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

Anonymous submission

Abstract

Multi-Agent Pathfinding solvers, such as Conflict-Based Search, 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 —originally designed for single-agent pathfinding—that allows agents 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

The Multi-Agent Pathfinding (MAPF) problem (Stern et al. 2019) is defined on a graph $G = (V, E)$ and a set of k agents $\{a_1, \ldots, a_k\}$. Each agent a_i is assigned a start vertex $s_i \in V$ and a goal vertex $g_i \in V$. Time is discretized, and at each time step, an agent can either move to an adjacent vertex or wait in place. A path p is a sequence of adjacent vertices $\langle v_0, v_1, \ldots, v_k\rangle$ representing the trajectory of an agent from its start to its goal. A conflict occurs when the planned paths of agents intersect in space and time, such as occupying the same vertex or traversing the same edge simultaneously. The objective is to find a set of conflict-free paths for all agents that minimize a global cost function.

Several MAPF solvers, including Conflict-Based Search (CBS) (Sharon et al. 2015) and Priority-Based Search (PBS) (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 full path recalculations grows exponentially. This global replanning strategy, although 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 Multi-Solution Jump Point Search (MS-JPS), a variant of Jump Point Search (Harabor and Grastien 2011) that generates multiple path candidates that can serve as alternatives when primary path become more costly due to conflicts. Second, we develop a localized conflict resolution strategy that handles conflicts within specific ranges between jump points, avoiding the need for complete path replanning.

While these improvements can be applied to a range of MAPF algorithms that require full path recalculation or seek efficient local conflict resolution mechanisms, we demonstrate their effectiveness by integrating them into the CBS framework. The resulting variant, JPSCBS, significantly reduces the computational overhead of conflict resolution with only a minor compromise in 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 (JPS)

Jump Point Search (JPS) is an optimal pathfinding algorithm based on A* . JPS searches undirected, 8-connected uniform-cost grid maps where straight move costs 1 and diagonal move costs $\sqrt{2}$ by employing pruning and jumping rules (Harabor and Grastien 2011).

Throughout the remainder of the paper, let $p(x)$ denote the parent of vertex x in the search tree; we define $p(x) = \emptyset$ if x is the start node. Let $\langle n_1, \ldots, n_k\rangle$ denote any valid path from n_1 to n_k in the graph. Given a vertex x and a path p, we write $p \setminus x$ to indicate that $x \notin p$, i.e., x does not appear on path p. We use $\text{len}(p)$ to represent the cost of path p.

Pruning Rules: Given a node x, and assuming no obstacles, JPS prunes a neighbor n of x if n satisfies one of the following two pruning rules.

Straight Moves If x is reached from $p(x)$ via a straight move, a neighbor n is pruned if there exists a path $\langle p(x), \ldots, n\rangle \setminus x$ such that:

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

Figure 1 (a) illustrates this case, where $p(x) = 4$ and all neighbours of x are pruned except for $n = 5$.

Diagonal Moves If x is reached via a diagonal move, n is pruned if there exists a path $\langle p(x), \ldots, n\rangle \setminus x$ such that:

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

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

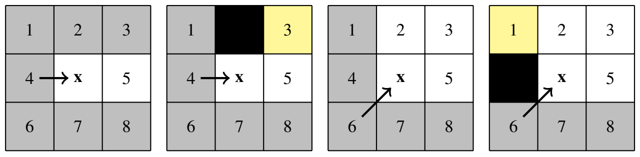


Figure 1: Illustration of pruning rules. Gray vertices: pruned neighbors; White vertices (excluding x): natural neighbors; Yellow Vertices: forced neighbors; Black vertices: obstacles

The remaining neighbors, not pruned, are referred to as the *natural neighbors* of x. In the presence of obstacles, if n is not a natural neighbor of x and $\text{len}(\langle p(x), x, n\rangle) < \text{len}(\langle p(x), \ldots, n\rangle \setminus x)$, n is defined as a *forced neighbor* of x and thus is not pruned (see Figures 1(b) and 1(d)).

Jumping Rules: JPS applies a simple recursive *jumping* procedure to each natural and forced neighbor of the current node x, aiming to find successors located further along the same direction during node expansion. Precise details of this procedure are provided in (Harabor and Grastien 2011).

Conflict-Based Search (CBS)

Conflict-Based Search (CBS) is a two-level optimal algorithm for solving MAPF problem (Sharon et al. 2015). It decomposes the problem into 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: The high-level search operates on a *Constraint Tree* (CT) (Sharon et al. 2013), which is explored in a best-first manner to resolve conflicts, prioritizing nodes by their cost. In case of ties, preference is given to CT nodes with fewer conflicts, further breaking ties in a First-In-First-Out (FIFO) manner. A node N in the CT is considered a goal node when the set of paths in N.solution are conflict-free. Each CT node N 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 time steps sequentially from t = 0. Upon detecting the first conflict, CBS performs a *split* operation, generating two child nodes with additional constraints. For example, 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.

CBS Variants and Other MAPF Solvers While CBS guarantees optimality, in worst-case scenarios, the number of CT nodes can grow exponentially (Sharon et al. 2015). This limitation becomes particularly pronounced in dense environments or when dealing with large numbers of agents. 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. MAPF-LNS2 (Li et al. 2022) employs Large Neighborhood Search to iteratively repair conflicts by replanning paths for dynamically selected agent subsets, achieving high scalability through its efficient integration of prioritized planning and adaptive neighborhood selection. Several approaches, including Rolling-Horizon Collision Resolution (RHCR) (Li et al. 2020) and X* (Vedder and Biswas 2021), utilize time windows to handle conflicts locally, demonstrating the effectiveness of local repair strategies in MAPF.

While CBS variants reduce CT node expansions through various strategies and PBS effectively reduces the search space, they all require complete path replanning for the affected agents. RHCR and X* primarily focus on temporal aspects without explicitly considering the spatial characteristics of the environment. In this paper, we propose an approach which leverages the geometric properties of grid maps with obstacles to define meaningful local ranges using jump points. By focusing on these local ranges, our approach enables more targeted and efficient conflict resolution.

Multi-Solution Jump Point Search

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

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 complete algorithm is presented in Algorithm 1. The overall search process retains the structure of Jump Point Search (JPS), with key modifications to the management of the open and closed lists. Specifically, if a node does not exist in the current search's $state.open$ or closed list, it is added to $state.open$ (line 19). If a node has already been expanded during the current search, it is skipped for this round and instead stored in a temporary set $temp_nodes$for

Algorithm 1: MS-JPS

```
Input: $state$
1  $closed \leftarrow \emptyset$ ;
2  $temp\_nodes \leftarrow \emptyset$ ;
3  while $state.open \neq \emptyset$ do
4      $current \leftarrow$ best node in $state.open$;
5      $closed \leftarrow closed \cup \{current\}$ ;
6      $state.closed \leftarrow state.closed \cup \{current\}$ ;
7      if $current.pos = goal$ then
8          $state.open \leftarrow state.open \cup temp\_nodes$ ;
9          return ReconstructPath($current$) ;
10     $successors \leftarrow$ IdentifySuccessors($current$) ;
11     foreach $succ \in successors$ do
12         $cost \leftarrow$ GetMoveCost($current.pos, succ$) ;
13         $g \leftarrow current.g + cost$ ;
14         $h \leftarrow$ Heuristic($succ, goal$) ;
15         $node \leftarrow$ CreateNode($succ, g, h, current$) ;
16         if $succ \in closed \vee succ \in state.open$ then
17             $temp\_nodes \leftarrow temp\_nodes \cup \{node\}$ ;
18         else
19             $state.open \leftarrow state.open \cup \{node\}$ ;
20 return $\emptyset$ ;
```

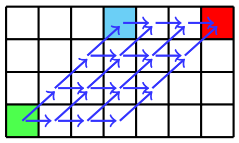


Figure 2: Symmetric Paths

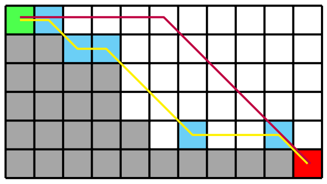


Figure 3: Interval Extension

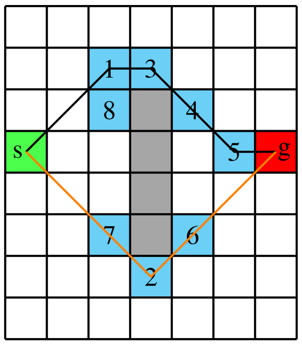
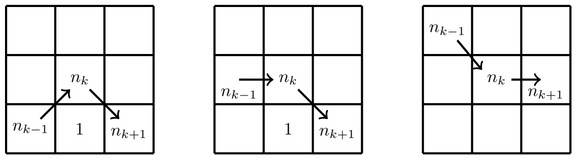


Figure 4: MS-JPS example. Green vertex(s): start vertex; Red vertex(g): goal vertex.



Type (a) Type (b) Type (c)

Figure 5: Turning Point

future searches (lines 16–17). This is because each individual search aims only to identify the best path under the current input state. Once a search round concludes, all nodes in the temporary set are reinserted into $state.open$ for subsequent exploration (line 8). This mechanism enables MS-JPS to incrementally explore alternative paths without restarting the search from scratch. Additionally, in the *jump* procedure, we ensure that any successor path generated from unpruned neighbors does not form a cycle with the path from the start node to the current node (line 10).

In the first search, the agent's search state is initialized with $state.open$ containing the start node and an empty $state.closed$, which is then provided to MS-JPS. In subsequent searches, the search state from the previous search is reused to continue pathfinding.

Figure 4 illustrates an example of the MS-JPS search process, where the octile distance is employed as the heuristic function. The search begins at the start vertex. Upon expanding this vertex, successors 1 and 2 are identified and inserted into $state.open$. The node with the smaller f-value, node 1, is expanded next, resulting in the discovery of successor 3. Expanding node 3 leads to successors 4 and 5. When node 5 is expanded, the goal vertex is reached, triggering path reconstruction. Nodes 2 and 4 remain in $state.open$ and are retained for subsequent searches.

In the second round of the search, the algorithm expands node 4 and discovers successor 6. Next, node 2 is expanded, revealing successors 6 and g. Since node 6 is already present in $state.open$, it is temporarily stored in $temp\_node$. Then g is expanded, triggering path reconstruction. The nodes stored in $temp\_node$ are inserted into $state.open$ for further exploration.

During the third round, the algorithm expands node 6 (with parent node 2), and the search proceeds accordingly. Throughout this process, two cycles $\{s, 1, 3, 4, 6, 2, 7, 8, 3\}$ and $\{s, 2, 6, 4, 3, 8, 7, 2\}$ are encountered, causing the jumping procedure of the corresponding direction to terminate. Ultimately, $state.open$ is exhausted, and MS-JPS terminates without finding additional paths, having already identified two distinct solutions.

Possible Interval

Once MS-JPS finds a path, we need to figure out which jump points need to be kept to handle conflicts. 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 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.

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

1. Diagonal-to-Diagonal (Figure 5 Type (a)).
2. Straight-to-Diagonal (Figure 5 Type (b)).
3. Diagonal-to-Straight (Figure 5 Type (c)).

Other turning points, such as Straight-to-Straight, are trivially suboptimal and are thus not considered, in accordance with the pruning rules (Harabor and Grastien 2011). Let $p = \langle n_{k-1}, n_k, n_{k+1} \rangle$ denote the path segment, where n_k is the turning point. For Types (a) and (b), n_{k+1} is not pruned because it is considered a forced neighbor (Figures 1 (b) and (d)). This implies that $\text{len}(p) < \text{len}(\langle n_{k-1}, \dots, n_{k+1} \rangle \setminus n_k)$.

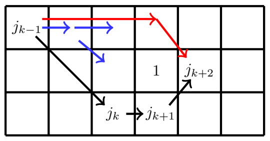


Figure 6: Case 1

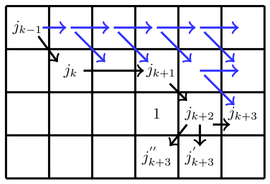


Figure 7: Case 2

Consequently, n_k must remain an essential turning point to preserve the path cost. For Type (c), n_{k+1} is not pruned due to the pruning rule for diagonal moves (Figure 1 (c)), meaning that a path $\langle n_{k-1}, \ldots, n_{k+1} \rangle$ may exist with the same cost as p. Therefore, for the path segment between the jump points preceding and following n_k, it forms a possible interval for finding an alternative path with the same cost.

While we have identified a possible interval, the actual scope for cost-equivalent paths may extend beyond these boundaries (Figure 3). This extension occurs because the removal of n_k invalidates the pruning rules for subsequent jump points. In the case of straight moves, neighbors with equal cost are pruned to prevent redundant expansions along directions already covered by diagonal moves (see Figure 1(a), where neighbors 3 and 8 are pruned). If these directions were not pruned, it would generate all cost-equivalent paths. Some of these cost-equivalent paths may lead to subsequent nodes beyond the initially defined possible interval.

To account for the potential extension of a possible interval, we examine the next turning point that follows the last turning point within the interval. There are two cases concerning the next turning point after a turning point of type (c). Let j_i denote the i^{th} jump point along the path.

In the first case, the direction from j_k to j_{k+1} differs from that of j_{k-1} to j_k (Figure 6). According to the pruning rules, vertex 1 must be an obstacle. Among the directions pruned for straight moves from j_{k-1} that could generate cost-equivalent paths, we examine these directions and confirm that it is impossible to reach j_{k+2} or any subsequent path vertices via cost-equivalent alternative paths. In the second case, the direction from j_k to j_{k+1} is the same as that from j_{k-1} to j_k (Figure 7). In this scenario, a cost-equivalent alternative path may exist only if the direction from j_{k+2} to j_{k+3} is consistent with that from j_k to j_{k+1}.

Note that in both cases, it is theoretically possible to reach later vertices from j_{k-1} with lower or equal cost, as shown by the red path in Figure 6. However, such paths would have already been explored in earlier MS-JPS searches. Therefore, our analysis focuses exclusively on cost-equivalent alternative paths that are pruned and thus omitted from the MS-JPS search results.

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 $(n_{k-1}, n_k, n_{k+1}, n_{k+2}, n_{k+3})$.

In this paper, we adopt the second strategy.

Reconstruct Path

When MS-JPS identifies a path, it returns not only the path itself but also the corresponding jump points and possible intervals along the path. Jump points here are defined as turning points expanded by MS-JPS that appear in the final path and stored in the order they are expanded during the search. The start vertex is always included as the first jump point. For each possible interval, only its start and end vertices are retained, with intermediate turning points removed from jump points.

Back to the example in Figure 4, the first reconstructed path, depicted by the black line, consists of the jump points $\{s, 3, g\}$ and includes two possible intervals: $(s, 1, 3)$ and $(3, 5, g)$. The second path found, depicted as an orange line in the figure, which consists of the jump points $\{s, 2, g\}$ but does not contain any possible interval.

JPSCBS

To validate the feasibility of our improvement, we integrate it with CBS and propose a new variant: CBS improved by JPS (JPSCBS).

High-Level

Here, 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, the solution of a CT is changed from a set of paths into a set of priority queues, where each queue stores paths originally generated by MS-JPS. The path with the lowest cost is selected as the agent's plan, while other paths serve as candidates.

Tie-Breaking Rule for Solution Priority Queues Given a set of agents $\mathcal{A} = \{a_1, \ldots, a_k\}$, let $\mathcal{P}_i$ denote the set of lowest-cost paths for agent a_i. The set of candidate solutions is the Cartesian product $\mathcal{P}_1 \times \cdots \times \mathcal{P}_k$. For each solution $\mathbf{p} = (p_1, \ldots, p_k)$, let $Conflicts(\mathbf{p})$ denote the total number of conflicts. The tie-breaking rule selects

$$\mathbf{p}^* = \arg \min_{\mathbf{p} \in \mathcal{P}_1 \times \cdots \times \mathcal{P}_k} Conflicts(\mathbf{p}),$$

breaking further ties in FIFO order. In practice, $|\mathcal{P}_i| > 1$ is rare, and simultaneous multiple choices across agents are even rarer, making the computational overhead negligible.

Processing a Node in the CT In JPSCBS, we directly generate the corresponding constraints for the agents involved in conflicts and specify the two jump points that define the path segment to be re-planned. This approach differs from CBS, where conflicts are first detected and then constraints are generated for the corresponding agents. The information generated for each agent is defined as follows:

Definition 1 *A constraint info is a tuple (c, jp_1, jp_2), where c represents a constraint that occurs on the path segment between jp_1 and jp_2, and jp_1 and jp_2 are the two jump points that delimit this segment to be re-planned.*

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      $N \leftarrow$ best node from OPEN;
5      $Cs \leftarrow$ GenerateConstraintInfos($N$);
6      if $Cs = \emptyset$ then
7          return $N$.solution;
8      if FindBypass($N$, $Cs$) then
9          continue;
10     foreach ConstraintInfo $c_i$ in $Cs$ do
11         $A$.constraints $\leftarrow N$.constraints
             $+c_i$.constraint;
12         $A$.solution $\leftarrow$ ResolveConflictLocally($N$, $c_i$);
13         $A$.cost $\leftarrow$ SIC($A$.solution);
14         UpdateSolutions($A$);
15         ValidateAndRepairNode($A$);
16         Insert $A$ into OPEN;
```

The high level process of JPSCBS is outlined in Algorithm 2. The root node is initialized by performing an initial search for the individual agent paths using MS-JPS, with the paths being stored in both the solution and backup lists (lines 1-2). The backup list retains all paths generated by MS-JPS, allowing for retrieval of alternative paths when necessary. Upon the generation of constraint information, JPSCBS splits the node into children. However, instead of replanning the entire path, JPSCBS performs a localized search between jp_1 and jp_2 to resolve the conflict.

Path Update Strategy When conflicts are resolved, the affected agent's path costs might increase. To maintain high-quality paths efficiently, we employ a path update strategy after resolving conflict (Algorithm 3). 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) (line 2-6). 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. If no path is found, it means there are no more possible paths and no need for further MS-JPS searches (line 7-14). 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 (line 16-21).

Node Validation and Repair Ensuring solution validity after path updates is essential. After each update, the best path of each agent is checked for constraint violations. If a violation is detected, the corresponding constraint info is generated and the path segment between $jp1$ and $jp2$ is replanned subject to the violated constraint. This process is repeated until all agents' best paths are consistent with the current set of constraints (Algorithm 2, line 15).

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          $p_{new} \leftarrow$ MS-JPS(agent_states[i])
8          if $p_{new} \neq \emptyset$ then
9              $cost_{backup} \leftarrow$ CalcPathCost($p_{new}$)
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
```

Finding Bypass JPSCBS can be integrated with existing CBS enhancements, such as Bypass (Boyarski et al. 2015a). In this work, we incorporate Bypass 1 into the JPSCBS framework (Algorithm 2, lines 8–9). Unlike the original Bypass approach, which attempts to find a bypass to conflict for the entire path, our method restricts the bypass operation to the path segment bounded by jump points, as specified in the constraint information. For each affected agent, we search for a cost-equivalent alternative path segment between jp_1 and jp_2 that maintains cost-equivalence that satisfies the given constraint c. If such a bypass is found, it replaces the original segment in the agent's path directly, thereby avoiding the generation of two child nodes.

Low-Level

To resolve conflicts between jump points, an additional low-level search algorithm other than MS-JPS is needed to find valid paths. In this paper, we employ Space-Time A*. Other algorithms, such as Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011), can also be employed. In principle, any low-level search algorithm that is compatible with CBS can be integrated into JPSCBS.

The final step is to identify jp_1 and jp_2 that delimit the segment to be replanned. In this paper, we select the nearest preceding and succeeding jump points to the conflicting vertex v along the path as jp_1 and jp_2, respectively, and replan the path segment between them. Other options, although potentially valuable, are not discussed here.

Jump Point Discarding Strategy During conflict resolution between jump points, discarding certain jump points can help preserve solution quality. In paths found by MS-JPS,

Algorithm 4: ResolveConflictLocally

```
Input: CT node $N$, ConstraintInfo $c_i$
1 $constraints \leftarrow N$.constraints + $c_i$.constraint;
2 $path \leftarrow$ STA*($c_i.jp_1$, $c_i.jp_2$, $constraints$, $c_i.c.t$);
3 if DiscardingNeeded(path) then
4     $new\_c_i \leftarrow$ ConstraintInfo($c, jp_1, next\_jp$)
      return FindPath($N$, $new\_c_i$);
5 return $path$;
```

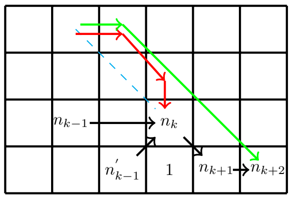


Figure 8: Case 1

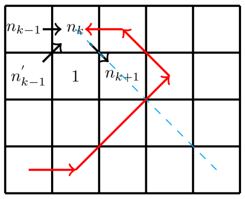


Figure 9: Case 2

only Type A and Type B jump points remain (as illustrated by the paths from n_{k-1} and n'_{k-1} in Figures 8 and 9). Accordingly, we restrict our analysis to these two types of turning points and provide the following discarding strategy.

Consider a path from n_{k-1} approaching n_k from the north of obstacle 1 (Figure 8). 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 newly found path segment, for example the red path segment 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 is not discarded if all path vertices lie on the same side as n_{k-1}. When n_{k-1} approaches n_k from the south of obstacle 1 (Figure 9), 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 of jp_2 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 (see Algorithm 4).

However, such an aggressive strategy may cause the low-level search for certain agents to quickly degenerate to that of standard CBS—namely, repeatedly replanning the entire path—particularly in scenarios where conflicts frequently arise near jump points. Developing less aggressive discarding strategies that could further enhance search efficiency. For example, one may opt to discard jp_2 only when a specified number k of constraints are encountered within an l-distance neighborhood of jp_2, where both k and l are empirically defined parameters.

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

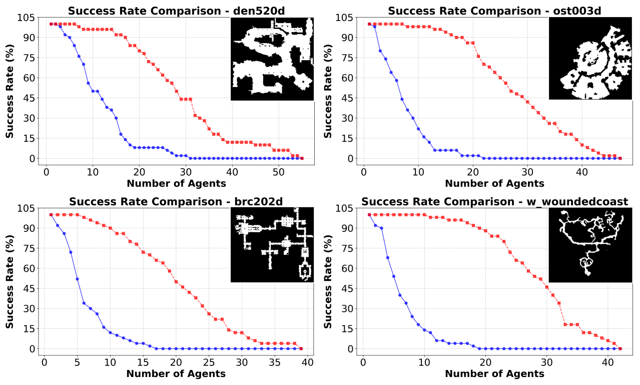


Figure 10: DAO map results (30-second time limit) where blue line represents CBS and red line represents JPSCBS

optimization. We chose it as the baseline because JPSCBS incorporates optimization strategies similar to bypass. Also, other improvements and variants of CBS—such as 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) (Li et al. 2022), 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 datasets provided in (Stern et al. 2019) for benchmarking. The evaluation methodology and related definitions such as types of conflicts, also follows the specifications outlined in that paper. Our experimental setting considers a MAPF problem on an 8-connected 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. 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 by each algorithm within a given time limit, Runtime — the average time taken to solve an instance successfully, and Expanded nodes — the number of nodes expanded during the successful solving of each instance.

On DAO and DAO2 maps with a 30-second time limit, JPSCBS demonstrates a significant performance advantage (Figure 10). This improvement is largely due to the structured layout of these maps and the clustered distribution of obstacles. On larger maps such as w_woundedcoast, the advantage becomes even more pronounced. The key factor behind this superior performance is the reduced computational

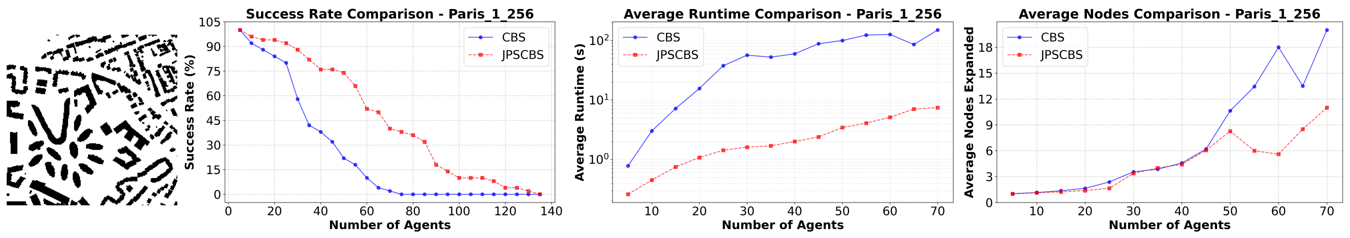


Figure 11: Performance comparison on the `Paris_1_256` map with a 5-minute time limit

overhead of resolving conflicts locally between jump points, as opposed to replanning complete paths. In maze-like grids and city maps, JPSCBS also shows notable improvements in success rate (Figure 11 and Figure 12).

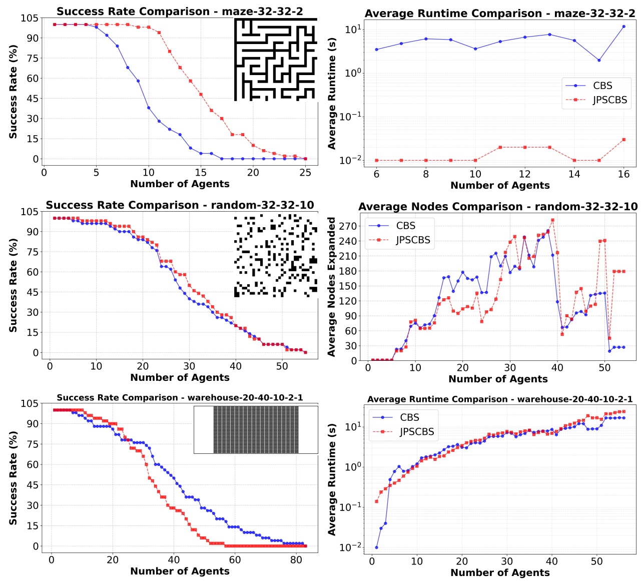


Figure 12: Benchmark results with a 30-second time limit of Maze-like grids, Open N x N Grids with Random Obstacles, Warehouse Grids

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 11 and Figure 12).

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(Figure 12). This is because in environments where obstacles are scattered but relatively close to each other, MS-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 (28 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 large 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; (5) Exploring alternative selections of jp_1 and jp_2. For example, selecting the start vertex and the nearest succeeding jump points to the conflicting vertex v as jp_1 and jp_2 might result in an optimal solution for JPSCBS; (6) Evaluating the completeness of JPSCBS.

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.

Li, J.; Chen, Z.; Harabor, D.; J. Stuckey, P.; and Koenig, S. 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.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. K. S.; and Koenig, S. 2020. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. *Adaptive Agents and Multi-Agent Systems*, 1898–1900.

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.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe Interval Path Planning for Dynamic Environments. *IEEE International Conference on Robotics and Automation*, 5628–5635.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219: 40–66.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195: 470–495.

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.

Vedder, K.; and Biswas, J. 2021. X*: Anytime Multi-Agent Path Finding for Sparse Domains using Window-Based Iterative Repairs. *Artificial Intelligence*, 291: 103417.