2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

P4: Pruning and Prediction-based Priority Planning

Rui Yang and Rajiv Gupta

***Abstract*— In recent years, multi-agent path finding (MAPF) has attracted widespread attention in the fields of artificial intelligence and robotics. Its main goal is to find paths for multiple agents, each having specified start and end locations on a grid, without collisions, while minimizing the total travel time. In this work, we present a new algorithm called P4 (Pruning and Prediction-based Priority Planning), designed to accommodate large numbers of agents with enhanced scalability. The P4 method combines three key components: Point-to-Point (PnP) algorithm, dynamic window approach, and path direction prediction. In this way we reduce the search space and increase the speed of the computation. Our experiments show that P4 consistently achieves shorter execution times and produces solutions that are close to optimal. For example, for 200 agents and real map `orz900d`, P4 is 4× faster than optimal algorithm CBSH while the sum of delays is within 15% of optimal. The P4 method outperforms other existing suboptimal methods in both performance and solution quality. We also show that our approach exceeds existing methods in success rate under time constraints. As time limit is increased from 0.1 to 100 seconds, success rate of P4 increases from 50% to 100%. On the other hand, the success rate for alternative sub-optimal methods is less than that of P4.**

I. INTRODUCTION

Multi-Agent Path Finding (MAPF) [1], [2] is a critical problem in the field of artificial intelligence and robotics, which involves the coordination of multiple agents in a shared environment. The environment is typically represented as a grid map, where each agent is assigned a unique start and target location. The primary objective of the MAPF problem is to determine collision-free paths for all agents such that the cumulative travel time (Sum Of Delays) is minimized.

In the MAPF problem, the agent can have two types of actions: *move* or *wait*. Both actions consume one timestep and incur a unit cost. The move action means that the agent moves to an adjacent cell in the grid map, and the wait action means that the agent waits in its current cell for a timestep.

The MAPF problem has many applications in various domains, including autonomous systems, logistics, and robotics [3]. It can be used to plan the paths of autonomous vehicles to avoid collisions or coordinate the movements of multiple robots in a shared environment to perform tasks efficiently. Although widely applicable, the MAPF problem remains computationally challenging due to the explosion of possible paths for each agent and further increases exponentially with numbers of agents and grid map size. Therefore, developing efficient and scalable solutions for the MAPF problem represents an significant research challenge.

The authors are with CSE Department, University of California, Riverside, CA 92521, USA. Email: {ryang088,rajivg}@ucr.edu

Definition of the MAPF Problem: Formally, the Multi-Agent Path Finding (MAPF) problem [1][2] can be defined as follows.

Given a strongly connected, simple graph $G = (V, E)$, we consider a set of m agents $\mathcal{A} = \{A_1, A_2, \ldots, A_m\}$ with $|\mathcal{A}| \leq |V|$, and each agent having its own initial vertex $S_i \in V$ and target vertex $G_i \in V$.

At each discrete timestep t, agents have two choices for their actions: move to an adjacent vertex or remain where they are. Their path P_i is comprised of consecutive vertices between S_i and G_i, and can either consist of adjacent or identical vertices to indicate a move action or wait action; we assume agents remain or disappear at their goal vertices indefinitely once reached.

The agents must avoid two types of conflicts: *vertex conflicts*, where two agents occupy the same vertex at the same timestep, and *swap conflicts*, where two agents exchange their positions.

A solution to the MAPF problem is a set of collision-free paths $P = \{P_1, P_2, ..., P_m\}$, one for each agent, that ensures all agents reach their respective target vertices. The objective is to find such a solution that minimizes the sum of paths $\sum_{A_i \in \mathcal{A}} l(P_i)$, where $l(P_i)$ is the length of path P_i, thus effectively minimizing the Sum of Delays(SoD = $\frac{\sum_{A_i \in \mathcal{A}} (l(P_i))}{\sum_{A_i \in \mathcal{A}} \text{dist}(S_i, G_i)}$) for agents. SoD is the normalized way to evaluate the quality of solutions. **In this work the offline version of problem is considered where all paths are preplanned on a server without involvement of agents. Then the paths and actions are communicated to agents that then execute the plan.**

A. Prior Work

The MAPF problem is a topic of significant interest. Various methods have been devised to address it, including CBSH [4], Explicit Estimation Conflict-Based Search (EECBS) [5], Priority Planning [6], Priority-Based Search [7], LaCAM [8]. *CBSH* is an advanced algorithm that builds upon traditional Conflict Based Search (CBS) [9], [10]. It employs a two-level framework to resolve conflicts between agents. At the low level, individual paths are planned for each agent. At the high level, a conflict tree is constructed and explored to resolve collisions. CBSH improves upon CBS by introducing Pairwise symmetry reasoning and rectangle and corridor reasoning techniques [4]. *EECBS* [5]uses Explicit Estimation Search for node selection on the high level and an informed heuristic for the high level that makes EECBS-Bounded to solve MAPF Problem suboptimal. *Priority Planning* [6] is a method that adopts a sequential approach to solve the MAPF problem. Each

agent is assigned a priority, and paths are found in the order of these priorities. While simple and easy to implement, this method can be inefficient in scenarios where high-priority agents block the paths for lower-priority agents. Some priority-based methods have show their potential advantages in MAPF problem [8]. Its main advantage lies in its straightforward nature, but it lacks in optimality and scalability [6]. However, scalability remains a persistent challenge [11], especially as the number of agents grows. The MAPF problem remains a challenging topic, especially in the context of scalability. While above methods offer unique approaches to solve the issue, each has its own set of advantages and limitations. Further research [12]is needed to develop methods that are both scalable and optimal. Finally, *LaCAM* [8] shows how to find sub-optimal solution quickly.

The Conflict-Based Search (CBS) algorithm, in particular, has seen extensive exploration [11], [13]. Boyarski et al. [14] further refined this approach, suggesting methods to bypass conflicts without resorting to path splitting. Moreover, Li et al. [15] introduced improved heuristics for MAPF using CBS, showcasing the algorithm's adaptability. Also, there are other improvements for CBS such as those by Li et al. [16]. *PBS* [7] combines priority based search with CBS.

Besides traditional pathfinding techniques, other paradigms have emerged. Biswas [12] proposed the X* algorithm, an anytime approach that employs window-based iterative repairs, that is especially suitable for sparse domains. While in [12], the window-based iterative repairs expand to cover the global path and find the optimal solution. In this paper, inspired by the above general idea, we develop a Window-Based Pruning algorithm that uses path directions to predict and dynamically adjust the window size to find sub-optimal solution in the local window that eliminates a collision.

Recent advancements have also seen the fusion of neural networks with MAPF [17], [18]. An approach proposed by Onken [17] uses neural networks to linearly scale with the control problem's dimension, addressing the curse of dimensionality. Similarly, Chen [18] combines deep reinforcement learning with a supervised contrastive loss, incorporating a self-attention mechanism in the policy network.

B. Motivation

Despite the significant advancements in the field of Multi-Agent Path Finding (MAPF), existing methods often struggle with scalability issues when managing a large number of agents as shown in Fig. 1. The computational complexity of these methods tends to increase exponentially with the number of agents, making them inefficient for large-scale applications. Also, most of the existing methods explore the entire search space, which can be very large and filled with a large number of invalid paths. This exhaustive search strategy increases the computational burden and makes the path finding process inefficient. There is a clear need for a more targeted search strategy that focuses on the most promising paths and prunes the search space to improve

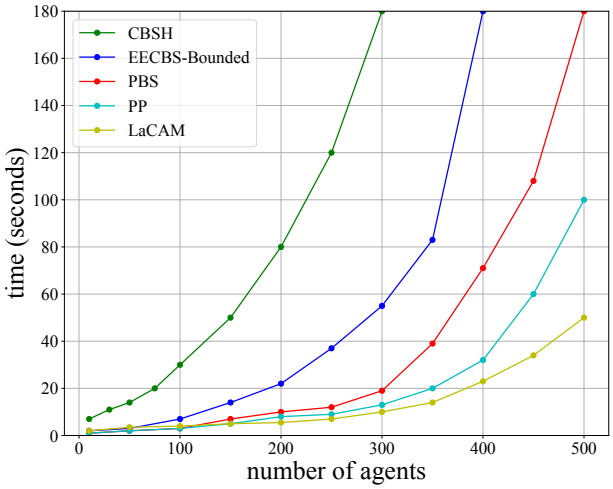


Fig. 1: Scalability of MAPF Algorithms.

efficiency. In the paper we propose the P4 algorithm to achieve this goal.

II. Our Approach: The P4 Method

In this paper we introduce the *P4* method, a novel approach to the MAPF problem that is specifically designed to address scalability issues and improve computational efficiency. Within the architecture of *P4*, three strategies are combined. First, we use the Point-to-Point (PnP) algorithm [19], which is a useful way to find the best paths for individual agents. Next, we apply a dynamic window method that resolves conflicts that might occur among multiple agents. Finally, we use a path direction prediction technique that predicts where agents will go in the future by looking at the paths they have taken in the past. By combining these three strategies, the *P4* method not only offers an efficient and scalable solution to the MAPF problem, it also achieves high solution qualities.

A. The P4 algorithm

The proposed P4 algorithm, detailed in Algorithm 1, presents a novel solution to the Multi-Agent Path Finding (MAPF) problem. It achieves scalability and efficiency by integrating three main strategies: initial path planning using the Point-to-Point (PnP) algorithm, dynamic window management for conflict resolution, and priority-based path adjustment.

a) ***Overall Workflow for P4****:* The algorithm starts by computing the initial paths for each agent using the PnP algorithm [19] and assigns a unique priority to each agent based on path length. It then iteratively monitors the agents' progress, dynamically handles conflicts through window adjustments, and updates priorities and paths accordingly until all agents reach their destinations.

b) **Initial Path Planning and Priority Assignment:**

- **Lines 1-2**: For each agent A_i, compute the initial shortest path P_i from the start node S_i to the goal node G_i using the PnP algorithm.
- **Lines 3-4**: Assign a unique priority based on the shortest path, sorting them in descending order, and introduce a random uniqueness parameter for tie-breaking.

Algorithm 1 P4: Pruning and Prediction-based Priority Planning.

Require: Graph $G = (V, E)$, Agents $\mathcal{A} = \{A_1, A_2, ..., A_n\}$, Start and Goal nodes S_i, G_i for A_i

1: **for** each A_i in $\mathcal{A}$ **do**
2: Initialize paths P_i from S_i to G_i using PnP algorithm
3: Assign a unique priority based on PnP shortest path descending sorted π_i with random uniqueness parameter
4: Sort P_i according to π_i
5: **end for**
6: **while** not all agents have reached their destinations **do**
7: **for** each node v in V **do**
8: Calculate the safe interval I_v
9: **end for**
10: **for** each agent A_i **do**
11: **if** Conflict exists on P_i **then**
12: Let C be the set of conflict points on P_i
13: Determine the dynamic window diameter D_w as the distance to the midpoint of the nearest conflict in C
14: Split the conflicting part of P_i into two paths P_{i1} and P_{i2} within the range defined by D_w, leaving the non-conflicting parts unchanged. Inherit the priority of P_i for both P_{i1} and P_{i2}
15: **for** each path P_{ij} in $\{P_{i1}, P_{i2}\}$ **do**
16: Let P_{ij} be the current path
17: If $P_{ij} = P_{i1}$, then wait at the current conflict point
18: If $P_{ij} = P_{i2}$, then use the PnP algorithm to find a new path using the remaining safe nodes within the dynamic window
19: If the above steps lead to a stuck agent, perform backtracking on P_{ij}
20: **end for**
21: Choose the path, either P_{i1} or P_{i2}, that has the lowest cost, if backtracking was necessary
22: **end if**
23: **end for**
24: **for** each agent A_i **do**
25: Update the priority and safe intervals for all nodes on P_i
26: Adjust P_i according to new priorities
27: **end for**
28: **end while**

c) **Safe Interval Calculation:**

- **Lines 7-9**: For each node v in the graph G, calculate the safe interval I_v during which node is unoccupied.

d) **Conflict Resolution via Dynamic Window Management:**

- **Lines 11-13**: For each agent A_i, if a conflict exists on path P_i, determine the set of conflict points and the dynamic window diameter D_w.
- **Line 14**: Split the conflicting path into two paths within the range defined by D_w, leaving non-conflicting parts unchanged.
- **Lines 15-20**: For each of the two paths, handle conflicts by either waiting or by finding a new path within the dynamic window.
- **Line 21**: Perform backtracking if necessary and choose the path with the lowest cost.

e) **Priority Update and Path Adjustment:**

- **Lines 25-26**: For each agent A_i, update the priority and safe intervals for all nodes on P_i.
- **Line 27**: Adjust the paths according to the new priorities to ensure collision-free navigation.

By combining these strategies, the P4 algorithm offers an effective and flexible solution to the MAPF problem. It ensures that all agents reach their goal without collisions, while avoiding conflicts and reducing the search space. This method is very effective for scenarios with a large number of agents. In Figure 2 we illustrate the different steps of the P4 algorithm.

B. Bi-directional Point-to-Point Algorithm

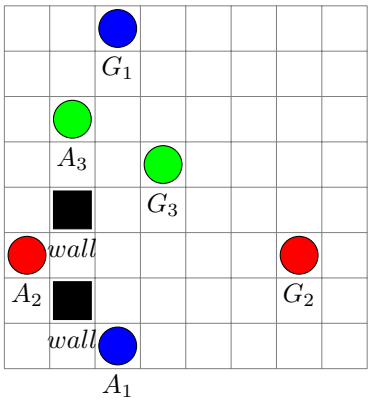
The *Two-Phase Point to Point Algorithm* [19] is a bidirectional search strategy, aimed at efficiently determining the reachability and shortest path from a source to a destination in a given graph. We use this Algorithm in P4 to find the initial paths for agents. Also, we use the length of path to decide that the agent with shorter path will be modified more to find a new path and the agent with longer path will experience relatively less change.

The Two-Phase Point to Point Algorithm is designed for determining the reachability and value at a destination or source point in a graph. The algorithm operates as follows:

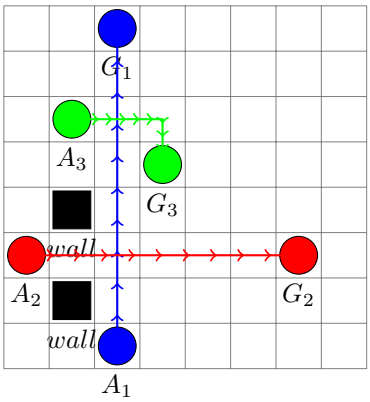
1) **Initialization:** Two sets of active vertices, `FActive` and `BActive`, are initialized for forward and backward searches starting from the source s and destination d, respectively. Arrays `VisitF[]` and `VisitB[]` are used to keep track of visited vertices, and a `safeApprox` initialization is performed for approximating reachability.
2) **Phase 1 - Bidirectional Search:** The algorithm updates the active vertex sets and checks for vertices visited by both forward and backward searches to update their estimates. The phase terminates based on a prediction condition related to the sizes of `FActive` and `BActive`.
3) **Phase 2 - Convergence:** Depending on the prediction made in Phase 1, the algorithm iterates in either the forward or backward direction until the active set is empty. Finally, it returns the reachability status and the value at either d or s.

C. Window-Based Pruning for Conflict Resolution

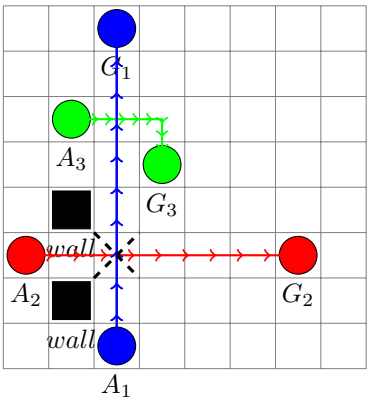
Our approach to window-based pruning is inspired by the work of Vedder and Biswas [12], yet it includes key differences that make it more efficient. Our algorithm, detailed in Algorithm 2, addresses the challenges of conflict resolution between agents A_i and A_j. Unlike the global search window



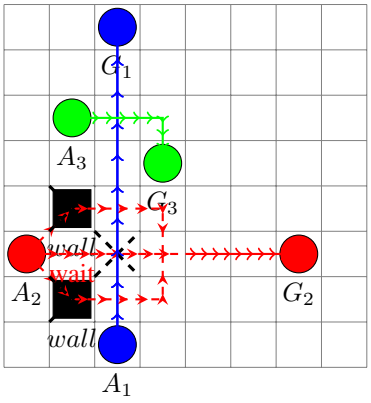
(a) Step 1: Given the grid map and agents with start (A_i) and goal (G_i).



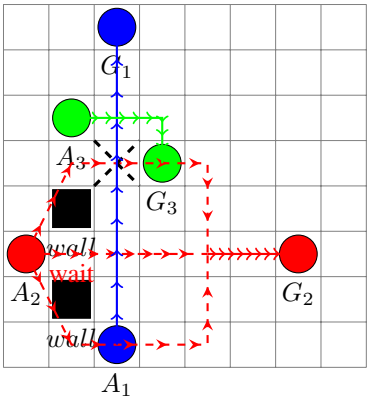
(b) Step 2: Agents start at the same time from initial positions planned by the PnP algorithm.



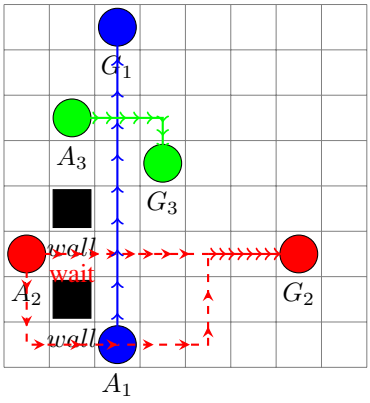
(c) Step 3: Conflict occurs at node (3,3) between A1 and A2.



(d) Step 4: A2 explores possible paths in small window then change of direction or waits.



(e) Step 5: window is expanded to allow A2 to explore more paths and change direction.



(f) Step 6: Comparison between waiting and new path leads to choice of waiting for one step.

Fig. 2: P4 Illustration: The figures represent various stages of the P4 algorithm, detailing the steps involved in its execution.

proposed in [12], our algorithm uses a dynamic, localized window that centers around conflict points. This allows for quicker resolution of conflicts and efficient computation, particularly in environments with sparse obstacles.

1) Overview: For initial path planning, the algorithm employs a basic point-to-point search technique. The window-based pruning comes into play when conflicts arise between agents. The algorithm seeks to resolve these conflicts by dynamically adjusting a local window around the conflict points and finding alternative paths for the lower-priority (with shorter initial path) agent.

2) Algorithm Components and Explanations:

a) Initialization:

- The algorithm starts by identifying the point and time where the conflict occurs between agents A_i and A_j.
- An initial window size ϵ is set to a small value, thereby limiting the search area for quicker conflict resolution.

b) Main Loop:

- The loop continues to execute as long as conflicts exist between the agents.
- A localized window W_ϵ is created around the identified conflict points.
- The algorithm then attempts to find a new local path for the lower-priority agent A_i within this window.
- If a new path is found or A_i can stay in place without moving, the agent's path is updated, and the conflict is resolved.
- If no resolution is found, the window size ϵ is dynamically adjusted within predefined bounds (ϵ_{max} and ϵ_{min}) to explore new paths.

c) Termination Condition:

- The algorithm terminates when either the conflict is resolved or the window size ϵ reaches its maximum limit ϵ_{max}.

3) Mathematical Definitions: Several mathematical constructs are utilized to formalize the problem.

- W_ϵ describes a window of width ϵ around the conflict.
- ϵ_{max} and ϵ_{min} are the bounds for window size ϵ. And δ is used to incrementally adjust the window width initially with one grid unit.

4) Dynamic Window Adjustment: The window size ϵ is dynamically adjusted during the algorithm's execution. It starts with a small value for quick conflict resolution and can be expanded up to ϵ_{max} based on the local environment's

Algorithm 2 Window-Based Pruning

Require: Graph G, agents A_i and A_j, window width ϵ, ϵ_{max}, ϵ_{min}

Determine the conflict point and time for A_i and A_j
$\epsilon \leftarrow$ set to initial small value ▷ Initial window size
while Conflict exists **do**
  Create a local window W_ϵ around conflict point for A_i & A_j
  NEW_PATH $\leftarrow$ FindLocalPath(A_i, W_ϵ)
  if NEW_PATH exists or A_i can wait in place **then**
    Update A_i's path with NEW_PATH
    Resolve conflict and break loop
  else if $\epsilon < \epsilon_{\text{max}}$ **then**
    $\epsilon \leftarrow \epsilon + \delta$ ▷ Increase window size
  else if $\epsilon \geq \epsilon_{\text{max}}$ **then**
    Conflict unresolved; take another action or report failure
    Break loop
  end if
end while
function FINDLOCALPATH(A_i, W_ϵ)
  Find a local path within W_ϵ for A_i or determine if A_i can wait in place **return** NEW_PATH or Wait
end function

complexity and the number of unresolved conflicts. This ensures that the algorithm remains both effective and efficient.

D. Path Direction Prediction

As depicted in Algorithm 3, the path direction prediction is divided into three functions: PathPlanning, AdjustPathsInWindow, and AdjustDirection. These functions form an integral part of our proposed method, leveraging historical path data to predict the future direction of each agent. With the Path Direction Prediction, Our P4 approach can change the path based on the parameters from the initial paths.

The underlying assumption is modeled by:

$$\vec{d}_{t+1} = \alpha \vec{d}_t + (1-\alpha)\vec{h}_t \tag{1}$$

where:

- $\vec{d}_{t+1}$ represents the predicted direction at time $t+1$.
- $\vec{d}_t$ is the actual direction at time t.
- $\vec{h}_t$ is the historical average direction up to time t.
- α is a weighting factor, $0 \leq \alpha \leq 1$, that determines the influence of the current direction over the historical data.

The `PathPlanning` function initializes the OPEN set with the start location and continues to adjust paths within a specified window. This process is repeated until the OPEN set is empty. The `AdjustPathsInWindow` function takes charge of iterating over the nodes in OPEN. It adjusts the direction of nodes whose cost has changed and are located within window W. The direction adjustment is carried out by the `AdjustDirection` function.

The `AdjustDirection` function computes the original and predicted directions and adjusts the deviation while

Algorithm 3 Path Direction Prediction

function PATHPLANNING$(G, s, g, H, W, \alpha, \beta)$
  Initialize set OPEN with start location s
  while OPEN is not empty **do**
    OPEN $\leftarrow$ AdjustPathsInWindow(OPEN, $g, G, H, W, \alpha, \beta$)
  end while
end function
function ADJUSTPATHSINWINDOW(OPEN, $g, G, H, W, \alpha, \beta$)
  Initialize set NEWOPEN
  for all $n \in$ OPEN **do**
    for all $m \in$ neighbors(n) **do**
      if cost(n, m) has changed and $m \in W$ **then**
        $m \leftarrow$ AdjustDirection(m, H, α, β)
        Add m to NEWOPEN
      end if
    end for
  end for
  return NEWOPEN
end function
function ADJUSTDIRECTION(m, H, α, β)
  Compute original direction $\vec{d}_t$ based on H
  Compute predicted direction: $\vec{d}_{t+1} = \alpha\vec{d}_t + (1-\alpha)\vec{h}_t$
  while conflict exists at m **do**
    Adjust deviation: $\vec{p}_{new} = \vec{p}_{old} + \beta\Delta\vec{p}$
    Update m based on $\vec{p}_{new}$
  end while
  return m_{adjusted}
end function

conflicts exist at a given node. This adjustment is based on:

$$\vec{p}_{new} = \vec{p}_{old} + \beta\Delta\vec{p} \tag{2}$$

where:

- $\vec{p}_{new}$ and $\vec{p}_{old}$ are the new and old path vectors, respectively.
- $\Delta\vec{p}$ is the path deviation vector.
- β determines the extent of deviation from the original path.

This gradual adjustment strategy optimizes the solution locally while maintaining the overall direction, aiming for a more efficient and safe path. It not only helps swiftly identify optimal paths but is particularly valuable in dynamic environments, where the path-finding algorithm needs to be re-invoked frequently.

III. EXPERIMENTAL EVALUATION

*a) **Experimental Settings**:* Experiments were conducted on a system with macOS 12.6.1, an Intel Core i9 2.3 GHz CPU, and 16 GB RAM, using Python (version 3.9.7). We evaluated P4, Conflict-Based Search with heuristic (CBSH) [4], Explicit Estimation Conflict-Based Search (EECBS) with bounded-suboptimal [5], Priority-Based Search(PBS) [7], lazy constraints addition search for MAPF (LaCAM) [8], and Priority Planning(PP) [6].

The experimental design was geared towards maximizing completeness and ensuring accurate comparisons.We run 20 instances to get the average result for each number of agents on each map.

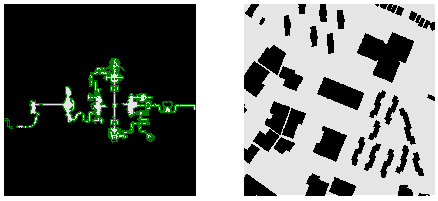
*b) **Metrics**:* Performance was evaluated based on following metrics:

1) **Efficiency**: This was measured as the running time from the start of the algorithm until it found a solution or terminated.
2) **Success Rate**: This refers to the ratio of the number of agents for which path planning was completed within a given time, to the total number of agents. If an algorithm cannot complete path planning for all agents, its success rate can be calculated as the ratio between agents successfully arrived their goals over total agents.
3) **Solution Quality**: This was assessed as the *Sum of Delay* (SoD = $\frac{\sum_{A_i \in \mathcal{A}} (l(P_i))}{\sum_{A_i \in \mathcal{A}} \text{dist}(S_i, G_i)}$) for the algorithms.

*c) **Maps Used**:* Algorithms were evaluated on several types of maps: real-world maps, randomly generated grid maps, and maze grids. Next we present our results.

A. Results for Real Maps

The orz900d is a 1491×656 map with 96,603 available grid points and the NewYork is a 1024×1024 map with 792,676 available grid points. Next we compare P4 with existing methods using these maps.



(a) orz900d (b) NewYork

Fig. 3: Real Maps Used in Evaluation.

*1) **Runtime Performance**:* The runtime performance results are show in Figure 4. We observe that P4 algorithm exhibits robust scalability. For `orz900d`, for 200 agents, P4 is over 4× faster than CBSH. With number of agents increasing , P4's performance advantage over other methods increases. These findings affirm the suitability of the P4 method for complex multi-agent scenarios, particularly in applications where time efficiency is more important and the number of agents is large.LaCAM has a similar running time to P4 but it is not as good as P4, because we find that LaCAM has higher SoD than P4 in most scenarios from Figure 6.

*2) **Success Rates for Real Maps**:* P4 demonstrates a higher success rate than the other tested algorithms, under a fixed time limit. As shown by Figure 5, the success rates of P4 are higher than other methods. As time limit is increased from 0.1 to 100 seconds, success rate of P4 increases from 50% to 100%. On the other hand, for the fast Priority Planning success rate increases from 40% to 80% and for CBSH that tries to find the optimal solution maximum success rate is 40%. The results affirm P4 as a reliable choice for multi-agent path finding, particularly in time-sensitive applications.

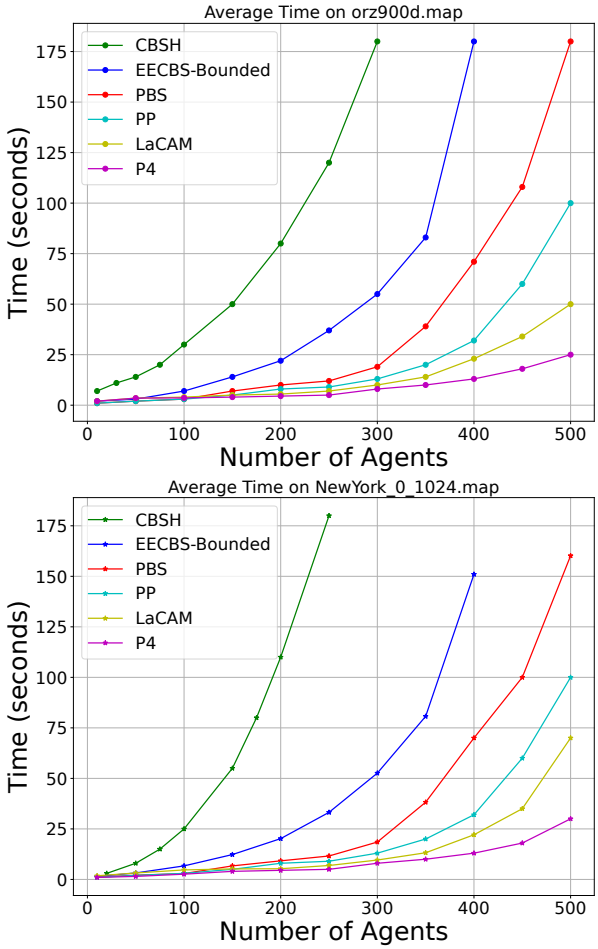


Fig. 4: Comparison of Running Times on Real Maps.

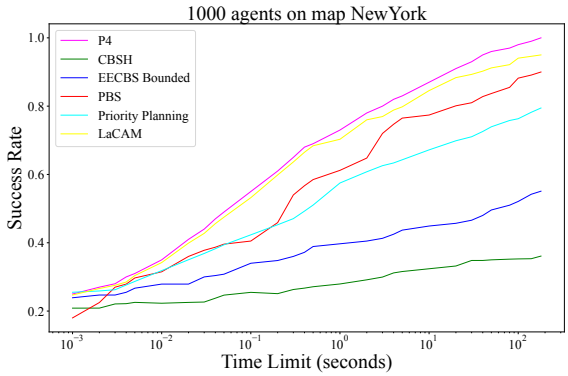


Fig. 5: Comparison of Success Rate on Real Maps.

Our method exhibits the capability to achieve higher success rates more rapidly in a continuous time framework. This characteristic underscores the efficiency of our approach, particularly in scenarios where time responsiveness is crucial, validating the applicability and robustness of our proposed algorithm in various multi-agent path-finding contexts.

*3) **Solution Quality**:* CBSH is an algorithm for optimal path solution while P4 and others are all suboptimal. Therefore we normalize the SoD metric (i.e., the measure of quality) for the suboptimal algorithms with respect to CBSH (optimal) and plot it in Figure 6 for comparison. We noticed that P4 is superior to other suboptimal solutions. We

also observe that, as expected, the suboptimality of P4 with respect to CBSH grows with number of agents due to its local search. However, we observe that for real maps, for 100 and 200 agents, the SoD for P4 is within 5% and 10% of the optimal. While for EECBS-Bounded or LaCAM SoD are 50% and 75% more and for the other methods,like PP the SoD are 2× higher.

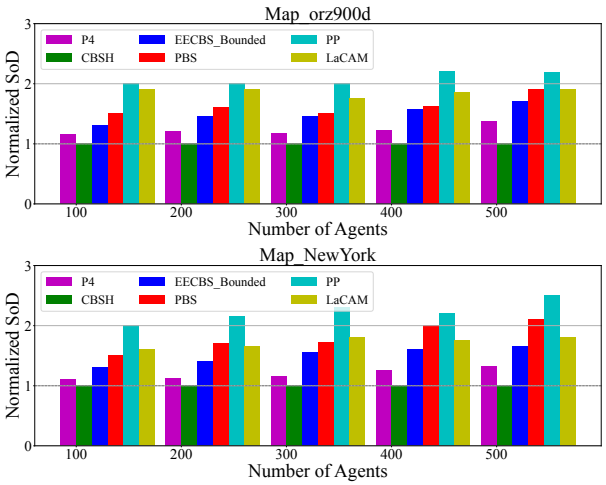


Fig. 6: Comparing Normalized SoD on Real Maps.

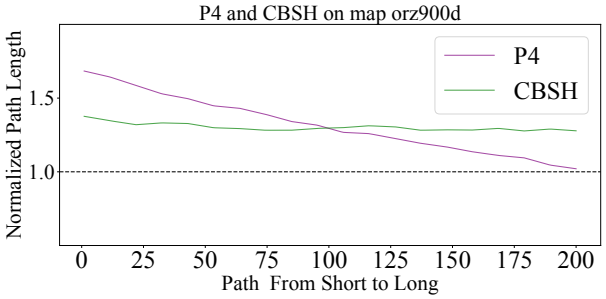


Fig. 7: Normalized (wrt shortest distance) Path Lengths.

Upon analyzing the paths generated by the CBSH and P4 algorithms in Figure 7, we can sort them from shortest to longest for comparison. The results indicate that although CBSH has an advantage in terms of the overall sum of path lengths, P4 manages to come close to CBSH's optimal solution in terms of the total path delay for agents. This is achieved by P4 making more adjustments to shorter paths and fewer modifications to longer ones. This suggests that P4 is effective in maintaining a competitive total path delay, even if its overall path lengths may be slightly longer. The shortest path in P4 is just 18% longer than the one in CBSH solution while the longest path in P4 is shorter than the one in CBSH by 13%.

These findings confirm the efficacy of P4 in tackling complex multi-agent path finding problems, outperforming state-of-the-art methods across various metrics and scenarios.

B. Results for Randomly Generated Grids and Maze Maps

For random grids of sizes 32×32 and 64×64 with obstacle ratios (ORs) of 10% and 20%, the results are presented in TableI. The experiment on random maps with different obstacle ratios evaluates the algorithms' adaptability in both sparse and dense environments.

TABLE I: Time and SoD Comparisons of Algorithms for MAPF on Random and Maze maps.

Map, number of agents	**Algorithm**	**Time**	**SoD**
Random-32×32-10(OR), 100	P4	2.1s	1.05
	CBSH	12.1s	1.0
	EECBS-Bounded	5s	1.05
	PBS	4s	1.1
	Priority Planning	3.4s	1.3
	LaCAM	2.1s	1.25
Random-32×32-20(OR), 100	P4	3.5s	1.10
	CBSH	18.2s	1.0
	EECBS-Bounded	15s	1.06
	PBS	6s	1.2
	Priority Planning	4.8s	1.40
	LaCAM	3.2s	1.55
Random-64×64-10(OR), 200	P4	2.5s	1.05
	CBSH	15.2s	1.0
	EECBS-Bounded	7s	1.06
	PBS	5s	1.3
	Priority Planning	3.8s	1.5
	LaCAM	2.2s	1.35
Random-64×64-20(OR), 200	P4	2.5s	1.1
	CBSH	20s	1.0
	EECBS-Bounded	8s	1.05
	PBS	6.5s	1.4
	Priority Planning	3s	1.5
	LaCAM	2s	1.3
maze-32×32, 100	P4	6.5s	1.15
	CBSH	Timeout	
	EECBS-Bounded	65s	1.1
	PBS	45s	1.3
	Priority Planning	10s	1.7
	LaCAM	5s	1.5
maze-128×128, 100	P4	3s	1.1
	CBSH	Timeout	
	EECBS-Bounded	25s	1.02
	PBS	15s	1.3
	Priority Planning	5s	1.6
	LaCAM	4s	1.55

From the data in Table I, we observe the following:

- In 32×32 grids with 100 agents, P4 is the most time-efficient, outperforming CBSH by approximately $5\times$. LaCAM is slightly faster but with a higher SoD value of about 20% or more.
- In 64×64 grids with 200 agents, P4 scales well, especially in denser environments (20% obstacles), where it is $7\times$ faster than CBSH and keep a lower SoD than other sub-optimal method.

These results highlight the balanced performance of the P4 algorithm in execution time and solution quality on random maps, making it an effective approach for multi-agent path planning.

In maze maps, the possible path is very narrow and it is hard for all methods to find the path. Especially, CBSH methods cannot compute the solution within the time bound. On the other hand, sub-optimal solutions are able to generate the solution though the Sum-of-Delay is larger as the number of agents increases. This is because they benefit from the underlying priority based scheduling used to handle agents one by one.

TABLE II: Ablation Experiment on P4

Experiment	Time	Conflicts	SoD
W/O PnP	$\uparrow 150\%$	$\uparrow 15\%$	$\uparrow 25\%$
W/O WP	$\uparrow 85\%$	$\uparrow 45\%$	$\uparrow 55\%$
W/O PDP	$\uparrow 65\%$	$\uparrow 90\%$	$\uparrow 75\%$

C. *Ablation Experiment and Analysis*

We do the ablation experiment for our approach by replacing three main components: Bi-directional Point-to-Point Algorithm, Window-Based Pruning and Path Direction Prediction with the base approach as Single Agent One-Way Path Planning (W/O PnP), no Window-Based Pruning (W/O WP), and no Path Direction Prediction (W/O PDP). The ablation experiment is on empty 128×128 map with 500 agents. The results are in the Table II and are normalized with the original P4 experiment result.

The ablation experiment results shows that these three components are all important for the whole approach. Specially, Bi-directional Point-to-Point Algorithm will improve the P4 algorithm running time $1.5\times$. Window-Based Pruning and Path Direction Prediction work in reducing the conflicts and sum of delays (SoD).

P4 vs. CBSH: CBSH is a variant of the CBS optimal solver that guarantees both the discovery of the optimal solution and the completeness of the solving process. P4 tends to perform more tuning on shorter path which means with low-priority in P4 compared to CBSH, and less on longer path which means with high-priority in P4. This allows P4 to get closer to the optimal solution quality of CBSH under different conditions and constraints, and the length of higher priority paths can be close to or even better than some of the paths in CBSH, results are shown in Figure 7. Also as the number of agents increases, P4 demonstrates better scalability, reducing the runtime by more than 50%. In sparse maps, P4's local window technique is advantageous, while in dense areas, its point-to-point approach is effective. Overall, P4 is more scalable than CBSH in multi-agent path planning.

P4 vs. other sub-optimal methods P4 outperforms other sub-optimal methods in terms of solution quality and running time. By employing local path adjustment, P4 is able to reduce the total delays in scenarios with a large number of agents. P4 achieves a better balance between algorithm solution time and solution quality. It ensures that the solution quality approaches the optimal path by using local windows and priority planning based on the initial path length. Meanwhile, the solution time is significantly reduced through point-to-point algorithms and dynamic window expansion. The advantage of P4 stems from the local path adjustments we implemented. This helps it maximize the utilization of existing paths and minimize additional search.

IV. Conclusion

In this paper, we propose the P4 method for the Multi-Agent Path Finding problem, a method that can effectively scale to a large number of agents. P4 enhances computational efficiency and accuracy by integrating techniques like pruning, predictive priority planning, and dynamic windows. This greatly strengthens the scalability for handling multiple agents. We have assessed the P4 algorithm through extensive experiments on various metrics, including running time, solution quality, and success rate within a limited time frame. Additionally, the high scalability of P4 provides a solid foundation for exploration in large-scale fields like artificial intelligence and robotics. Future work should focus on further optimizing P4 and exploring its application in more diverse and challenging environments.

Acknowledgment This work is supported by NSF grants CCF-2226448 and CCF-2002554 to UC Riverside.

References

[1] R. Stern, N. Sturtevant, A. Felner, et al., "Multi-agent pathfinding: Definitions, variants, and benchmarks," in Proc. Int. Symp. Combinatorial Search, vol. 10, 2019, pp. 151–158.

[2] R. Stern, et al., "Multi-agent path finding–an overview," in Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures, 2019, pp. 96–115.

[3] O. Salzman and R. Stern, "Research challenges and opportunities in multi-agent path finding and multi-agent pickup and delivery problems," in Proc. 19th Int. Conf. Autonomous Agents and MultiAgent Systems, 2020, pp. 1711–1715.

[4] J. Li, et al., "Pairwise symmetry reasoning for multi-agent path finding search," Artificial Intelligence, vol. 301, 2021, Art. no. 103574.

[5] J. Li, W. Ruml, and S. Koenig, "EECBS: A bounded-suboptimal search for multi-agent path finding," in Proc. AAAI Conf. Artificial Intelligence, vol. 35, 2021, pp. 12353–12362.

[6] M. Phillips, V. Hwang, S. Chitta, and M. Likhachev, "Prioritized planning algorithms for trajectory coordination of multiple mobile robots," IEEE Trans. Robotics, vol. 27, no. 6, pp. 1163–1173, 2011.

[7] H. Ma, D. Harabor, P. J Stuckey, J. Li, and S. Koenig, "Searching with consistent prioritization for multi-agent path finding," in Proc. AAAI Conf. Artificial Intelligence, vol. 33, 2019, pp. 7643–7650.

[8] K. Okumura, "Lacam: Search-based algorithm for quick multi-agent pathfinding," in Proc. AAAI Conf. Artificial Intelligence, vol. 37, 2023, pp. 11655–11662.

[9] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," Artificial Intelligence, vol. 219, pp. 40–66, 2015.

[10] A. Felner, J. Li, et al., "Adding heuristics to conflict-based search for multi-agent path finding," in Proc. Int. Conf. Automated Planning and Scheduling, vol. 28, 2018, pp. 83–87.

[11] A. Felner, R. Stern, et al., "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges," in Proc. Int. Symp. Combinatorial Search, vol. 8, 2017, pp. 29–37.

[12] K. Vedder and J. Biswas, "X*: Anytime multi-agent path finding for sparse domains using window-based iterative repairs," Artificial Intelligence, vol. 291, 2021, Art. no. 103417.

[13] J. Li, et al., "New techniques for pairwise symmetry breaking in multi-agent path finding," in Proc. Int. Conf. Automated Planning and Scheduling, vol. 30, 2020, pp. 193–201.

[14] E. Boyarski, A. Felner, G. Sharon, and R. Stern, "Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding," in ICAPS, 2015, pp. 47–51.

[15] J. Li, A. Felner, E. Boyarski, H. Ma, and S. Koenig, "Improved heuristics for multi-agent path finding with conflict-based search," in IJCAI, 2019, pp. 442–449.

[16] J. Li,et al., "Disjoint splitting for conflict-based search for multi-agent path finding," in ICAPS, 2019, pp. 279–283.

[17] D. Onken, L. Nurbekyan, et al., "A neural network approach for high-dimensional optimal control applied to multiagent path finding," IEEE Trans. Control Systems Technology, vol. 31, no. 1, pp. 235–251, 2021.

[18] W. Chen, J. Wang, M. Ma,et al., "Multiagent path finding using deep reinforcement learning coupled with hot supervision contrastive loss," IEEE Trans. Industrial Electronics, 2023.

[19] C. Xu, K. Vora, and R. Gupta, "PNP: Pruning and prediction for point-to-point iterative graph analytics," in ASPLOS '19, 2019, pp. 587–600.