Localized Conflict Resolution in Multi-Agent Pathfinding with Multi-Solution Jump Point Search

Anonymous submission

Abstract

Multi-Agent Pathfinding (MAPF) solvers, such as Conflict-Based Search (CBS) and Priority-Based Search (PBS), require complete path replanning for conflict resolution. When conflicts are frequent, repeatedly searching for complete paths becomes computationally expensive. To address this issue, this paper proposes a variant of Jump Point Search (JPS) that allows agent to search alternative paths as candidates, along with a localized conflict resolution strategy that resolves conflicts within specific segments between jump points. For evaluation, we introduce JPSCBS, a CBS variant that incorporates these improvements.

Introduction

MAPF (Stern et al. 2019) problem is specified by a graph $G = (V, E)$ and a set of k agents $\{a_1, \ldots, a_k\}$, where agent a_i has start location $s_i \in V$ and goal location $g_i \in V$. Time is assumed to be discretized, and in every time step, an agent can either move to an adjacent vertex or wait at its current vertex. A conflict happens when two agents occupy the same vertex or traverse the same edge in opposite directions at the same timestep. The objective is to find a set of conflict-free paths which move all agents from their start vertices to their goal vertices while minimizing the objective function of these paths.

Some MAPF solvers, such as Conflict-Based Search (Sharon et al. 2012) and its variants, as well as Priority-Based Search (Ma et al. 2019), resolve conflicts by adding constraints at a high level and replanning paths consistent with these constraints at a low level. The number of times an agent's entire path needs to be recalculated is exponential in the number of conflicts found between agents' paths. This global replanning strategy, while ensuring completeness and optimality, becomes computationally expensive in scenarios with frequent conflicts, particularly in large maps with obstacles.

To address these limitations, this paper proposes an approach that combines two key ideas. First, we introduce Muti-Solution Jump Point Search (MS-JPS), a variant of Jump Point Search (JPS) (Harabor and Grastien 2011) that can generate multiple candidate paths. Unlike traditional pathfinding methods that provide only a single path, MS-JPS can efficiently search for alternative sub-optimal solutions that can be utilized when dealing with conflicts. Second, we develop a localized conflict resolution strategy that handles conflicts within specific segments between jump points, avoiding the need for complete path replanning.

While these improvements can be applied to any MAPF algorithm that requires complete path recalculation or needs efficient local conflict resolution mechanisms, we demonstrate their effectiveness by integrating them into the CBS framework. The resulting new variant, JPSCBS, significantly reduces the computational overhead of conflict resolution while maintaining solution quality.

Background and Related Work

In this section, we provide a detailed overview of two algorithms: Jump Point Search (JPS), which serves as the theoretical foundation of our approach, and Conflict-Based Search (CBS), upon which our improvements are built.

Jump Point Search

Jump Point Search (JPS) is a pathfinding algorithm based on A* that efficiently search in undirected, 8-connected grid maps by eliminating symmetric paths through a set of pruning and jumping rules. (Harabor and Grastien 2014)

Definition 1 *A path $\pi = \langle n_0, n_1, \ldots, n_k \rangle$ is a cycle-free ordered path starting at node n_0 and ending at n_k. The set-minus operator in the context of a path: for example, $\pi \setminus x$, means that the subtracted node x does not appear on (i.e., is not mentioned by) the path.*

Pruning Rules: Given a node x, reached via a parent node p, we prune from the neighbours of x any node n for which one of the following rules applies:

1. There exists a strictly shorter path $\pi' = \langle p, y, n \rangle$ or $\pi' = \langle p, n \rangle$ than $\pi = \langle p, x, n \rangle$.
2. There exists a path $\pi' = \langle p, y, n \rangle$ with the same length as $\pi = \langle p, x, n \rangle$, but π' contains a diagonal move earlier than π.

We illustrate these rules in Figure 1(a) and 1(c). Observe that to test each rule we need to look only at the neighbours of the current node x. Pruned neighbours are marked in grey. Remaining neighbours, marked white, are called the *natural neighbours* of node x.

In Figure 1(b) and 1(d), we show that obstacles can modify the list of neighbours for x: when the alternative path

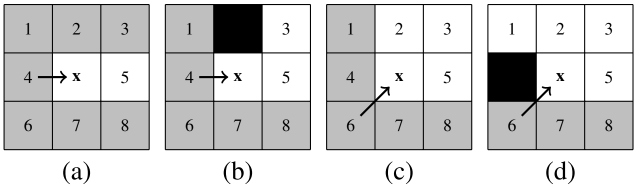


Figure 1: Pruning Rules

$\pi' = \langle p, y, n \rangle$ is not valid, but $\pi = \langle p, x, n \rangle$ is, we will refer to n as a *forced neighbours* of x.

Jumping Rules: Given a current node x heading in direction $\vec{d}$, JPS recursively applies the jump procedure as follows:

1. If an obstacle blocks further movement in direction $\vec{d}$, the jump terminates.
2. If the target node is reached, it is immediately identified as a jump point, and the jump terminates.
3. If a node with at least one forced neighbor is encountered, it is identified as a jump point, and the jump continues.
4. If $\vec{d}$ is diagonal, recursive jumps are performed in both straight directions that compose the diagonal. If a jump point is detected in either direction, the current node is identified as a jump point and the diagonal jump continues.

Starting from the initial node, JPS Expands the node with the lowest f-value from the open list. Apply pruning rules to identify natural and forced neighbours. For each remaining neighbour, JPS recursively applies jumping rules in the direction from the current node to that neighbour until a jump point is found or the jump terminates. Discovered jump points are added to the open list with their f-values computed as A* does. This process continues until either the goal is reached or the open list is exhausted.

Conflict-Based Search (CBS)

Conflict-Based Search (CBS) is a two-level optimal algorithm for solving MAPF problem. It consists of the following components:

Low-Level: Each agent computes an optimal path individually using a single-agent pathfinding algorithm, subject to the constraints imposed by the high-level search.

High-Level: CBS maintains a *Constraint Tree* (CT). The high-level search explores the CT in a best-first manner to resolve conflicts between agents. Each node N in the constraint tree consists of:

- A set of constraints N.constraints, where each constraint is a tuple $\langle a_i, v, t \rangle$ or $\langle a_i, u, v, t \rangle$ prohibiting agent a_i from occupying vertex v or traversing edge (u, v) at timestep t.
- A solution N.solution containing paths for all agents that satisfy N.constraints.
- The cost N.cost, typically the sum of cost.

Conflict Resolution in CBS During CT node processing, CBS finds conflicts by checking timesteps sequentially from t = 0. Upon detecting the first conflict, CBS performs a *split* operation, generating two child nodes with additional constraints. For a vertex conflict $\langle a_i, a_j, v, t \rangle$, one child adds constraint $\langle a_i, v, t \rangle$, the other adds constraint $\langle a_j, v, t \rangle$. For each child node, CBS invokes the low-level search to find new paths for the constrained agents. This process continues until a conflict-free solution is found.

While CBS guarantees optimality, in worst-case scenarios, the number of conflicts can grow exponentially. This limitation becomes particularly pronounced in dense environments or when dealing with large numbers of agents. Also, each CT node requires solving individual shortest path problems for affected agents. When conflicts are frequent, CBS repeatedly invokes low-level search, significantly increasing runtime.

CBS Variants and Other MAPF solvers Several notable enhancements to CBS have been proposed to improve its computational efficiency. Improved CBS (ICBS) (Boyarski et al. 2015b) reduces the number of expanded CT nodes through conflict bypassing and cardinal conflict prioritization. CBS with Heuristics (CBSH) (Felner et al. 2018) incorporates admissible heuristics to guide the high-level search more effectively. Enhanced CBS (ECBS) (Barer et al. 2014) trades optimality for efficiency by employing focal search to find bounded-suboptimal solutions.

Other than CBS and its variants, Priority-Based Search (PBS) (Ma et al. 2019) assigns priorities to agents and resolves conflicts by enforcing these priorities during planning. Large Neighborhood Search for MAPF (MAPF-LNS2) (Jiaoyang Li et al. 2022) iteratively refines suboptimal solutions by selecting a subset of agents and re-planning their paths locally, making it particularly effective for large-scale problems.

However, these approaches still have limitations. While CBS variants reduce CT node expansions through various strategies and PBS effectively reduces the search space, they all require complete path re-planning for the affected agents. Although MAPF-LNS2's local repair strategy is effective, its local range selection relies on factors such as the conflict's location, the affected time steps, heuristic-based selection of impacted agents, and iterative optimization, without taking the underlying map structure into account for determining path re-planning regions. These limitations motivate our approach in grid map with obstacles, which resolves conflicts locally and utilizes jump points as local ranges. This approach leverages the map's geometric properties to identify natural boundaries for path refinement, enabling more efficient and targeted conflict resolution.

Multi-Solution Jump Point Search

We propose Multi-Solution Jump Point Search (MS-JPS), an extension of JPS that allows efficiently further search for sub-optimal alternative paths. MS-JPS introduces two key modification: (1) a state-preservation mechanism that enables continuous search for alternative paths, (2) the identification of specific jump points for conflict resolution.

Algorithm 1: Multi-Solution Jump Point Search

Input: $start$, $goal$, $grid$, $state$
$closed \leftarrow \emptyset$;
$temp\_nodes \leftarrow \emptyset$;
while $state.open \neq \emptyset$ **do**
  $current \leftarrow$ best node in $state.open$;
  $closed \leftarrow closed \cup \{current\}$;
  $state.closed \leftarrow state.closed \cup \{current\}$;
  if $current.pos = goal$ **then**
    $state.open \leftarrow state.open \cup temp\_nodes$;
    return ReconstructPath($current$) ;
  $successors \leftarrow$ IdentifySuccessors($current$) ;
  foreach $succ \in successors$ **do**
    $cost \leftarrow$ GetMoveCost($current.pos, succ$) ;
    $g \leftarrow current.g + cost$;
    $h \leftarrow$ Heuristic($succ, goal$) ;
    $node \leftarrow$ CreateNode($succ, g, h, current$) ;
    if $succ \in closed$ **then**
      $temp\_nodes \leftarrow temp\_nodes \cup \{node\}$;
    else
      $state.open \leftarrow state.open \cup \{node\}$;
return $\emptyset$;

Algorithm Description

Each agent maintains a search state, consisting of:

- **An agent-specific open list** ($state.open$): Stores unexpanded nodes across searches.
- **An agent-specific closed list** ($state.closed$): Stores permanently expanded nodes.

The handling of the open list and closed list differs from standard A* search. Specifically, if a node does not exist in $state.open$ or close list in current search, it is added to $state.open$. If a node has already been expanded in current search, it is not considered in this round of search and temporarily stored in a separate set for future search. This is because we only need to identify the best path under the given state. Once the current search finishes, all nodes in the temporary set are added back to $state.open$. This mechanism allows MS-JPS to incrementally explore alternative paths without restarting from the beginning.

In the first search, we initialize the agent's search state with $state.open$ containing only the start node and an empty $state.closed$, and pass this state to MS-JPS. In subsequent searches, we reuse the search state maintained from the previous search to continue pathfinding. The full algorithm is detailed in Algorithm 1.

Possible Interval

Consider a pathfinding problem as shown in Figure 2, where multiple optimal paths exist with same costs. These paths differ only in their sequence of diagonal and horizontal movements. JPS only selects a single representative path (following the diagonal-first rule) from the start vertex (green) through an intermediate jump point (cyan) to the goal vertex (red), while others are discarded. These path intervals, where symmetric paths are pruned during JPS exploration, can serve as alternative routes in conflict resolution.

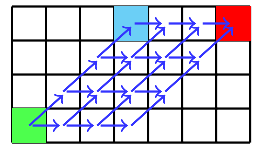


Figure 2: Example of Symmetric Paths

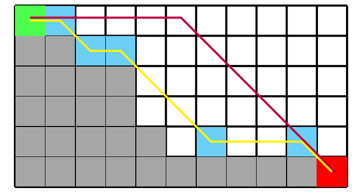
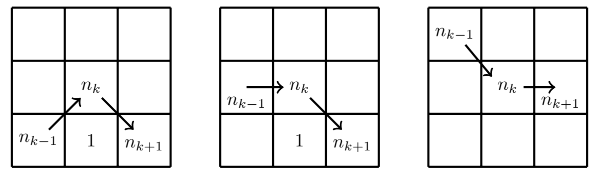


Figure 3: Example of Possible Interval



Case(a) Case(b) Case(c)

Figure 4: Turning Point

There are three possible types of turns at a turning point:

1. Diagonal-to-Diagonal (Figure 4 Case A),
2. Straight-to-Diagonal (Figure 4 Case B),
3. Diagonal-to-Straight (Figure 4 Case C).

Other turning points, such as Straight-to-Straight, are trivially suboptimal and are thus not considered, following the pruning rules. Let $\pi = \langle n_{k-1}, n_k, n_{k+1} \rangle$ be the current path segment in following discussion.

Turning Point Cases (a) and (b) In Case (a) and Case (b), n_{k+1} is not pruned according to pruning rule 1 (see Figure 1 (b) and (d)). It means that a node n_{k+1} is preserved if and only if $\nexists \pi' = \langle n_{k-1}, y, n_{k+1} \rangle$ or $\pi' = \langle n_{k-1}, n_{k+1} \rangle$ where cost(π') $<$ cost(π). Consequently, n_k must be retained as an essential turning point to ensure optimality.

Turning Point Case (c) In Case (c), n_{k+1} is not pruned according to pruning rule 2 (see Figure 1 (c)). This means that we may find $\pi' = \langle n_{k-1}, y, n_{k+1} \rangle$ with the same length as π. This leads to our definition of possible intervals: n_{k-1} is the interval start point, n_k is the discardable turning point, n_{k+1} is the interval end point, the triplet (n_{k-1}, n_k, n_{k+1}) is a possible interval.

Different choices of Possible Intervals

While our previous analysis identified possible intervals in the form of (n_{k-1}, n_k, n_{k+1}) for Case (c) turning points, the actual scope for cost-equivalent paths may extend beyond these boundaries (Figure 3). This extension occurs because the removal of n_k alters the nature of subsequent jump points, invalidating their pruning rules. To determine the interval scope, we analyze two possible scenarios:

Case 1 Consider a configuration where the post-n_k turning point follows a diagonal direction different from the n_{k-1} to

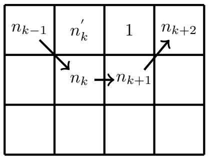


Figure 5: Case 1

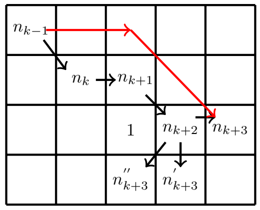


Figure 6: Case 2

n_k vector (Figure 5). When n_k is eliminated, the diagonal-first rule is no longer followed, allowing arbitrary permutations of diagonal and straight movements in the path from n_{k-1} to n_{k+1}. However, the potential parent nodes of n_{k+1} only have two possibilities: n_k and $n_k^{'}$. In both cases, it is impossible to reach n_{k+2} with a path of equal or lower cost without traversing through n_{k+1}. Therefore, n_{k+1} must be retained to maintain path optimality in this case.

Case 2 When n_{k+2} mirrors n_k's diagonal-to-straight pattern with identical directional components (Figure 6), we identify three potential turning directions at n_{k+2}. We first analyze n_{k+3}, where the path segment from n_{k-1} to n_{k+3} potentially admits a cost-equivalent alternative that bypasses both n_k and n_{k+2} (illustrated by the red line). Then we check another two potential directions n'_{k+3} and n''_{k+3}. For both vertices, it is impossible to be reached with a path of equal or lower cost without traversing through n_{k+1}.

Interval Selection Strategies Based on our analysis, we propose three strategies for defining possible intervals:

1. Maintain the basic form (n_{k-1}, n_k, n_{k+1}), containing jump points in one diagonal-to-straight pattern.
2. Include all successive, identical diagonal-to-straight patterns in a single interval $(n_{k-1}, n_k, n_{k+1}, \ldots, n_{k+i})$.
3. Include a predetermined number (k) of successive patterns. For example, with $k = 2$, intervals take the form $(n_{k-1}, n_k, n_{k+1}, n_{k+2}, n_{k+3})$.

The performance of each option is influenced by the specific map's size, as well as the shape and distribution of obstacles.

Reconstruct Path

For the purpose of resolving conflicts locally between jump points, when MS-JPS finds a path, we return not only the found path but also all the jump points along the path and all the possible intervals on the path.

Jump points here refer exclusively to the turning points expanded by MS-JPS that appear in the final path, and they are stored in the order in which they were expanded during the search. Furthermore, all intermediate turning points within a possible interval are removed, retaining only the start and end vertices of each interval.

JPSCBS

To validate the feasibility of our improvement, we integrate it with CBS and propose a new algorithm: CBS with JPS (JPSCBS). To demonstrate that our improvement can be integrated with other CBS enhancements, we incorporate it alongside the bypass improvement in our algorithm.

Algorithm 2: High-Level JPSCBS

```
1  Initialize R.solution and backups with MS-JPS;
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;
```

High-Level

In this section, we describe the high-level process of JPSCBS, a modified version of CBS high-level that integrates MS-JPS.

The Constraint Tree The overall structure of JPSCBS remains similar to CBS, which searches a constraint tree (CT). However, modifications are made to adapt it to MS-JPS. The solution of a CT is changed from a set of paths into a set of priority queues, where each queue stores paths originally from the MS-JPS. The highest-priority path is selected as the agent's plan, while alternative paths serve as candidates.

A node N in the CT is a goal node when the set paths each for an agent retrieved from priority queues in N.solution are valid, i.e., conflict-free. The high level performs a best-first search on the CT, prioritizing nodes by cost (Sharon et al. 2012). In case of ties, preference is given to CT nodes with fewer conflicts, further breaking ties in a First-In-First-Out (FIFO) manner.

Processing a Node in the CT The high-level process of JPSCBS is outlined in Algorithm 2. The root node is initialized by computing individual agent paths using MS-JPS, storing them in both the solution and the backup list (lines 1-2). The backup list stores all paths from MS-JPS, enabling retrieval of alternative paths if needed.

At each iteration, the CT node N with the lowest cost is selected from OPEN. Path validation is performed by iterating through time steps and checking for conflicts. If no conflicts are found, N is a goal node, and its solution is returned. Otherwise, a set of constraint info $Cs = (c, jp_1, jp_2)$ is identified, and the node is declared a non-goal (lines 5-8).

Algorithm 3: Update Path

Input: CT node N, agent set A, grid G

1 **for** *agent* $i \in A$ **do**
2    $Q_i \leftarrow N$.solution[i]
3    $B_i \leftarrow$ backups[i]
4    $cost_{current} \leftarrow$ CalcPathCost(Q_i.top())
5    $cost_{backup} \leftarrow$ CalcPathCost(B_i.back())
6   **while** $cost_{current} \geq cost_{backup}$ **do**
7      $p_{new} \leftarrow$ SearchByMSJPS(agent_states[i])
8   **if** $p_{new} \neq \emptyset$ **then**
9        $cost_{backup} \leftarrow$ CalculatePathCost(p_{new})
10   B_i.push_back(p_{new})
11   **else**
12   agent_states[i].clear()
13   break
14   **end**
15   **end**
16   start_idx $\leftarrow \min(|Q_i|, |B_i|)$
17   **for** $j \leftarrow$ *start_idx* **to** $|B_i| - 1$ **do**
18   **if** *CalcPathCost(*$B_i[j]$*)* $\leq cost_{current}$ **then**
19   Q_i.push($B_i[j]$)
20   **end**
21   **end**
22 **end**

At the high level of standard CBS, a conflict is identified here and the constraints are generated and imposed to agents later when solving conflicts. However, in JPSCBS, we directly generate the corresponding constraints for the affected agents upon detecting a conflict.

Definition 2 *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 define the segment to be replanned.*

Upon detecting conflicts, JPSCBS first attempts to find a bypass path as in ICBS (lines 9-10). If bypassing fails, the node is split into children as in CBS. However, instead of re-planning the full path, JPSCBS performs localized search between jp_1 and jp_2.

Path Update Strategy When conflicts are resolved, the affected agent's path costs increases. To maintain high-quality paths efficiently, we employ a dynamic path update mechanism. An further MS-JPS search is triggered when $cost(p_{current}) > cost(p_{worst})$, where $p_{current}$ is the best path in Q_i (priority queue of agent i in N.solution), and p_{worst} is the highest-cost path in B_i (back up paths searched by MS-JPS for agent i). It means MS-JPS further search might find a path better than current best path in Q_i. Once a new path is found, it is added into B_i. Regardless of whether or not a further MS-JPS search is needed, we will add all the paths in the backup list that were not added in Q_i. This strategy ensures incremental exploration of potentially better solutions.

Node Validation and Repair Ensuring solution feasibility after path updates is crucial. After each update, the agent's best path is examined to check for constraint violations. If any constraint is violated, we apply the same resolution strategy as in the high-level search: first seeking a bypass, and if none exists, handling the constraint locally.

Algorithm 4: Bypass

1 **foreach** *ConstraintInfo ci in* Cs **do**
2    $A \leftarrow N$;
3    $path\_segment \leftarrow$ FindPath(A, ci);
4   Replace $path\_segment$ at A;
5   **if** $A.cost = P.cost$ ***and*** $A.N_C < P.N_C$ **then**
6   N.solution $\leftarrow A$.solution;
7   Insert N into OPEN;
8   **return** *true*;

Tie-Breaking Rules for the Solution Priority Queue If multiple minimum-cost paths exist for an agent, ties are broken by selecting the path combination that results in the fewest conflicts. If conflicts remain, ties are resolved in FIFO order. For example, consider a k-agent scenario ($k > 2$) where two agents, a and b, each have two minimum-cost paths, while the other $k - 2$ agents have unique paths. This results in four possible solutions:

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

The algorithm evaluates the number of conflicts for each combination and selects the one minimizing conflicts. In practice, multiple minimum-cost paths per agent are rare, and simultaneous occurrences across multiple agents are even rarer. Thus, the overhead of conflict evaluation remains negligible compared to a simple FIFO tie-breaking strategy.

Finding Bypass We adopt Bypass 1, as detailed in (Boyarski et al. 2015a), within the JPSCBS framework (Algorithm 4). Unlike the original approach, which considers the entire path from the start to the goal, our method restricts the bypass search to the path segments between jump points as defined in the constraint information.

For each affected agent, we search for an alternative path segment between jp_1 and jp_2 that maintains cost-equivalence while satisfying constraint c. Upon finding a viable bypass, it replaces the original segment in the agent's path (line 4).

Resolving conflict If the attempt to find a bypass fails, we retain the path discovered during the bypass search and directly use the saved new path segment to replace the corresponding segment in the local conflict resolution process.

Low-Level

In JPSCBS's high-level search process, we maintain a global state repository for all agents. During CT node processing, when path updates are required, MS-JPS utilizes these stored states to resume the search.

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

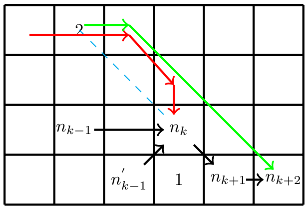


Figure 7: Case 1

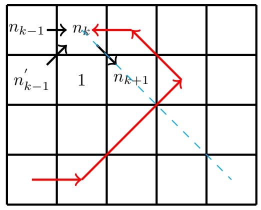


Figure 8: Case 2

paths. In our approach, we employ Space Time A*. Various other algorithms can also be used, such as Safe Interval Path Planning (SIPP). In principle, any low-level search algorithm compatible with CBS can be used by JPSCBS.

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 determine the jp_1 and jp_2 in the constraint info. In this paper, we select jp_1 and jp_2 as the nearest preceding and succeeding jump points to the conflicting vertex v, respectively, and replan the path segment between them. (Algorithm 5)

Jump Point Discarding Rule During conflict resolution between jump points, certain jump points must be strategically discarded to maintain solution quality. We identify two critical scenarios that necessitate jump point discarding.

Case 1: Consider a path from n_{k-1} approaching n_k from the north of obstacle 1 (Figure 7). Let r be the ray from n_k toward n_{k+1}. The jump point n_k should be discarded if any vertex in the alternative path segment, for example the red path in the figure, lie on the opposite side of r from n_{k-1}. There might be a path with less or equal cost, as demonstrated by the green path in the figure. Conversely, n_k remains necessary if all path vertices lie on the same side as n_{k-1}.

Case 2: When n_{k-1} approaches n_k from the south of obstacle 1 (Figure 8), n_k should also be discarded for the possible existence of path with less or equal cost.

When the newly searched path segment meets the discarding rule, we discard current jp_2 and extend the search to $next\_jp$ — the subsequent jump point in the jump point sequence returned by MS-JPS. This extension process continues recursively until either the discarding rule is not meet or the current segment's endpoint is the agent's goal vertex.

Although discarding jp_2 immediately when the discarding rule is satisfied may help identify a optimal solution, our observations suggest that delaying the discarding of jp_2 often has a negligible impact on the final solution quality. This finding opens up possibilities for exploring alternative, less aggressive discarding strategies that could enhance search efficiency. For example, one could discard jp_2 only when a certain number k of constraints are encountered within an l-distance neighborhood around jp_2, where both l and k can be determined empirically.

Algorithm 5: Find Path

$temp\_constraints \leftarrow N$.constraints + ci.constraint;
$path \leftarrow$ STA(start, goal, $temp\_constraints$, time);
if *HasBetterSolution($path$)* **then**
  $new\_ci \leftarrow$ ConstraintInfo($c, jp1, next\_jp$)
  return *FindPath(N, new_ci)*);

Experimental Results

We conducted extensive experiments across a wide range of environmental parameter settings and report representative results in this section. Our evaluation focuses on comparing JPSCBS with the standard CBS enhanced by the bypass optimization. We chose it as the baseline because JPSCBS incorporates optimization strategies similar to bypass. Also, other improvements and variants of CBS—such as MA-CBS, conflict prioritization, and CBSH—discussed in the background section, could also be adapted to further optimize JPSCBS with additional modifications. Moreover, as other several effective MAPF solvers have already been comprehensively compared with CBS variants in previous work (Boyarski et al. 2015b), we do not repeat those comparisons in this paper. All experiments were implemented in C++ and executed on a single thread of an AMD Ryzen 7 5800H processor (3.2 GHz) with 16 GB of RAM.

We use the map and scenario dataset in paper (Stern et al. 2019) for benchmark. Our setting considers a MAPF problem on an 8-neighbor grid with the following conditions:

1. 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 movement and waiting action is 1.
4. The objective function is the sum of costs.
5. The agent behavior at target is disappear at target.
6. Include all successive, identical diagonal-to-straight patterns in a single interval $(n_{k-1}, n_k, n_{k+1}, \ldots, n_{k+i})$ as possible interval.
7. The certain jump point is discarded immediately when Jump Point Discarding Rule is met.

We compare our method against the baseline from three perspectives: Success rate — the number of instances solved

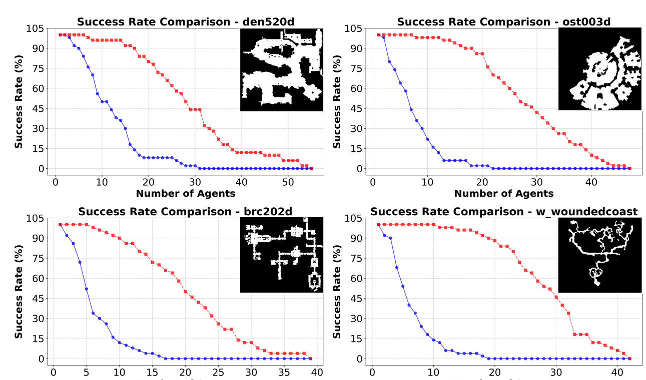


Figure 9: DAO map results (30-second time limit)

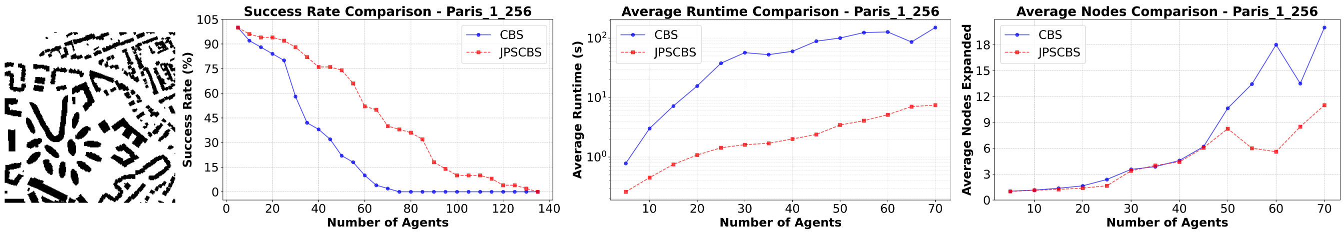


Figure 10: Performance comparison on the `Paris_1_256` map with a 5-minute time limit

by each algorithm within a given time limit, Runtime — the time taken to solve each instance successfully, and Expanded nodes — the number of nodes expanded during the successful solving of each instance.

In DAO and DAO2 maps tested with 30 seconds as time limit, JPSCBS demonstrates a significant performance advantage (Figure 9). This superiority can be attributed to the structured nature of these maps and the characteristic clustering of obstacles. On larger maps like `w_woundedcoast`, this advantage becomes even more pronounced. The substantial difference between the computational cost of resolving conflicts locally between jump points versus replanning entire paths explains JPSCBS's exceptional performance on these large-scale environments. In maze grids and city maps, there is also a considerable improvement in the success rate (Figure 10 and Figure 11).

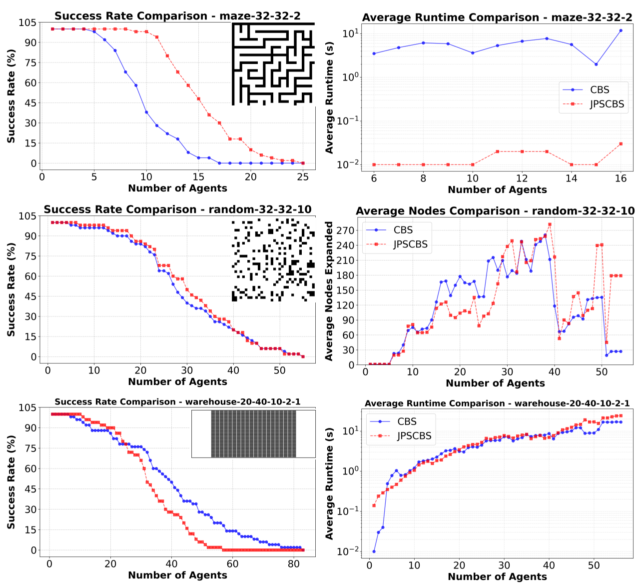


Figure 11: Benchmark results with a 30-second time limit of Maze-like grids, Open N x N Grids with Random Obstacles, Warehouse Grids

In terms of runtime, JPSCBS achieves an average speedup of approximately one order of magnitude on city maps against CBS. This performance advantage is even more pronounced on specific map types, such as maze-like grids, where JPSCBS can be up to two orders of magnitude faster.

Regarding the number of expanded nodes, both JPSCBS and CBS perform at a comparable level overall, with no significant differences observed (Figure 10 and Figure 11).

In warehouse-like maps with narrow corridors and in open grids with randomly distributed obstacles, the performance of JPSCBS is generally comparable considering success rate and run time to that of standard CBS(11). This is because in environments where obstacles are scattered but relatively close to each other, Jump Point Search (JPS) tends to generate a large number of jump points. As a result, MS-JPS may produce many paths with similar costs. JPSCBS must then maintain multiple paths, and after resolving conflicts, several alternative paths may compete as replacements. Each of these paths must be re-validated and checked against the existing constraints, leading to increased overhead.

While JPSCBS is not guaranteed to be optimal, its solution quality is empirically very close to optimal in many scenarios. For instance, in 50 test cases on the `Paris_1_256` map with a five-minute timeout, JPSCBS found the optimal solution in all 49 instances where both JPSCBS and CBS succeeded. Sub-optimal solutions were more frequently observed in warehouse-like maps and open grids with random obstacles with dense, scattered obstacles, where approximately half of the successful instances within a 30-second limit yielded sub-optimal solutions. However, across our comprehensive benchmark suite (32 maps, 50 scenarios each, 30-second limit), the cost difference between JPSCBS solutions and the optimal solutions never exceeded 0.5% in any test case.

Conclusion and Future Work

In this paper, we proposed a localized conflict resolution strategy for MAPF and applied it to the Conflict-Based Search (CBS) framework, resulting in a suboptimal variant JPSCBS. The proposed approach demonstrates strong performance, particularly on structured maps.

Several directions remain for future work: (1) Extending the proposed localized conflict resolution strategy to other MAPF solvers beyond CBS; (2) Investigating the impact of different possible interval selection strategies on overall performance; (3) Exploring more relaxed variants of the jump point discarding rule to further balance efficiency and solution quality; (4) Integrating other CBS improvements into the JPSCBS framework.

References

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. *Symposium on Combinatorial Search*, 961–962.

Boyarski, E.; Felner, A.; Sharon, G.; and Stern, R. 2015a. Don't Split, Try to Work It out: Bypassing Conflicts in Multi-Agent Pathfinding. *International Conference on Automated Planning and Scheduling*, 47–51.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, E. 2015b. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. *International Joint Conference on Artificial Intelligence*, 740–746.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. K. S.; and Koenig, S. 2018. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. *International Conference on Automated Planning and Scheduling*, 83–87.

Harabor, D.; and Grastien, A. 2011. Online Graph Pruning for Pathfinding on Grid Maps. *AAAI Conference on Artificial Intelligence*, 1114–1119.

Harabor, D.; and Grastien, A. 2014. Improving Jump Point Search. *International Conference on Automated Planning and Scheduling*, 128–135.

Jiaoyang Li; Zhe Chen; Daniel Harabor; Peter J. Stuckey; and Sven Koenig. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. *Proceedings of the ... AAAI Conference on Artificial Intelligence*, 36(9): 10256–10265.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with Consistent Prioritization for Multi-Agent Path Finding. *AAAI Conference on Artificial Intelligence*, 33(1): 7643–7650.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Conflict-Based Search for Optimal Multi-Agent Path Finding. *Artificial Intelligence*, 563–569.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Barták, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *Symposium on Combinatorial Search*, 151–159.