduling* (Jan. 2018), pp. 83–87. DOI: 10.1609/icaps.v28i1.13883.

[4] Max Barer et al. “Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem”. In: *Symposium on Combinatorial Search* (Aug. 2014), pp. 961–962. DOI: 10.3233/978-1-61499-419-0-961.

[5] Guillaume Sartoretti et al. “PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning”. In: *arXiv: Robotics* (Sept. 2018).

[6] Trevor Standley and Trevor Standley. “Finding Optimal Solutions to Cooperative Pathfinding Problems”. In: (July 2010), pp. 173–178.

[7] Trevor Standley et al. “Complete Algorithms for Cooperative Pathfinding Problems”. In: (July 2011), pp. 668–673. DOI: 10.5591/978-1-57735-516-8/ijcai11-118.

[8] Roni Stern et al. “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks.” In: *Symposium on Combinatorial Search* (Jan. 2019), pp. 151–159. DOI: 10.1609/socs.v10i1.18510.

[9] David Silver and David Silver. “Cooperative Pathfinding”. In: *Artificial Intelligence and Interactive Digital Entertainment Conference* (June 2005), pp. 117–122. DOI: 10.1609/aiide.v1i1.18726.

[10] Rina Dechter and Judea Pearl. “Generalized best-first search strategies and the optimality of A*”. In: *J. ACM* 32.3 (July 1985), pp. 505–536. ISSN: 0004-5411. DOI: 10.1145/3828.3830. URL: https://doi.org/10.1145/3828.3830.

[11] Mike Phillips et al. “SIPP: Safe Interval Path Planning for Dynamic Environments”. In: *IEEE International Conference on Robotics and Automation* (May 2011), pp. 5628–5635. DOI: 10.1109/icra.2011.5980306.

[12] Daniel Harabor et al. “Online Graph Pruning for Pathfinding on Grid Maps”. In: *AAAI Conference on Artificial Intelligence* (Aug. 2011), pp. 1114–1119. DOI: 10.1609/aaai.v25i1.7994.

[13] Guni Sharon et al. “The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding”. In: (July 16, 2011), pp. 662–667. DOI: 10.5591/978-1-57735-516-8/ijcai11-117.

[14] Jiaoyang Li et al. “Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search: Preliminary Results.” In: *Symposium on Combinatorial Search* (Jan. 2019), pp. 182–183. DOI: 10.1609/socs.v10i1.18481.

[15] Jiaoyang Li et al. “Disjoint Splitting for Multi-Agent Path Finding with Conflict-Based Search”. In: *International Conference on Automated Planning and Scheduling* 29 (July 2019), pp. 279–283. DOI: 10.1609/icaps.v29i1.3487.

[16] Nathan R. Sturtevant. *Moving AI Lab Benchmark Maps and Scenarios*. 2012. URL: https://movingai.com/benchmarks/.

[17] MAPF Community. *MAPF Benchmark Dataset*. 2022. URL: https://mapf.info/index.php/Data/Data.

A Appendix

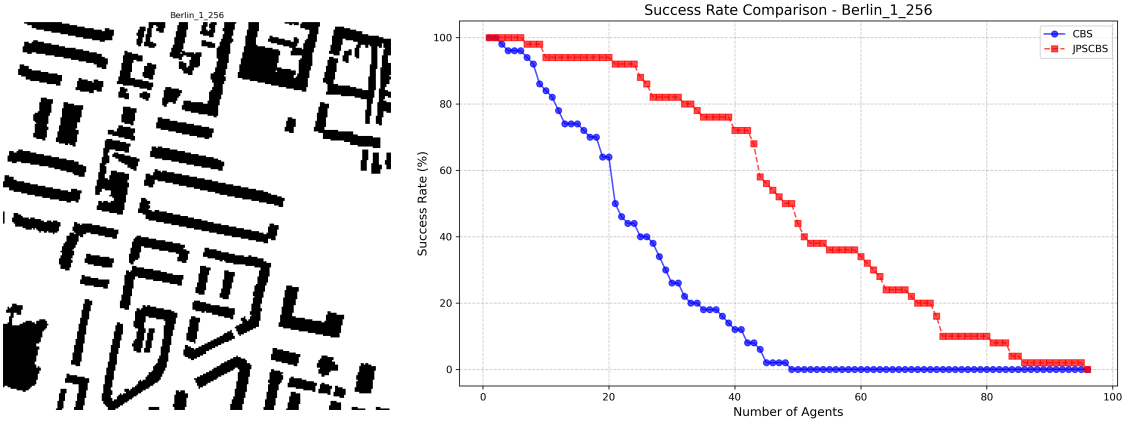


Figure A.1: Berlin_1_256

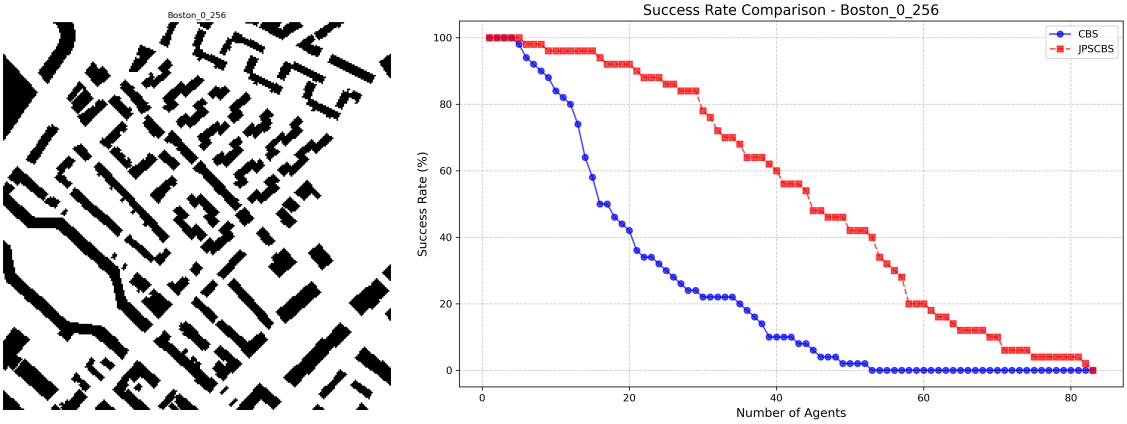


Figure A.2: Boston_0_256

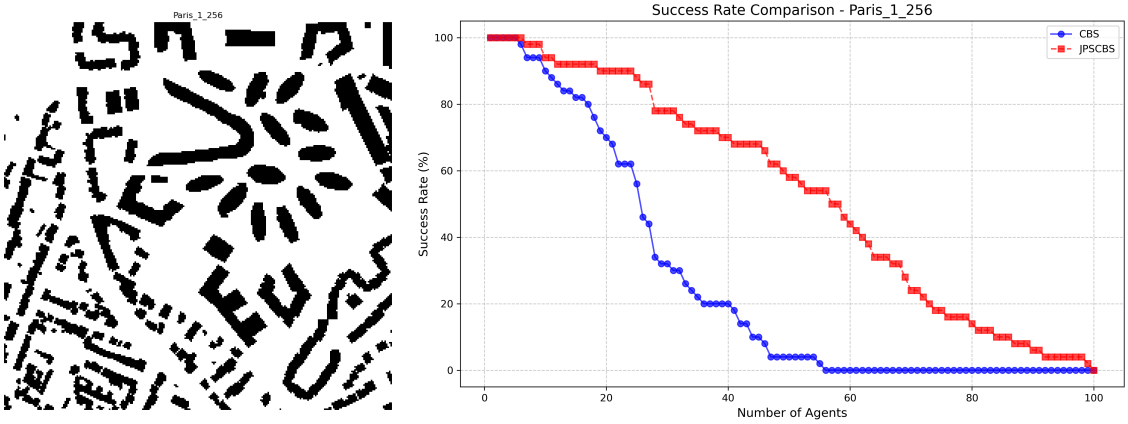


Figure A.3: Paris_1_256

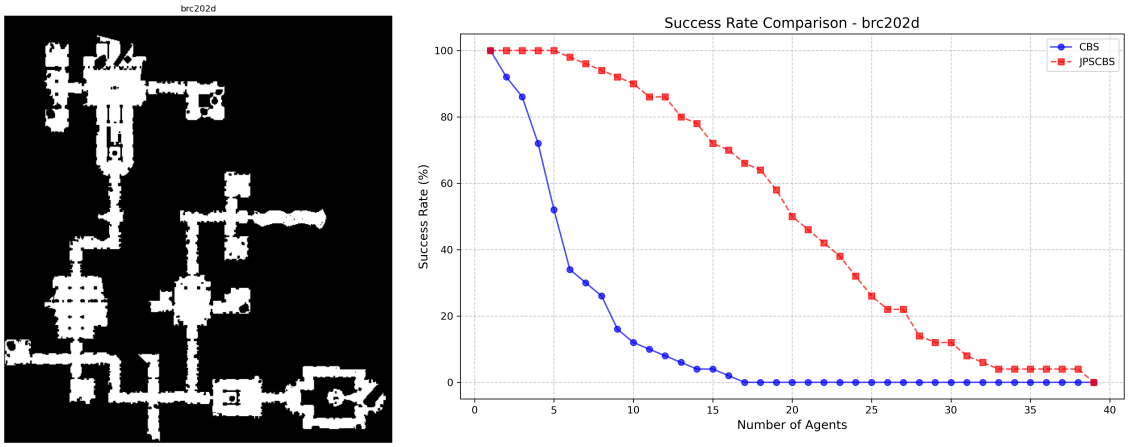


Figure A.4: brc202d

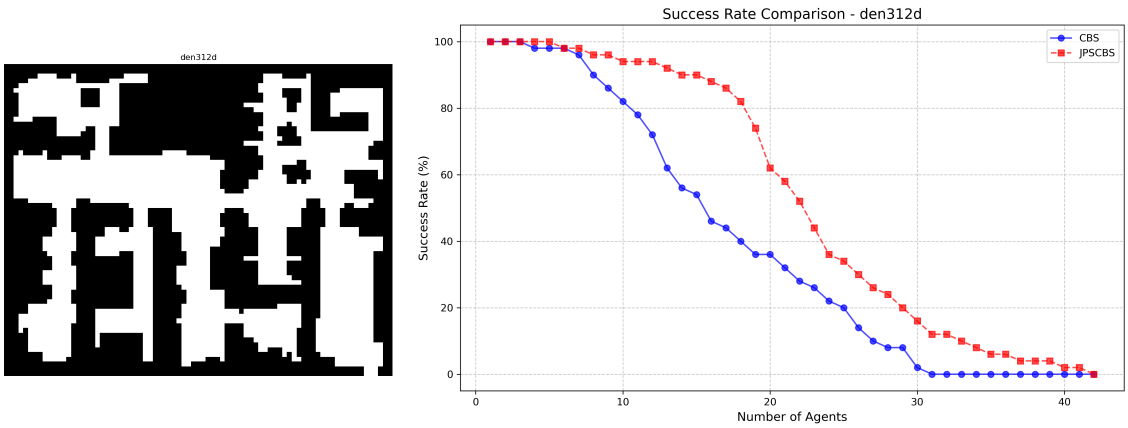


Figure A.5: den312d

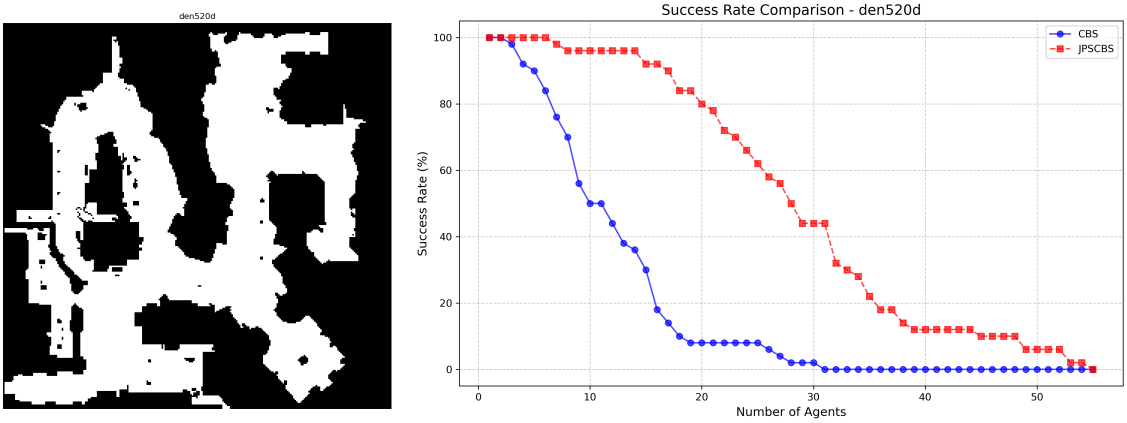


Figure A.6: den520d

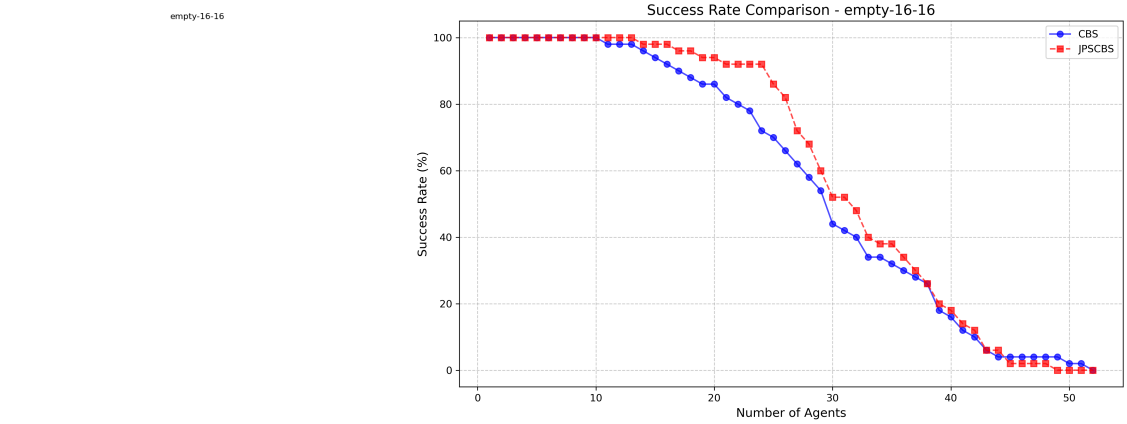


Figure A.7: empty-16-16

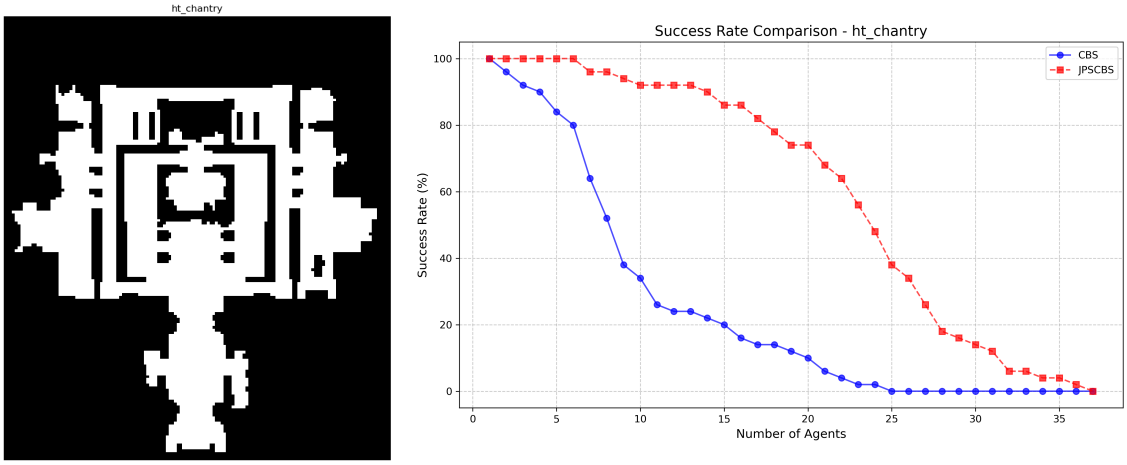


Figure A.8: ht_chantry

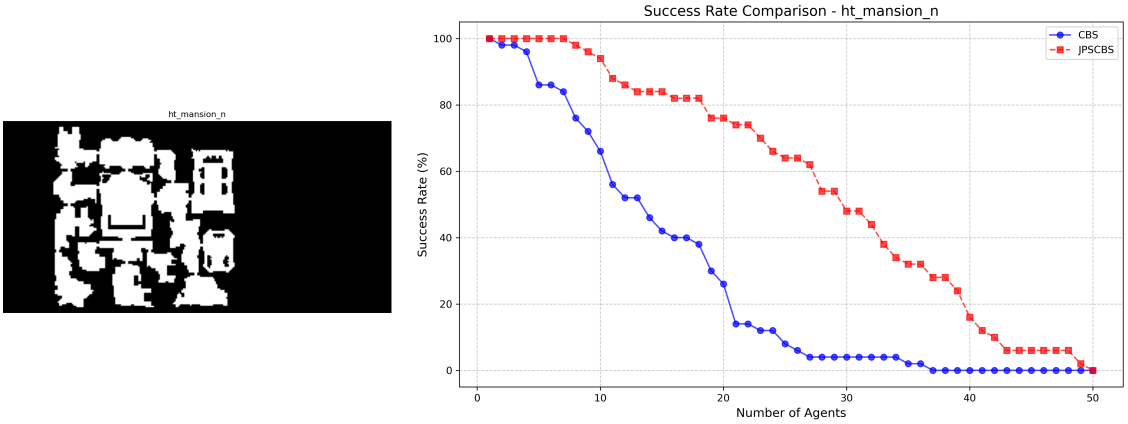


Figure A.9: ht_mansion_n

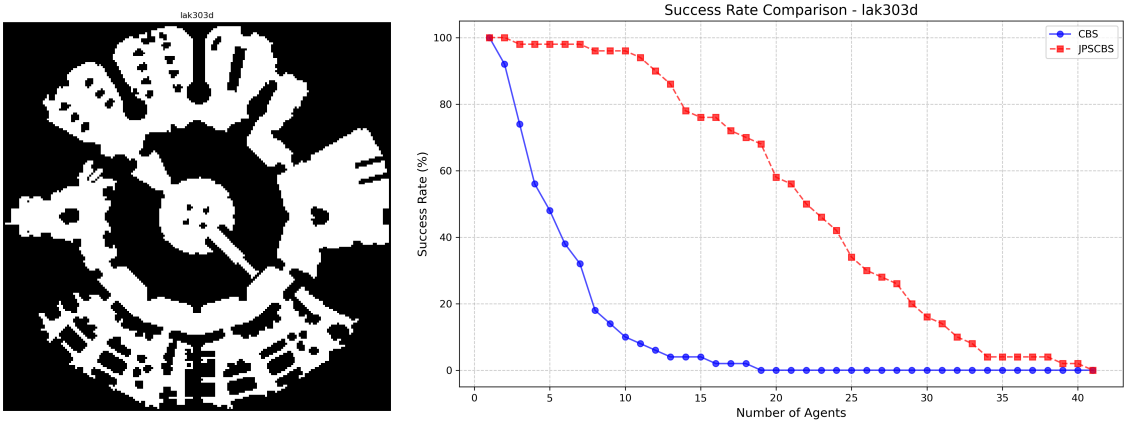


Figure A.10: lak303d

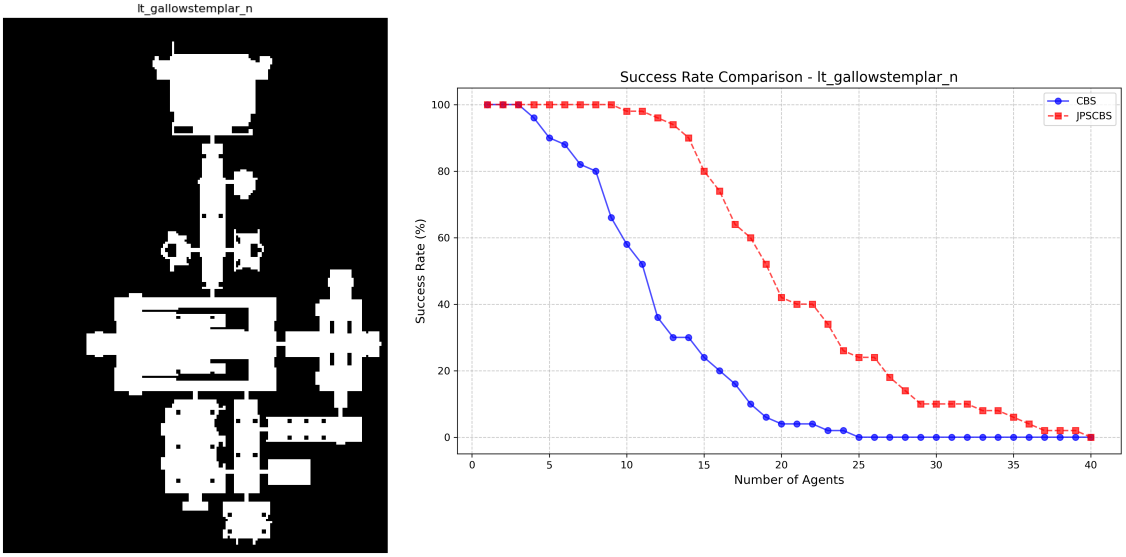


Figure A.11: lt_gallowstemplar_n

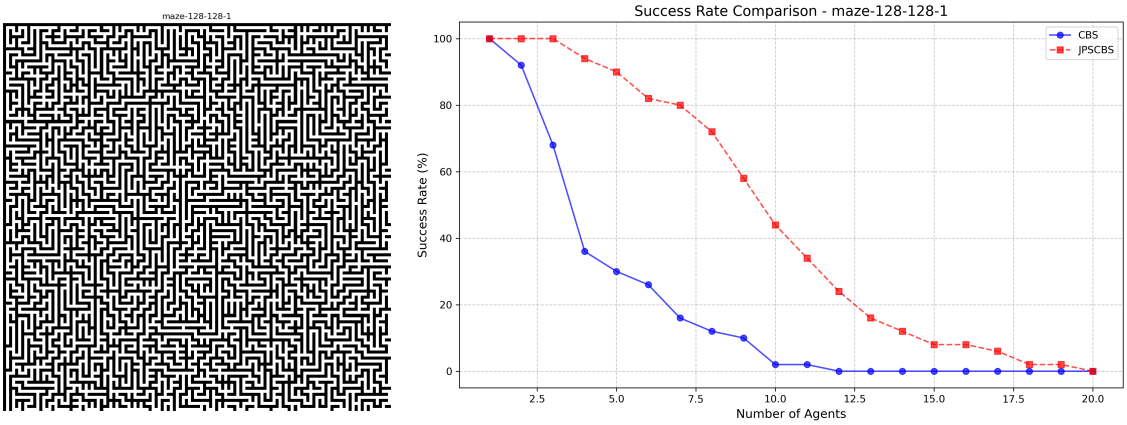


Figure A.12: maze-128-128-1

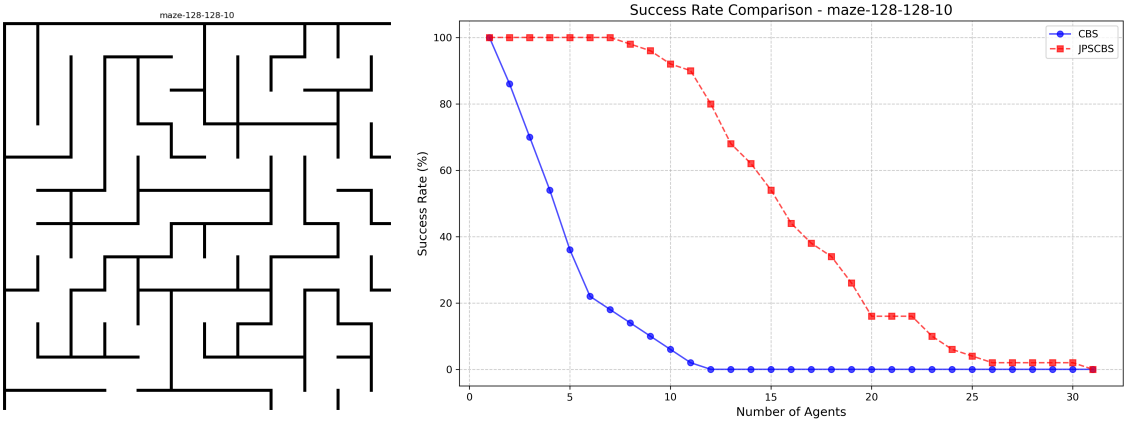


Figure A.13: maze-128-128-10

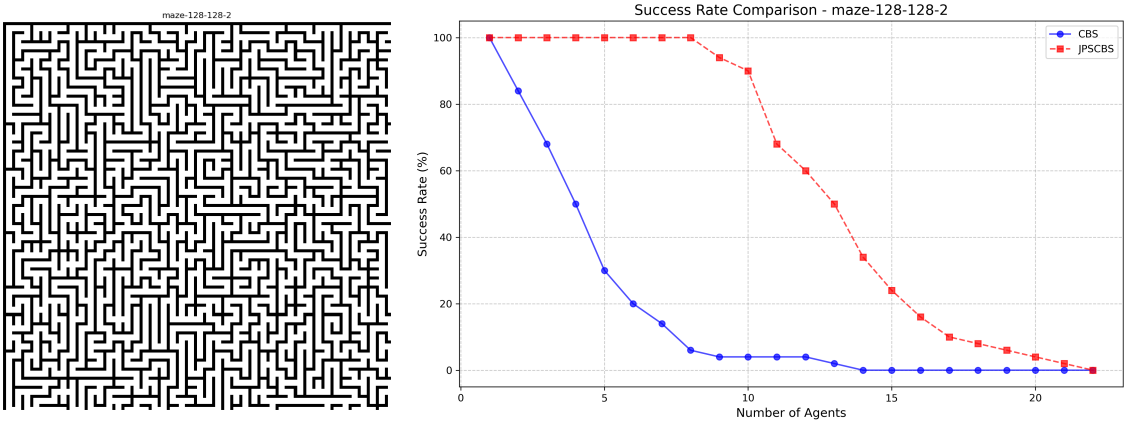


Figure A.14: maze-128-128-2

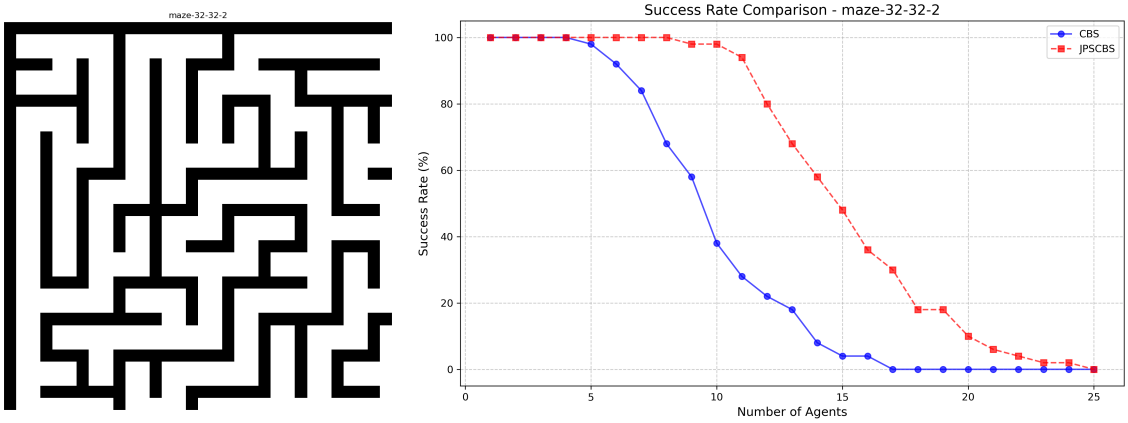


Figure A.15: maze-32-32-2

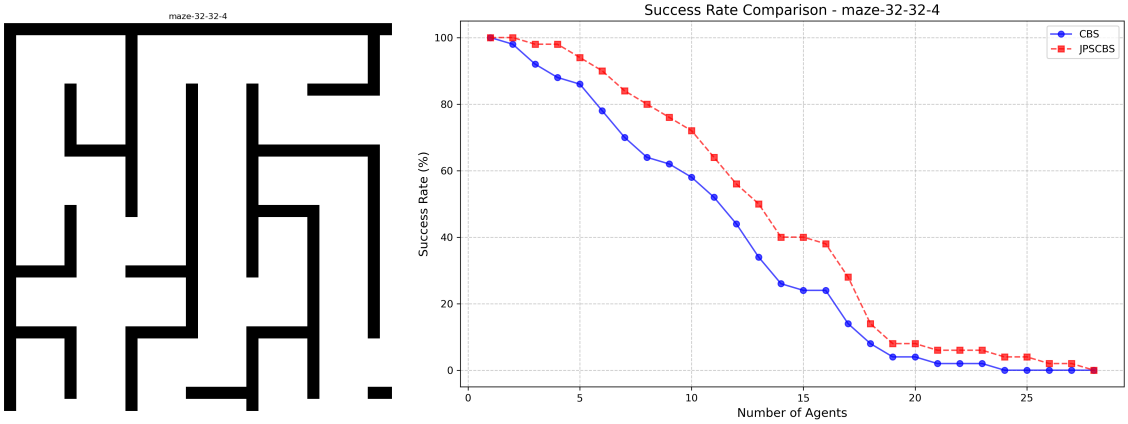
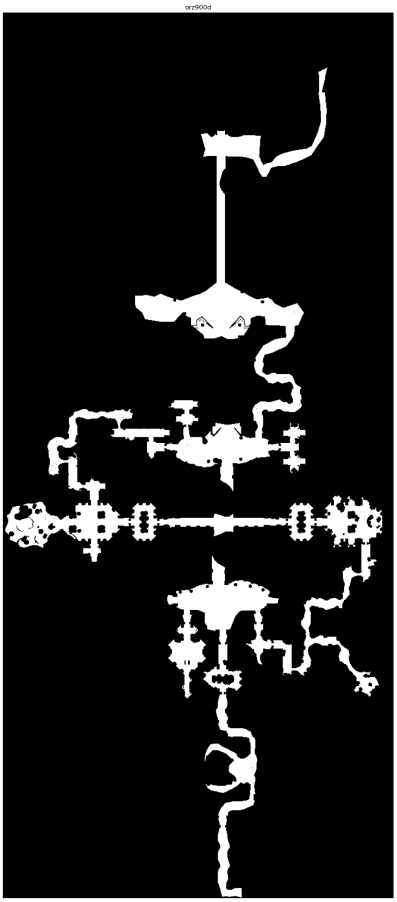


Figure A.16: maze-32-32-4



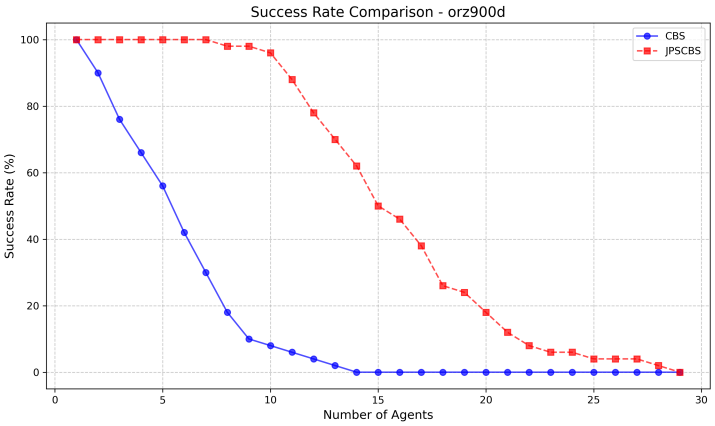
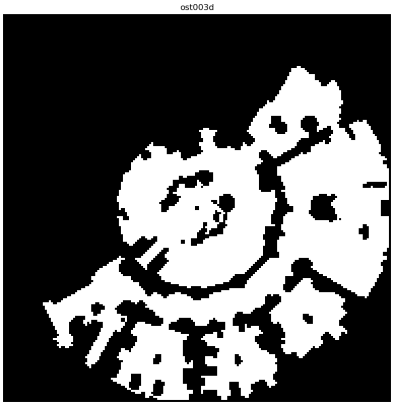


Figure A.17: orz900d



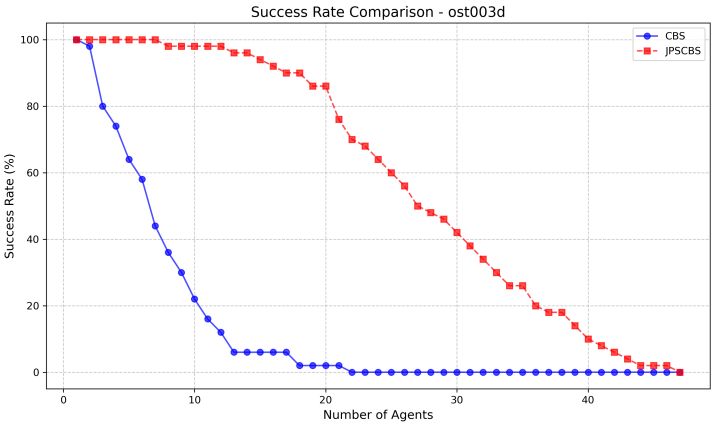


Figure A.18: ost003d

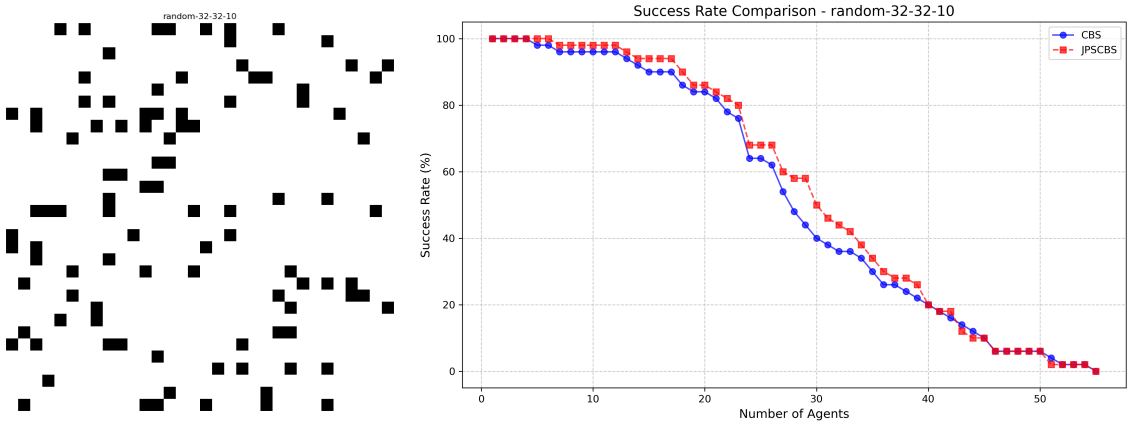


Figure A.19: random-32-32-10

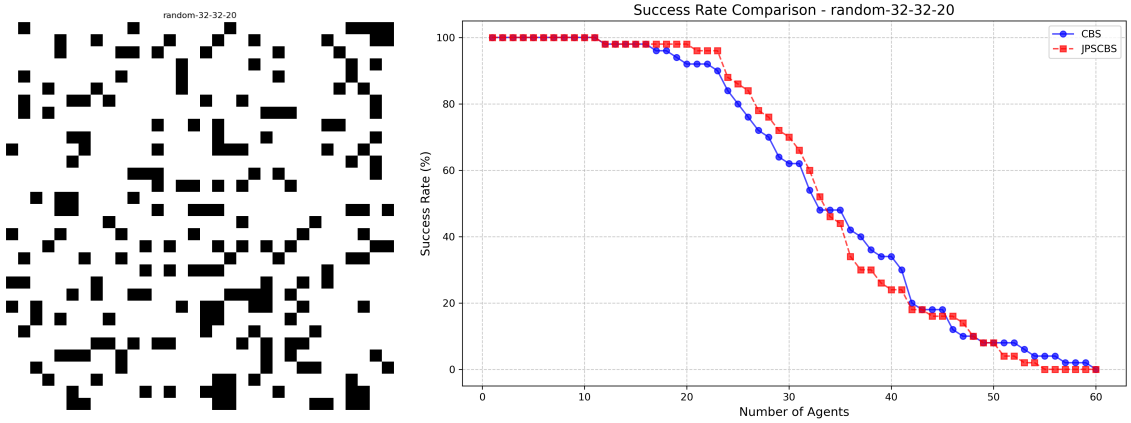


Figure A.20: random-32-32-20

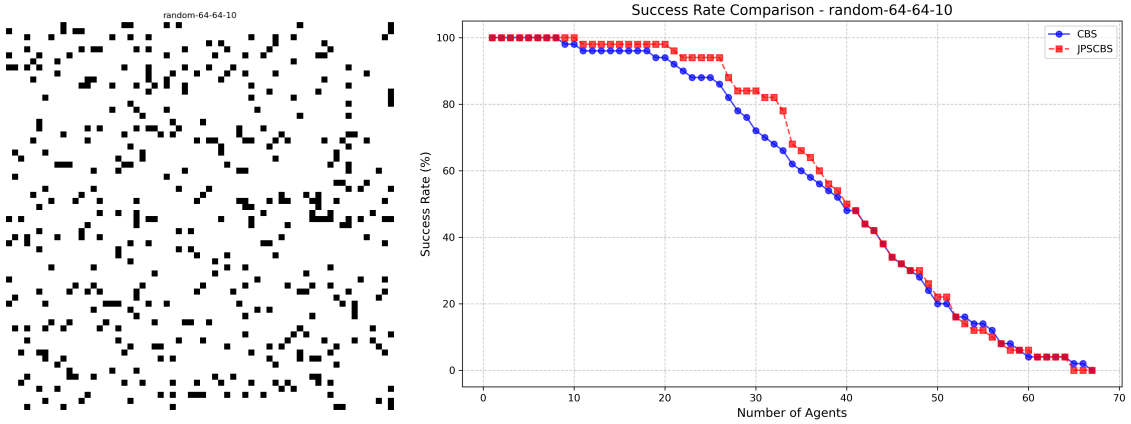


Figure A.21: random-64-64-10

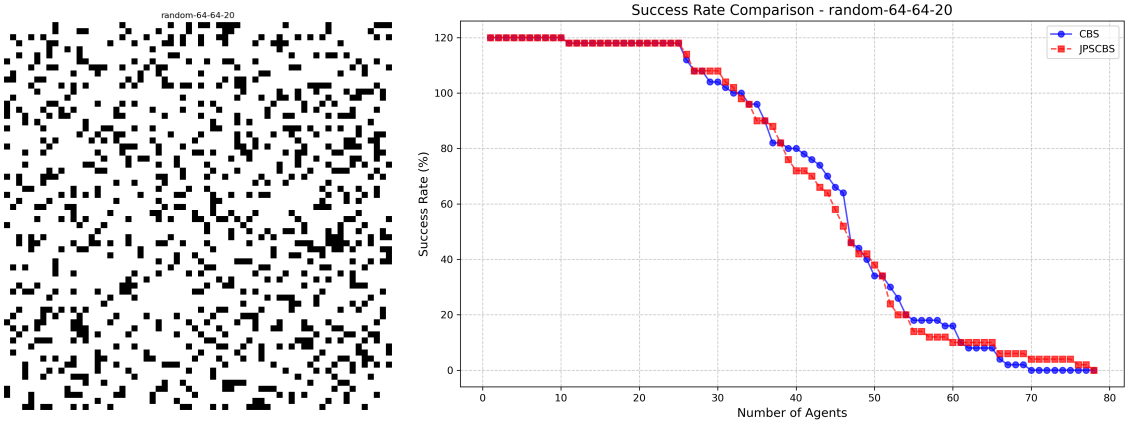


Figure A.22: random-64-64-20

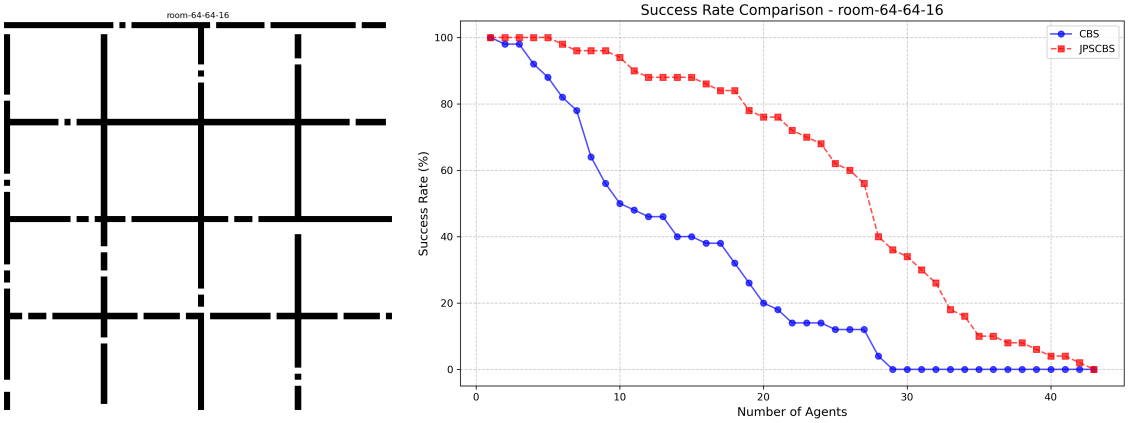


Figure A.23: room-64-64-16

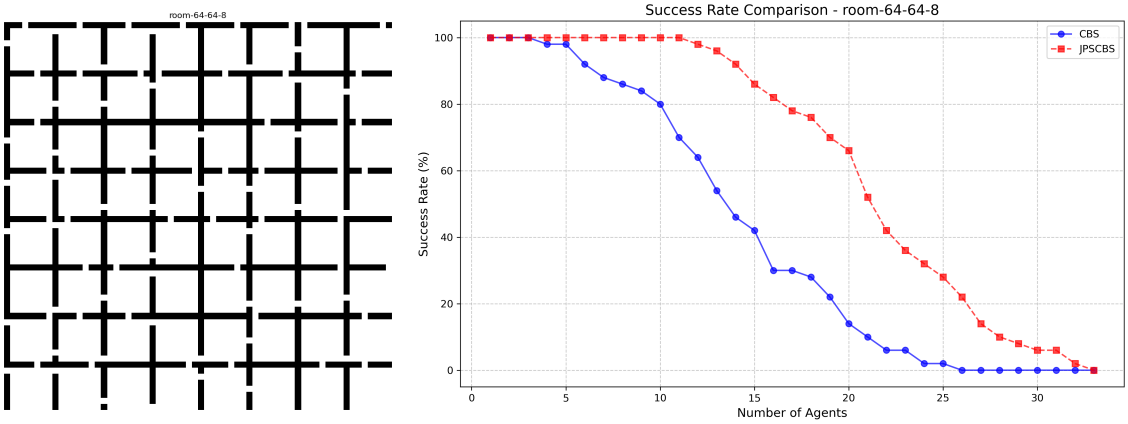


Figure A.24: room-64-64-8

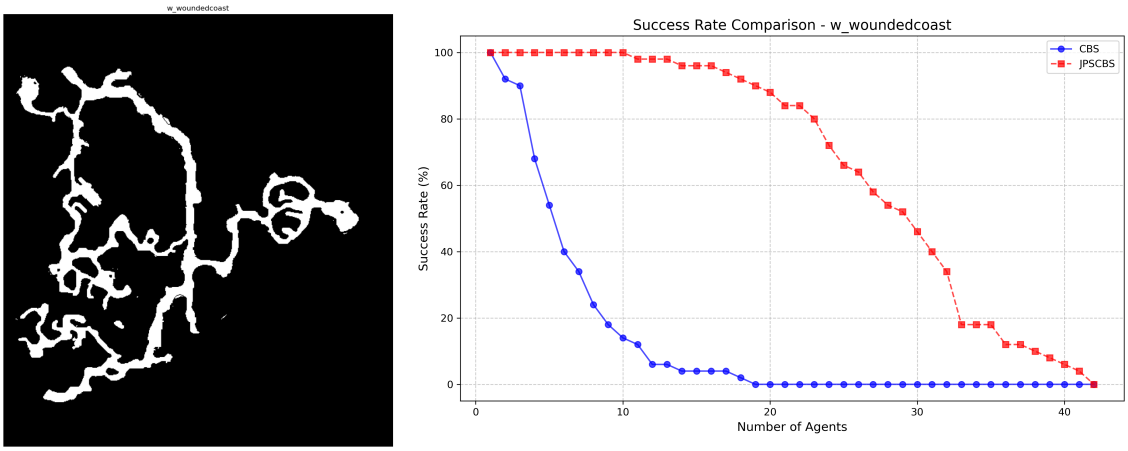


Figure A.25: w_woundedcoast

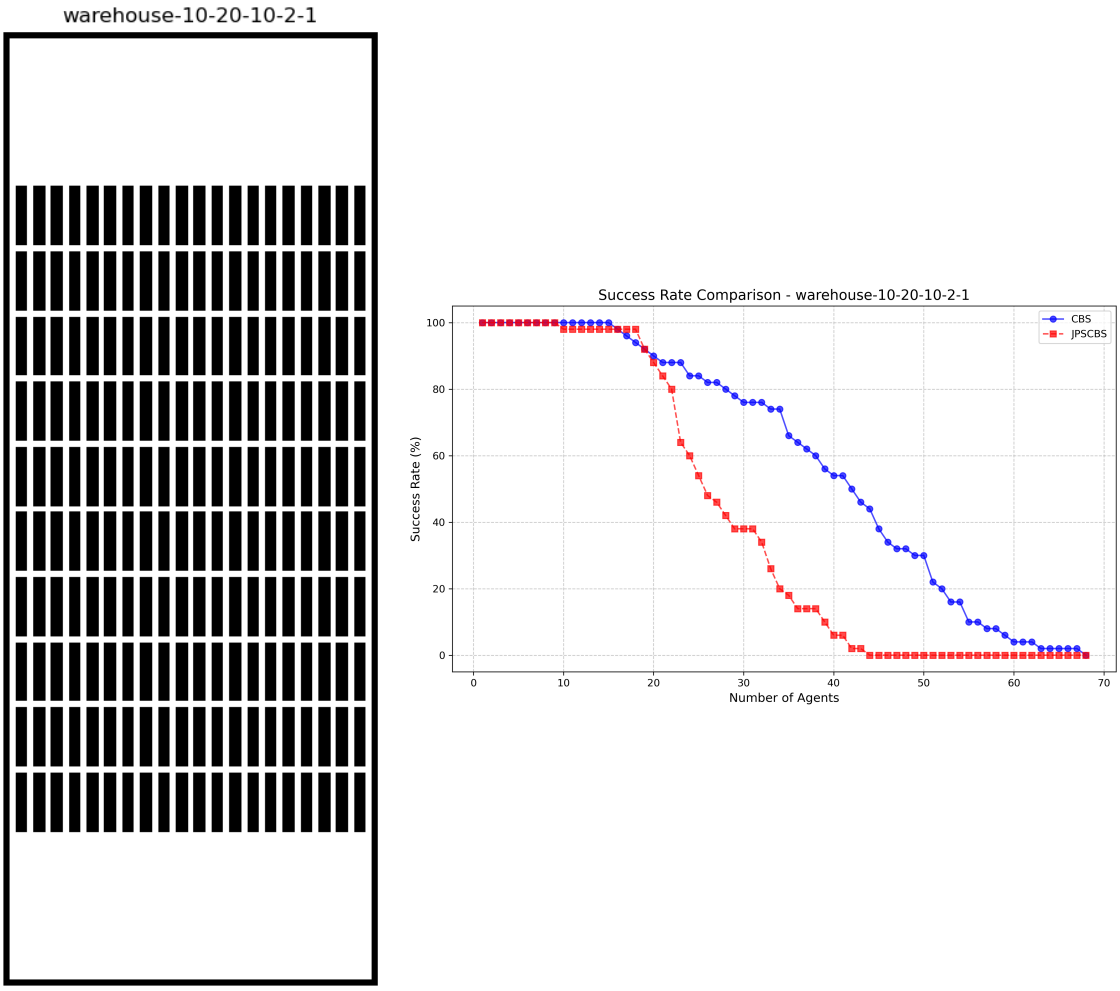


Figure A.26: warehouse-10-20-10-2-1

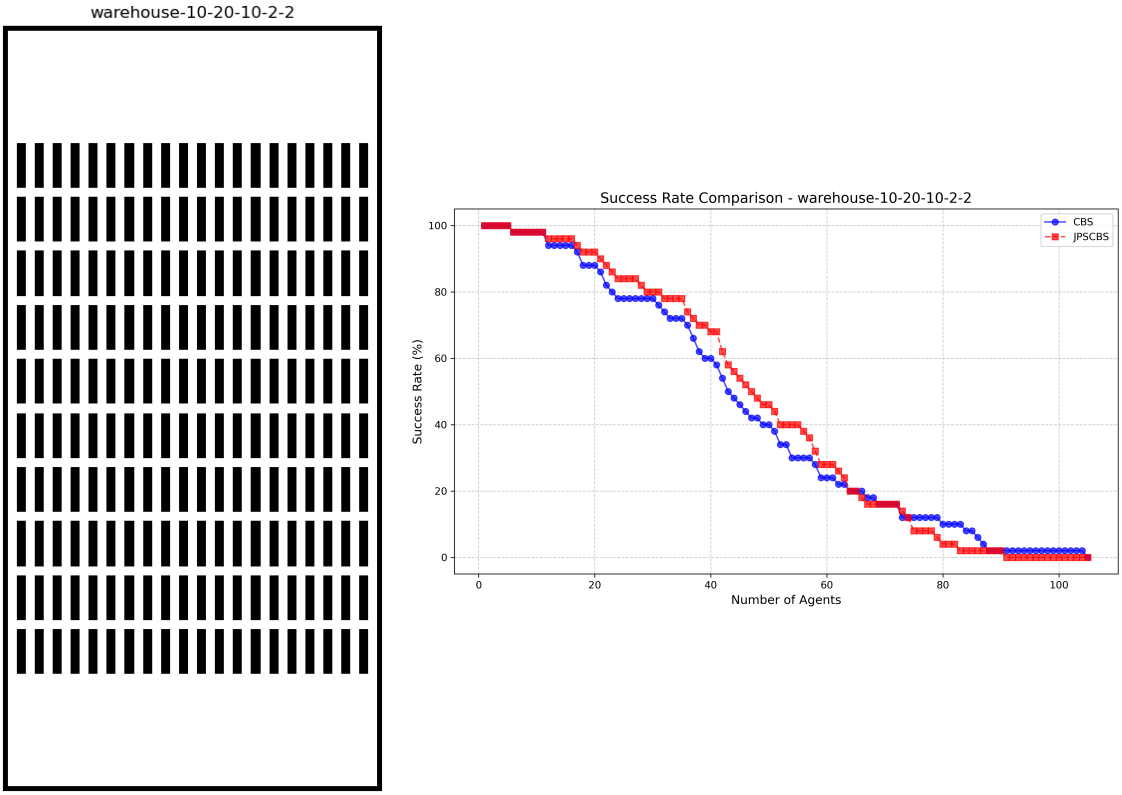
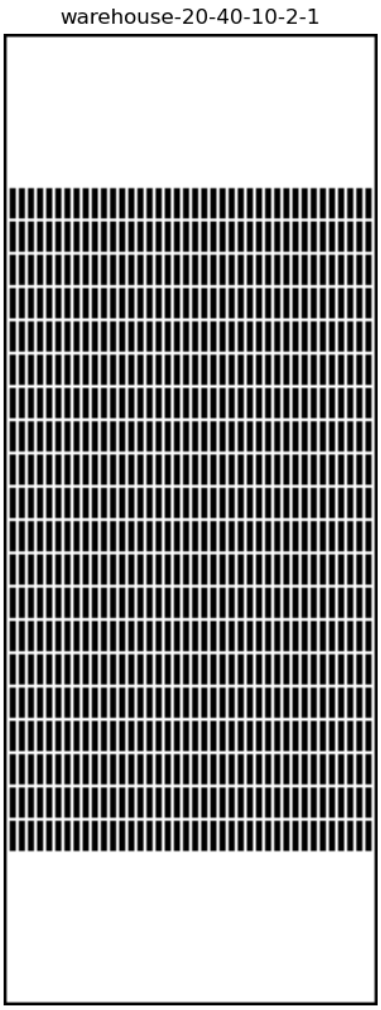


Figure A.27: warehouse-10-20-10-2-2



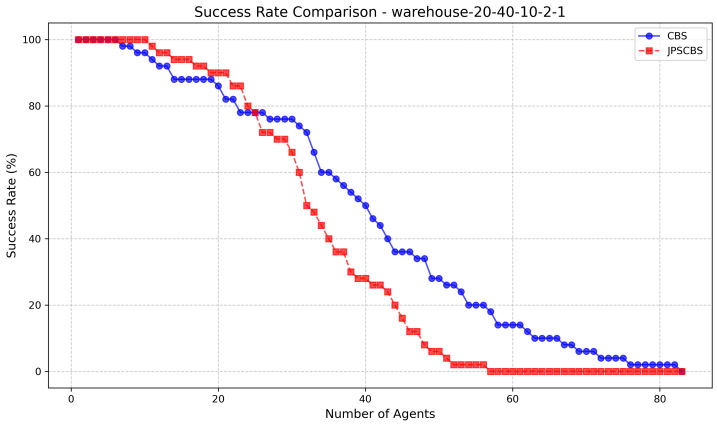


Figure A.28: warehouse-20-40-10-2-1

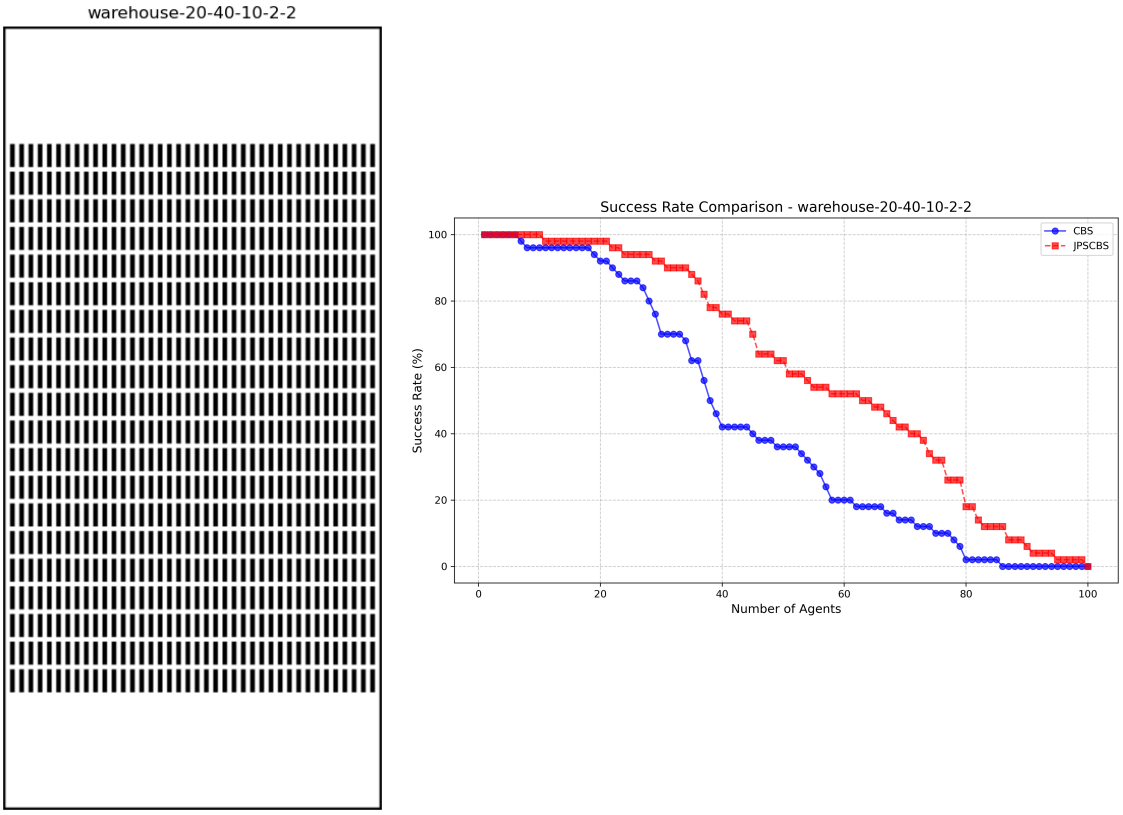


Figure A.29: warehouse-20-40-10-2-2

Technical
University of
Denmark

Building 324
2800 Kgs. Lyngby
Tlf. 4525 1700

www.compute.dtu.dk