Probabilistic Robust Multi-Agent Path Finding

Dor Atzmon,[1] Roni Stern,[1,2] Ariel Felner,[1] Nathan R. Sturtevant,[3] Sven Koenig[4]
[1]Ben-Gurion University of the Negev, [2]Palo Alto Research Center (PARC), [3]University of Alberta, [4]USC
dorat@post.bgu.ac.il, sternron@post.bgu.ac.il, felner@bgu.ac.il, nathanst@ualberta.ca, skoenig@usc.edu

Abstract

In a multi-agent path finding (MAPF) problem, the task is to move a set of agents to their goal locations without conflicts. In the real world, unexpected events may delay some of the agents. In this paper, we therefore study the problem of finding a p-robust solution to a given MAPF problem, which is a solution that succeeds with probability at least p, even though unexpected delays may occur. We propose two methods for verifying that given solutions are p-robust. We also introduce an optimal CBS-based algorithm, called pR-CBS, and a fast suboptimal algorithm, called pR-GCBS, for finding such solutions. Our experiments show that a p-robust solution reduces the number of conflicts compared to optimal, non-robust solutions.

1 Introduction

The *Multi-Agent Path Finding* (MAPF) problem is defined by a graph $G = (V, E)$ and a set of n agents $\{a_1, \dots a_n\}$, where each agent a_i has a start location $s_i \in V$ and a goal location $g_i \in V$. At each time step, an agent can either *move* to an adjacent location or *wait* in its current location. The task is to find a sequence of adjacent locations for each agent a_i (single-agent plan) that moves it from s_i to g_i such that agents do not *conflict* (or synonymously, collide), i.e., occupy the same location or swap locations at the same time step. MAPF has applications in video games, traffic control, and robotics (Felner et al. 2017; Ma and Koenig 2017). In many cases, there is also a requirement to minimize a given cost function, such as the sum of costs incurred by all agents before reaching their goal locations. Solving MAPF optimally is NP-hard (Yu and LaValle 2013; Surynek 2010). Nonetheless, practical optimal algorithms exist, some even capable of finding optimal solutions for more than one hundred agents (Wagner and Choset 2015; Surynek 2012; Felner et al. 2018; Lam et al. 2019).

In practice, unexpected events may delay some of the agents, preventing them from following their pre-determined single-agent plans. When such an event occurs, the MAPF solution should be adjusted to avoid conflicts. Such re-planning may require costly computing and communication capabilities or may even be impossible. Thus, it is often desirable to generate a *robust solution* that can withstand such unexpected events, avoiding the need for re-planning if they occur, especially in safety-critical settings such as air traffic control.

To this end, Atzmon et al. (2018) introduced the notion of a *k-robust solution*. A solution is k-robust iff each agent can be delayed up to k times and still no conflict will occur. This notion is especially suitable in cases where the maximum number of delays is known in advance. In this paper, we explore a different, novel form of robustness called p-robust. A solution is p-robust iff the probability that no conflict will occur is *larger than or equal to* $0 \leq p \leq 1$. This notion is, arguably, a more realistic form of robustness, since it does not assume a strict limit on the number of unpredictable delays per agent. Previously proposed MAPF algorithms that consider delay probabilities, such as UM* (Wagner and Choset 2017) and Approximate Minimization in Expectation (AME) (Ma, Kumar, and Koenig 2017), do not return a p-robust solution (see Section 8 for details).

We propose pR-CBS, an algorithm based on *Conflict-Based Search* (CBS) (Sharon et al. 2015), that finds optimal p-robust solutions. Whereas classic CBS resolves a conflict by disallowing it (using a two-way split function), pR-CBS also considers to allow it. This results in a novel three-way split function. We also propose pR-GCBS, a fast suboptimal version of pR-CBS that reduces the planning time while keeping the cost close to optimal. A challenge of both algorithms is how to verify that a given solution is p-robust. We propose two methods for doing so and prove their correctness. Finally, we evaluate our algorithms experimentally, showing that they indeed find p-robust solutions and that executing p-robust solutions can reduce the number of conflicts compared to executing optimal, non-robust solutions.

2 Definitions and Background

A single-agent plan π_i for agent a_i is a sequence of locations from location s_i to location g_i. $\pi_i(t)$ denotes the location of agent a_i at time step t (assuming no delays). Thus, $\pi_i(0) = s_i$ and $\pi_i(|\pi_i| - 1) = g_i$. In a single-agent plan π_i, between every two consecutive locations $\pi_i(t)$ and $\pi_i(t+1)$, the agent can perform either a *move* action, i.e., locations

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

$\pi_i(t)$ and $\pi_i(t+1)$ are adjacent locations, or *wait* action, i.e., $\pi_i(t) = \pi_i(t+1)$. All actions have a unit cost. A solution to a MAPF problem instance is a set of single-agent plans $\pi = \{\pi_1, \ldots \pi_n\}$.

Definition 1 (Conflict). *A conflict $\langle a_i, a_j, x, t\rangle$ in a solution π occurs iff agents a_i and a_j ($i \neq j$) occupy the same location x at time step t, i.e., when $\pi_i(t) = \pi_j(t) = x$ (vertex conflict), or when they traverse the same edge x in opposite directions from time step $t-1$ to time step t, i.e., when $(\pi_i(t-1) = \pi_j(t)) \wedge (\pi_i(t) = \pi_j(t-1))$, where $(\pi_i(t-1), \pi_i(t)) = x$ (swapping conflict).* [1]

We say that a solution π is **valid** iff it is conflict-free. A MAPF solver is *sound* iff it outputs a valid solution. In this work, we allows following and cycle conflicts in a valid solution (Stern et al. 2019). A following conflict occurs iff agents a_i and a_j ($i \neq j$) occupy the same location at time steps t and $t+1$, respectively, i.e., $\pi_i(t) = \pi_j(t+1)$, and a cycle conflict occurs iff each pair of agents in a set of agents $\{a_1, a_2, \ldots, a_l\}$ has a following conflict, in a cycle, i.e., $\pi_1(t) = \pi_2(t+1) \wedge \pi_2(t) = \pi_3(t+1) \wedge \cdots \wedge \pi_l(t) = \pi_1(t+1)$. Also, we assume that two agents cannot conflict while traversing two different edges. We say that π is **optimal** iff it is valid and has the lowest cost among all valid solutions. In this paper, we focus on minimizing the *sum-of-costs* (SOC) cost function, which is the sum of the number of actions (including wait actions) performed by all agents until all agents reach their goal locations and stay there, i.e., $\sum_{i=1}^{n}(|\pi_i| - 1)$.

2.1 Conflict-Based Search

Conflict-Based Search (CBS) (Sharon et al. 2015) is a commonly used MAPF solver that has two levels. The high level of CBS searches the binary *constraint tree* (CT). Each node $N \in CT$ contains: **(1)** a set of constraints imposed on the agents ($N.constraints$); **(2)** a single solution ($N.\pi$) that is *consistent* with (i.e., satisfies) all constraints; and **(3)** the cost of solution $N.\pi$ ($N.cost$). CBS imposes two types of constraints. A vertex constraint is a tuple $\langle a_i, v, t\rangle$, meaning that agent a_i is prohibited from occupying location v at time step t. An edge constraint is a tuple $\langle a_i, e, t\rangle$, meaning that agent a_i is prohibited from moving along edge e at time step t. The root node contains an empty set of constraints. The high level performs a best-first search on the CT, expanding the nodes in increasing order of their costs.

Generating a node in the CT. Given a node N, the low level of CBS finds a shortest single-agent plan for each agent from its start location to its goal location that satisfies all constraints of node N imposed on the agent.

Expanding a node in the CT. Once CBS has chosen node N for expansion, it checks its solution $N.\pi$ for conflicts. If it is conflict-free, then node N is a goal node and CBS returns its solution. Otherwise, CBS *splits* node N on one of the conflicts $\langle a_i, a_j, x, t\rangle$ by generating two children for node N. Each child node has a set of constraints that is the union of $N.constraints$ and a new constraint. The new constraint is either $\langle a_i, x, t\rangle$ or $\langle a_j, x, t\rangle$, where x is the vertex or edge the conflict $\langle a_i, a_j, x, t\rangle$ refers to.

[1] We follow the terminology of Stern et al. (2019).

2.2 Multi-Agent Path Finding with Delays

We follow the terminology of Atzmon et al. (2018) to define what delays are, how they affect the execution of a MAPF solution, and how to handle them by creating a robust solution before execution.

A *delay* in a MAPF solution π is a tuple $\langle \pi, a_i, t\rangle$ with $t \geq 1$, representing that agent a_i stays idle in location $\pi_i(t-1)$ instead of performing the move action from location $\pi_i(t-1)$ at time step $t-1$ to location $\pi_i(t)$ at time step t. A solution π is *robust* to a delay iff the delayed agent can continue to follow its single-agent plan after the delay without causing a conflict. To define this formally for a solution π and a delay $D = \langle \pi, a_i, t'\rangle$, let $D(\pi)$ be the solution after experiencing the delay. $D(\pi)$ is identical to π except that π_i is replaced with:

$$\pi_i'(t) = \begin{cases} \pi_i(t) & t < t' \\ \pi_i(t-1) & t \geq t'. \end{cases} \tag{1}$$

Definition 2 (Robust Solution). *A solution π is* robust *to a delay D iff the solution $D(\pi)$ is valid. A solution π is* robust *to a set of delays $\mathcal{D} = \{D_1, \ldots, D_r\}$ iff the solution $D_r(D_{r-1}(\ldots D_1(\pi)\ldots))$ is valid.*

In real-life, the number of delays may be unlimited. For example, an agent might be stuck and stay idle in its start location forever. Therefore, for every type of robustness, some bound on the degree of robustness must be given. One type of robustness is k-robustness (Atzmon et al. 2018), which is summarized in the next paragraph. In this paper, we introduce a new type of robustness, called p-robustness, which uses some of the building blocks of k-robustness.

k-Robust MAPF. A *k-robust MAPF* (kR-MAPF) solver returns a *k-robust* solution, if possible, which is a valid solution that is robust to any set of delays that contains at most k delays for each agent. Classic MAPF is a special case of kR-MAPF for $k = 0$.

Definition 3 (k-Delay-Conflict). *A* k-delay conflict *$\langle a_i, a_j, x, t\rangle$ in a solution π occurs iff there exists a $\Delta \in \{0, \ldots k\}$ such that agents a_i and a_j ($i \neq j$) occupy the same location x at time steps t and $t+\Delta$, respectively, i.e, $\pi_i(t) = \pi_j(t+\Delta) = x$, or when they traverse the same edge x in opposite directions from time step $t-1$ to time step t for agent a_i and from time step $t+\Delta-1$ to time step $t+\Delta$ for agent a_j, i.e., when $(\pi_i(t-1) = \pi_j(t+\Delta)) \wedge (\pi_i(t) = \pi_j(t+\Delta-1))$, where $(\pi_i(t-1), \pi_i(t)) = x$.*

A solution is k-robust iff it does not contain any k-delay conflicts. A classic conflict is a special case of a k-delay conflict for $k = 0$. We use k-delay conflicts in Section 5, as part of verifying whether a solution is p-robust.

3 p-Robust MAPF

A limitation of k-robust MAPF is that one needs to set k appropriately. Setting k too low can result in too many conflicts during execution of a MAPF solution, while setting k too high can increase the planning time and resulting solution cost.

One way of approaching this issue is to consider additional information about the problem at hand. In particular, we assume that we are given the *delay probability* p_d, i.e., the probability that an agent will be delayed at a given time step instead of performing a move action. We also assume that such delays are independent of each other, i.e., a delay occurring does not affect the probability of other delays occurring. For simplicity, we further assume that the delay probability is constant across all agents, locations, and time steps, although this assumption can be easily generalized. In fact, some of our algorithms are already suitable for any stochastic movement of the agents. Under these assumptions, avoiding all k-delay conflicts for a fixed value of k may be ineffective. Instead, it makes more sense to resolve conflicts based on their probabilities of occurring, rather than a fixed k.

In this work, we explore a new form of robustness, p-robustness, that considers such additional information.

Definition 4 (p-Robust Solution)**.** *A solution π is p-robust iff the probability that it will be executed without a conflict is at least p.*

The p-robust MAPF (pR-MAPF) problem is the problem of finding a p-robust solution for a given p ($0 \leq p \leq 1$). Different from the kR-MAPF problem, setting the parameter is now related to the probability that the solution execution succeeds. Given the cost of a failure execution and the benefit of a successful one, in some cases, the parameter of pR-MAPF can be determined by a cost-benefit analysis.

We now present necessary and sufficient conditions for a solution to be p-robust. Let π be a solution with m move actions and $\mathcal{D} = \{D_1, \dots D_r\}$ be a set of r delays. The probability $P(\pi, \mathcal{D})$ for experiencing exactly the delays in $\mathcal{D}$ and no other delays while executing π is:

$$P(\pi, \mathcal{D}) = {p_d}^r \cdot (1 - p_d)^m \tag{2}$$

since the agents experience r delays when attempting to execute move actions, each with probability p_d, and do not experience delays when executing m move actions, each with probability $1 - p_d$. Let $\mathcal{RD}(\pi)$ be the set of all sets of delays that a solution π is robust to (Definition 2). The probability $P_0(\pi)$ that one of the sets of delays in $\mathcal{RD}(\pi)$ occurs is:

$$P_0(\pi) = \sum_{\mathcal{D} \in \mathcal{RD}(\pi)} P(\pi, \mathcal{D}). \tag{3}$$

A solution is p-robust iff $P_0(\pi) \geq p$. However, computing $P_0(\pi)$ as defined in Equation 3 is impossible, since the cardinality of $\mathcal{RD}(\pi)$ may be infinite. We thus define the notion of *potential conflicts*.

Definition 5 (Potential Conflict)**.** *A solution π has a potential conflict $C = \langle a_i, a_j, x, t \rangle$ iff there exists a $\Delta(C) \geq 0$ such that $\langle a_i, a_j, x, t \rangle$ is a $\Delta(C)$-delay conflict.*

A potential conflict $C = \langle a_i, a_j, x, t \rangle$ *occurs* iff agent a_i experiences $d_i \geq \Delta(C)$ delays before successfully performing its tth action and agent a_j experiences $d_i - \Delta(C)$ delays before successfully performing its $t + \Delta(C)$ action. The agents conflict at time step $t + d_i$ since $\pi_i(t) = \pi_j(t + \Delta(C))$.

A key observation is that a set of delays $\mathcal{D}$ is not in $\mathcal{RD}(\pi)$ iff it causes a potential conflict to occur. Thus, $1 - P_0(\pi)$ can be calculated by calculating the probability that at least one potential conflict occurs. Computing this probability is challenging since the probability with which the different potential conflicts occur are not independent, despite the fact that delays occur independently (with probability p_d). To see this, consider a solution π with two potential conflicts involving the same pair of agents. Clearly, these two potential conflicts cannot both occur, because, when the first one occurs, the agents cannot continue. An in-depth discussion of this conflict dependency is provided by Wagner and Choset (2017). Nevertheless, we provide practical ways of calculating and estimating $P_0(\pi)$ in Section 5 by computing the probability that at least one potential conflict occurs.

4 p-Robust CBS

Solution π is **optimal** for pR-MAPF iff it is p-robust and has the lowest cost among all p-robust solutions. Solving MAPF optimally is NP-hard (Yu and LaValle 2013; Surynek 2010). MAPF is a special case of p-robust MAPF for $p_d = 0$ and $p = 1$, and thus solving pR-MAPF optimally is also NP-hard. Next, we describe pR-CBS (Algorithm 1), a CBS-based algorithm that finds p-robust solutions. pR-CBS is complete and optimal. First, pR-CBS creates a root CT node with no constraints and calculates an initial solution for the agents as classic CBS does (Line 2). Then, it inserts that node into the OPEN list (denoted by OPEN; Line 3). pR-CBS is unique in how it handles CT nodes, chooses conflicts, and splits CT nodes, as explained next.

4.1 Handling a CT Node

When a CT node N is chosen for expansion (Line 5), $N.\pi$ is sent to a verifier (Line 6) that returns TRUE if the solution is p-robust ($P_0(\pi) \geq p$) and FALSE otherwise, which is similar to the goal test in classic CBS. We discuss implementations of specific verifiers in Section 5. If the verifier returns TRUE, then the CT node is a goal node (Line 7). In this case, the search terminates and $N.\pi$ is returned. If the verifier returns FALSE, then further high-level CT node expansions are required.

4.2 Choosing a Conflict

pR-CBS checks $N.\pi$ for potential conflicts. Choosing which potential conflict to resolve cannot affect the correctness of pR-CBS, as is explained below. In our implementation, we choose the potential conflict C with the lowest $\Delta(C)$, i.e., the difference in arrival time steps of the two agents at the same location. Ties are broken in favor of the earliest time step t. Conflicts with low values of $\Delta(C)$ and t are likely to have a high probability of occurring.

4.3 Splitting a CT Node

Consider a conflict $\langle a_i, a_j, x, t \rangle$ in solution $N.\pi$ at location or edge x for classic CBS. There are three possible cases: (1) a_i does not occupy x at time step t. To enforce this case, we can add the constraint $\langle a_i, x, t \rangle$. (2) a_j does not occupy x at time step t. To enforce this case we can add the constraint

Algorithm 1: High level of pR-CBS

1 **Main(MAPF problem** $instance$**)**
2 Init $Root$ with an initial solution and no constraints
3 Insert $Root$ into OPEN
4 **while** OPEN *not empty* **do**
5 $N \leftarrow$ Pop the node with the lowest cost in OPEN
6 **if** *Verify(*$N.\pi$*)* **then**
7 **return** $N.\pi$ // *N is goal*
8 $C(a_1, a_2, t) \leftarrow$ get-conflict(N)
9 $\Delta \leftarrow$ get-delta(C)
10 $x \leftarrow$ get-conflicted-location-or-edge(C)
11 $A_1 \leftarrow$ GenChild(N,negConst(a_1, x, t))
12 $A_2 \leftarrow$ GenChild(N,negConst(a_2, x, $t+\Delta$))
13 $A_3 \leftarrow$ GenChild(N,posConst(a_1, a_2, x, t, $t+\Delta$))
14 Insert A_1, A_2, and A_3, into OPEN
15 **return** No Solution
16 **GenChild(Node** N**, Constraint** $NewCons$**)**
17 $A.constraints \leftarrow N.constraints \cup \{NewCons\}$
18 $A.\pi \leftarrow N.\pi$
19 **if** *$NewCons$ is a negative constraint* **then**
20 Update $A.\pi$ to satisfy $A.constraints$
21 $A.cost \leftarrow SOC(A.\pi)$
22 **return** A

$\langle a_j, x, t\rangle$. (3) Both agents occupy x at time step t. Since this case is illegal, CBS splits node N according to Cases 1 and 2 only, thereby eliminating Case 3.

A fundamental difference between CBS and pR-CBS is that pR-CBS cannot eliminate Case 3. Case 3 is no longer illegal because the probability for that conflict to actually occur might be small enough to result in p-robust solutions.

Let N be a non-goal CT node selected for expansion by pR-CBS, let $C = \langle a_i, a_j, x, t\rangle$ be the chosen unresolved potential conflict in $N.\pi$. To resolve C, pR-CBS generates two children of N with the additional constraints $\langle a_i, x, t\rangle$ and $\langle a_j, x, t + \Delta(C)\rangle$, respectively (Lines 11-12). These are *negative constraints*, i.e., constraints that prevent the agents from occupying a certain location at a certain time step or traverse a certain edge in opposite directions between two consecutive time steps (Cases 1 and 2 above). However, as mentioned above, an optimal p-robust solution might contain this potential conflict (Case 3). Therefore, pR-CBS generates a third child of N with the additional *positive constraint* (Li et al. 2019) $\langle a_i, a_j, x, t, t + \Delta(C)\rangle$, that forces this potential conflict to occur by specifying that agent a_i *must* occupy location x at time step t and agent a_j must occupy location x at time step $t + \Delta(C)$, or that agent a_i must traverse edge x between time steps $t-1$ and t and agent a_j must traverse edge x in an opposite direction between time step $t + \Delta(C) - 1$ and $t + \Delta(C)$ (Line 13). After imposing a positive constraint, this conflict is marked as revolved and will not be selected again in this subtree. This third child is not a goal node as it contains the same solution as node N, but a descendant of it might be a goal node. Finally, pR-CBS inserts the three children into OPEN (Line 14) and iterates.

Example. Figure 1(a) shows an example of a pR-MAPF problem instance. Each of the three agents a_i needs to

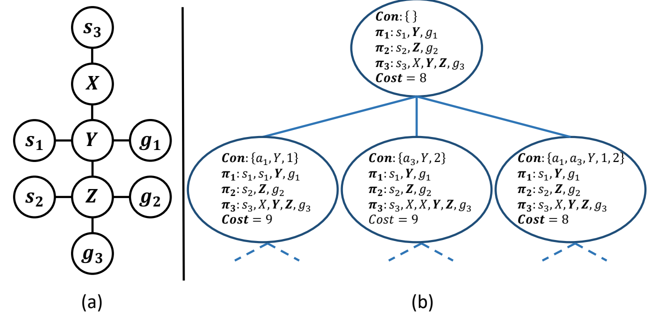

Figure 1: Example of a pR-CBS constraint tree.

move from s_i to g_i. Figure 1(b) shows the corresponding CT. The root of the CT contains the shortest single-agent plans of the three agents. There are two potential conflicts in the root, namely $C_Y = \langle a_1, a_3, Y, 1\rangle$ (in location Y) with $\Delta(C_Y) = 1$ and $C_Z = \langle a_2, a_3, Z, 1\rangle$ (in location Z) with $\Delta(C_Z) = 2$. Assume that the verifier returned that the root is not a goal node, i.e., the solution is not p-robust since the probability of no conflicts is less than p. pR-CBS splits the root according to the conflict $C_Y = \langle a_1, a_3, Y, 1\rangle$ because $\Delta(C_Y) \leq \Delta(C_Z)$. It resolves C_Y by adding the constraints $\langle a_1, Y, 1\rangle$, $\langle a_3, Y, 2\rangle$, and $\langle a_1, a_3, Y, 1, 2\rangle$, each generating a new CT node. After new single-agent plans for the agents are returned from the low level, it chooses the node with the lowest cost for the next expansion, and performs another expansion.

4.4 Theoretical Properties

A verifier (discussed below) is sound iff it correctly determines whether a solution is p-robust. Given a sound verifier, pR-CBS is sound. Its completeness and optimality are proven next.

Lemma 1 (Completeness)**.** *If a p-robust solution exists, pR-CBS returns a p-robust solution.*

Proof outline. pR-CBS performs a best-first search on the CT, where the cost cannot decrease, i.e., newly generated CT nodes cannot have lower cost than the current lowest costs of any CT node in OPEN. The number of solutions with any given cost and the number of potential conflicts of each solution are finite. Hence, the number of CT nodes with any given cost is finite and, after expanding a finite number of CT nodes, pR-CBS finds a p-robust solution, if one exists.

Lemma 2 (Optimality)**.** *When pR-CBS returns a solution, the returned solution has the lowest cost among all p-robust solutions.*

Proof outline. pR-CBS never eliminates solutions by splitting a CT node. It performs a best-first search on the CT where the costs cannot decrease. Thus, the cost of an expanded node is a lower bound on the cost of all p-robust solutions, and the first expanded node with a p-robust solution contains the p-robust solution with the lowest cost.

5 Verifiers

Next, we describe two verifiers that can be called on Line 6 of Algorithm 1. They determine whether $P_0(\pi) \geq p$ for a given solution π. The first verifier, which we refer to as the *Deterministic Verifier*, performs a search over the different sets of delays and calculates the probability of conflicts occurring. The second verifier, which we refer to as the *Monte-Carlo Verifier*, executes simulations to determine the probability of no conflicts occurring and performs a statistical hypothesis test.

5.1 Deterministic Verifier

The Deterministic Verifier, described on Lines 1-10 of Algorithm 2, uses the following two functions: The first function, $P(\pi, d)$, calculates the probability that each agent experiences at most d delays while trying to execute solution π. The second function, $P_0(\pi, d)$, calculates the probability that no conflicts occur during solution execution, given that each agent experiences at most d delays. Using these two functions, the Deterministic Verifier iterates over different values of d until it is able to verify whether solution π is p-robust. The number of delays that each agent can experience is unlimited. Bounding this number by d is done only to compute upper and lower bounds on $P_0(\pi)$, as we now explain in more detail.

Calculating $P(\pi, d)$. The Deterministic Verifier calculates the probability $P(\pi_i, d)$ of each agent experiencing at most d delays during execution (Line 4), as follows: Consider a single agent a_i that executes m_i move actions during execution. There are $\binom{r+m_i-1}{r}$ different ways of inserting r (indistinguishable) delays into an action sequence that contains m_i move actions, since a delay can occur only before a move action or another delay. Consider a specific way of inserting r delays into the action sequence. The probability of this way occurring during execution is, similar to Equation 2, ${p_d}^r \cdot (1-p_d)^{m_i}$. Now consider all different ways of inserting r delays into the action sequence. The probability of any of these ways occurring during execution or, equivalently, the probability of agent a_i experiencing exactly r delays during execution is, ${p_d}^r(1-p_d)^{m_i}\binom{r+m_i-1}{r}$. The probability of agent a_i experiencing at most d delays during execution is

$$P(\pi_i, d) = \sum_{r=0}^{d} {p_d}^r (1-p_d)^{m_i} \binom{r+m_i-1}{r}, \quad (4)$$

where m_i is the number of move actions in π_i. Thus, the probability of each agent experiencing at most d delays during execution is:

$$P(\pi, d) = \prod_{i=1}^{n} P(\pi_i, d). \quad (5)$$

Calculating $P_0(\pi, d)$. The Deterministic Verifier calculates the probability $P_0(\pi, d)$ of no conflicts occurring during execution provided that each agent experiences at most d delays during execution (Line 5), as follows: First, it determines all d-delay conflicts (Definition 3). Second, it partitions all agents that participate in at least one d-delay conflict into the largest number b of sets such that all d-delay conflicts occur only between agents in the same set. Third, for each resulting set of agents G_i ($1 \leq i \leq b$), it calculates the probability $P_0(\pi, G_i, d)$ of no conflicts occurring during execution between agents in G_i, in a way similar to the outline given the context of Equation 3: It considers only the part of the solution for the agents in G_i and enumerates all sets of delays for these agents that contain at most d delays for each agent in G_i and that the resulting part of the solution is robust to, i.e., that do not result in conflicts occurring during execution between agents in G_i. It then calculates $P_0(\pi, G_i, d)$ by adding the probabilities of experiencing each such set of delays. Fourth, it calculates $P_0(\pi, d)$ using

$$P_0(\pi, d) = \prod_{i=1}^{b} P_0(\pi, G_i, d) \quad (6)$$

Verification. Let X be the event that no conflicts occur during execution and Y be the event that each agent experiences at most d delays during execution. Then, the Deterministic Verifier calculates an upper bound UB_0 and a lower bound LB_0 on $P_0(\pi)$ as follows (Lines 6-7):

$$\begin{aligned} LB_0 =& P_0(\pi, d)P(\pi, d) = P(X|Y)P(Y) \\ =& P(X \wedge Y) \leq P(X) = P_0(\pi) \\ =& P(X) = P(X \wedge Y) + P(X \wedge \overline{Y}) \\ =& P(X|Y)P(Y) + P(X|\overline{Y})P(\overline{Y}) \\ \leq& P(X|Y)P(Y) + (1 - P(Y)) \\ =& P_0(\pi, d)P(\pi, d) + (1 - P(\pi, d)) = UB_0. \end{aligned}$$

The Deterministic Verifier calculates these bounds for increasing values of d, starting with $d = 0$ (Line 2). If $LB_0 \geq p$, then $P_0(\pi) \geq LB_0 \geq p$ (which means that the solution is p-robust) and the Deterministic Verifier returns TRUE (Line 8). If $UB_0 < p$, then $P_0(\pi) \leq UB_0 < p$ (which means that the solution is not p-robust) and the Deterministic Verifier returns FALSE (Line 9). Otherwise, the Deterministic Verifier increments d and iterates (Line 10).

5.2 Monte-Carlo Verifier

The Monte-Carlo Verifier (Lines 11-21 of Algorithm 2) determines whether a solution is p-robust based on executing random simulations of the given solution and performing a statistical hypothesis test. $1 - \alpha$ is the confidence level of the statistical test. In our implementation, we used $\alpha = 0.05$ as this is standard in statistical tests. Next, we describe how such a statistical test can be performed, how to set the initial number of simulations to execute, and how the verification process works.

Statistical test. Assume that no conflicts occur during P_0 percent of s_0 random simulations of executions. First, we use as null hypothesis that no conflicts occur with probability less than p and as alternate hypothesis that no conflicts occur with probability greater than p. According to the null

Algorithm 2: Verifiers

```
1  Verify(Solution π) // Deterministic Verifier
2      d ← 0
3      while TRUE do
4          Calculate P(π, d) // Equation 5
5          Calculate P_0(π, d) // Equation 6
6          LB_0 ← P(π, d) · P_0(π, d)
7          UB_0 = P(π, d) · P_0(π, d) + (1 − P(π, d))
8          if LB_0 ≥ p then return TRUE
9          if UB_0 < p then return FALSE
10         d ← d + 1
11 Verify(Solution π) // Monte-Carlo Verifier
12     Initialize s // Equation 8
13     Run s simulations
14     while TRUE do
15         Approximate P_0 based on the simulations
16         Calculate c_1 // Equation 7
17         Calculate c_2 // Equation 9
18         if P_0 ≥ c_1 then return TRUE
19         if P_0 < c_2 then return FALSE
20         s ← s + 1
21         Run one more simulation
```

hypothesis, the percentage of simulations with no conflicts is a normal distribution with mean p and variance $\frac{p(1-p)}{s_0}$ for large s_0 (say, larger than 30). Thus, we can use a Z-test to reject the null hypothesis (and conclude that the solution is likely p-robust) if

$$P_0 > p + z_{1-\alpha}\sqrt{\frac{p(1-p)}{s_0}}(= c_1). \tag{7}$$

This inequality can only be satisfied if $c_1 < 1$, which implies

$$s_0 > {z_{1-\alpha}}^2 \cdot \frac{p}{1-p}. \tag{8}$$

Thus, one needs to run at least $s_0 = \max(30, \lceil {z_{1-\alpha}}^2 \frac{p}{1-p} \rceil)$, simulations. Second, we use as null hypothesis that no conflicts occur with probability greater than p and as alternate hypothesis that no conflicts occur with probability less than p. A similar derivation shows that we can use a Z-test to reject the null hypothesis (and conclude that the solution is likely not p-robust) if

$$P_0 < p - z_{1-\alpha}\sqrt{\frac{p(1-p)}{s_0}}(= c_2). \tag{9}$$

Verification. The Monte-Carlo Verifier calculates s, runs s simulations, and calculates c_1 and c_2 (Lines 12-17). If $P_0 \geq c_1$ (similarly to Line 8 of the Deterministic Verifier), then the solution is likely p-robust and the Monte-Carlo Verifier returns TRUE (Line 18). If $P_0 < c_2$, then the solution is likely not p-robust and the Monte-Carlo Verifier returns FALSE (Line 19). Otherwise, the Monte-Carlo Verifier increments s, runs one more simulation, and iterates (Lines 20-21).

Since the Monte-Carlo Verifier runs simulations of executions, it can be used with any motion model for the agents, not only a motion model with a uniform delay probability for all agents. Thus, pR-CBS (and pR-GCBS, that we introduce next) can be used with any motion model if they use the Monte-Carlo Verifier.

6 Greedy p-Robust CBS

Obtaining optimal solutions for high p values is prohibitively slow in some cases. For such cases, we introduce *Greedy p-Robust CBS* (pR-GCBS), which is based on pR-CBS. pR-GCBS can use both verifiers described above. In our implementation, we used the Monte-Carlo Verifier as it performed better than the Deterministic Verifier in terms of runtime. pR-GCBS runs faster than pR-CBS but loses optimality. It contains the three modifications below.

Choosing a CT node for expansion. In order to obtain optimal solutions, pR-CBS performs a best-first search, always expanding the CT node with the lowest cost, since an optimal pR-MAPF solution has the lowest cost among all p-robust solutions. pR-GCBS calculates P_0 for each generated CT node and chooses to expand next the CT node with the highest P_0. Expanding nodes in increasing order of their proximity to the goal, based on a given heuristic function, showed great speed up for greedy algorithms. While this expansion order may lose optimality, it increases the chance to find a CT node with a high P_0 value and thus the chance to find a goal node quickly.

Resolving conflicts. In order to obtain optimal solutions, pR-CBS splits CT nodes according to Cases 1-3, since the potential conflict might exist in the optimal solution. pR-GCBS splits CT nodes only according to Cases 1-2 to ensure that the conflict is resolved. Splitting CT nodes according to Case 3 increases the size of the CT and, in many cases, not useful as the conflict is not resolved, but it is necessary for obtaining optimal solutions. With this conflict-resolution strategy, pR-GCBS may lose optimality (in case the optimal solution contains the potential conflict), but often increases the chance to find a goal node quickly.

Imposing constraints. In order to obtain optimal solutions, pR-CBS imposes the constraints $\langle a_i, x, t\rangle$ and $\langle a_j, x, t+\Delta(C)\rangle$ on agents a_i and a_j, respectively, to resolve the potential conflict $C = \langle a_i, a_j, x, t\rangle$ (Cases 1-2). pR-GCBS imposes the range constraints (Atzmon et al. 2018) $\langle a_i, x, [t, t+\Delta(C)]\rangle$ and $\langle a_j, x, [t, t+\Delta(C)]\rangle$ to resolve the potential conflict $\langle a_i, a_j, x, t\rangle$. These constraints prohibit both agents from occupying x during the time interval $[t, t+\Delta]$ and indeed resolve the potential conflict, but also resolve additional potential conflicts between the agents that might exist in the optimal solution. Thus, they increase the chance to find a goal node quickly.

7 Experimental Results

In this section, we experimentally evaluate pR-CBS with both verifiers and pR-GCBS with the Monte-Carlo Verifier on an Intel® Xeon E5-2660 v4 @ 2.00GHz processor with 16 GB of RAM.

	Cost			Expansions			Runtime (ms)		
	$p = 0.6$	$p = 0.7$	$p = 0.8$	$p = 0.6$	$p = 0.7$	$p = 0.8$	$p = 0.6$	$p = 0.7$	$p = 0.8$
CBS	35.5	35.5	35.5	16.77	16.77	16.77	7	7	7
pR-GCBS	35.7	36.0	39.7	4.16	6.71	7.22	154	218	374
pR-CBS (MC)	35.5	35.6	35.9	148.63	161.00	183.50	2,811	3,505	4,823
pR-CBS (DT)	35.5	35.6	35.9	148.63	161.00	184.00	9,445	27,545	62,325

Table 1: Average solution cost, number of high-level node expansions, and runtime for CBS, pR-GCBS, and pR-CBS on an 8x8 4-neighbor empty grid with 8 agents and delay probability $p_d = 0.1$.

	Cost			Runtime (ms)		
	$p = 0.6$	$p = 0.7$	$p = 0.8$	$p = 0.6$	$p = 0.7$	$p = 0.8$
CBS	659.8	659.8	659.8	81	81	81
pR-GCBS	660.0	661.7	664.7	732	1,201	3,473
pR-CBS (MC)	659.8	659.9	659.9	2,943	3,516	7,901

Table 2: Average solution cost and runtime for CBS, pR-GCBS, and pR-CBS on a 64x64 4-neighbor empty grid with 16 agents and delay probability $p_d = 0.1$.

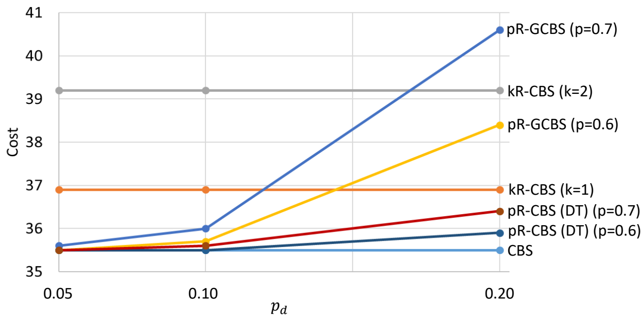

Figure 2: Average solution cost for CBS, kR-CBS, pR-CBS, and pR-GCBS on an 8x8 4-neighbor empty grid with 8 agents.

7.1 Runtime and Solution Cost

We first compare classic CBS, pR-CBS (DT) with the Deterministic Verifier, pR-CBS (MC) with the Monte-Carlo Verifier, and pR-GCBS with the Monte-Carlo Verifier, for different values of p $(0.6, 0.7, 0.8)$ and delay probability $p_d = 0.1$ on an 8x8 4-neighbor empty grid with 8 agents, where the start and goal locations are chosen uniformly at random from all cells. Table 1 shows the solution cost, the runtime, and the number of high-level node expansions for the different algorithms, averaged over 50 instances.

The cost of an optimal MAPF solution (for CBS) and the costs of optimal p-robust solutions (for pR-CBS) are relatively close (between 35.5 and 35.9). However, pR-CBS expands about one-order of magnitude more nodes and runs about two to three order-of-magnitude more slowly. To guarantee p-robustness, pR-CBS splits each CT node into 3 children and performs a verification on each node, which explains why it is slower than CBS (that does not provide any robustness guarantee). The Deterministic Verifier performs an exhaustive search, which explains why it is slower than the Monte-Carlo Verifier. On the other hand, it is simpler because it does not require the additional confidence parameter. pR-GCBS results in a good trade-off between the runtime and the resulting solution cost. Its solution cost is slightly higher than that of pR-CBS but it runs much faster with fewer node expansions than pR-CBS with either verifier.

We repeat the experiment of Table 1 on a 64x64 4-neighbor empty grid with 16 agents. Table 2 shows that the results remain similar. pR-CBS (DT) with the Deterministic Verifier does not solve any instance within a 5 minute runtime limit and is thus omitted from the table.

7.2 Delay Probability

We repeat the experiment of Table 1 for different values of the delay probability p_d $(0.05, 0.1, 0.2)$. Figure 2 shows the solution cost as a function of p_d. We also included kR-CBS for different values of k $(1, 2)$.

As expected, larger values of p_d and p result in higher solution costs for both pR-GCBS and pR-CBS, while they do not change the solution costs of CBS and kR-CBS as these algorithms do not use these values. kR-CBS results in a higher solution cost than the optimal pR-CBS and thus may be too restrictive.

7.3 Number of Conflicts

We repeat the experiment of Table 1 for different values of p $(0.6, 0.7, 0.8, 0.90)$, different values of k $(1, 2)$, and delay probability $p_d = 0.2$. Table 3 shows the percentage R of 50 simulations with no conflicts. pR-CBS does not solve all instances and is thus omitted from the table. CBS results in the lowest runtime and solution cost but also the lowest percentage of simulations with no conflicts. Larger values of p and k result in higher runtimes and solution costs but also higher percentages of simulations with no conflicts. As expected, pR-GCBS results in a percentage of simulations with no conflicts that is larger than its value of p since it provides a probabilistic robustness guarantee except for its greedy nature. kR-CBS with $k = 2$ and pR-GCBS with $p = 0.7$ result in about the same percentage of simulations with no conflicts. kR-CBS runs faster but does not provide any probabilistic robustness guarantee, and it is unclear how to choose the value of k for a desired probability of no conflicts.

We also performed a similar experiment on a larger map (brc202d) from the Dragon Age Origin video game, available in the `movingai` repository (Sturtevant 2012), for different numbers of agents $(10, 20, 30)$, different values of p $(0.6, 0.7, 0.8, 0.9)$, different values of k $(1, 3, 5, 7)$, and delay probability $p_d = 0.2$. Table 4 shows the results. Different from the smaller environment, the solution costs are now about the same in all cases (since this environment is

	Cost	Runtime (ms)	R
CBS($k = 0$)	35.5	7	0.42
kR-CBS ($k = 1$)	36.9	297	0.62
kR-CBS ($k = 2$)	39.2	973	0.84
pR-GCBS (MC) ($p = 0.6$)	38.4	1,734	0.77
pR-GCBS (MC) ($p = 0.7$)	40.6	5,550	0.84
pR-GCBS (MC) ($p = 0.8$)	41.1	10,683	0.89
pR-GCBS (MC) ($p = 0.9$)	45.5	25,402	0.94

Table 3: Average solution cost, runtime, and the percentage of conflict-free simulations R for CBS, kR-CBS on an 8x8 4-neighbor empty grid with 8 agents and delay probability $p_d = 0.2$.

	Runtime (ms)			R		
#Agents	10	20	30	10	20	30
CBS ($k = 0$)	268	888	2,377	0.77	0.54	0.37
kR-CBS ($k = 1$)	728	3,568	11,382	0.81	0.57	0.38
kR-CBS ($k = 3$)	1,635	9,771	28,361	0.85	0.64	0.44
kR-CBS ($k = 5$)	7,219	27,086	61,832	0.93	0.76	0.57
kR-CBS ($k = 7$)	12,014	41,913	104,449	0.96	0.85	0.73
pR-GCBS (MC) ($p = 0.6$)	8,833	36,051	70,815	0.96	0.90	0.86
pR-GCBS (MC) ($p = 0.7$)	10,063	40,114	79,081	0.98	0.90	0.88
pR-GCBS (MC) ($p = 0.8$)	10,141	53,091	87,206	0.99	0.94	0.92
pR-GCBS (MC) ($p = 0.9$)	10,192	79,299	112,994	0.99	0.98	0.96

Table 4: Average runtime and the percentage of conflict-free simulations R for CBS, kR-CBS, and pR-GCBS for different numbers of agents on the `brc202d` map.

not congested with agents) and thus not shown in the table. For example, the runtime is 8.833ms and the solution cost is $1,309.3$ for 10 agents and $p = 0.6$, while the runtime increases to $10,192$ms but the solution cost stays at $1,309.4$ for 10 agents and $p = 0.9$. Larger values of p and k again result in higher runtimes but also higher percentages of simulations with no conflicts. The percentage of simulations with no conflicts decreases for both pR-GCBS and kR-CBS as the number of agents increases. But it decreases only slightly for pR-GCBS and remains larger than its value of p since pR-GCBS provides a probabilistic robustness guarantee except for its greedy nature, while it decreases substantially for kR-CBS, which provides no such guarantee. For example, for 10 agents, CBS achieves $R = 0.77$, and kR-CBS with $k = 7$ achieves $R = 0.96$, while for 30 agents, CBS achieves $R = 0.37$, and kR-CBS with $k = 7$ achieves $R = 0.73$. The same trends are also observed in other domains, for other numbers of agents, and for other delay probabilities, which demonstrates the inability of CBS and kR-CBS to guarantee a specific value of p. Thus, one might prefer to use pR-CBS, if p is known or can be calculated, otherwise one could use kR-CBS as it is simpler.

8 Related Work

UM* (Wagner and Choset 2017) is a MAPF algorithm designed for delay probabilities. It returns a solution in which the probability of conflicts for each agent is below a given threshold. This is different from finding a p-robust solution, where the probability of conflicts for all agents together is below a given threshold. For example, assuming we have a threshold of $p = 0.02$ in UM* and a solution consisting of 100 agents, in which each agent has a probability of 0.01 to conflict. Although the total probability for no conflict at all is very low, in UM* this solution is acceptable. In contrast, a solution in which only two agents (out of 100) have a probability of 0.03 to conflict and all other agents have 0 probability to conflict is not an acceptable solution. Moreover, assuming a solution that is acceptable for both pR-MAPF and UM*, by removing a single-agent plan of one agent from the solution, the resulting solution will stay p-robust. However, it might no longer be an acceptable solution for UM* (see an example in Wagner and Choset (2017)).

AME (Ma, Kumar, and Koenig 2017) is a MAPF algorithm designed for delay probabilities. It aims to minimize the expected travel time and only guarantees no conflicts in the next time step. For subsequent time steps, it can utilize a set of *robust online execution policies* to avoid conflicts. However, these policies demand communication during execution, which might not be possible.

Lastly, one may consider formulating the pR-MAPF problem as a single, large *Markov Decision Process (MDP)*, in which a state consists of the locations of all agents and an action is the joint action of all agents. However, this would result in a large state space and a large branching factor, resulting in large runtime for more than a few agents.

9 Conclusions and Future Work

We proposed a new form of robustness for MAPF, called p-robustness, where p is the probability that no conflicts (i.e. collisions) occur during execution. Finding robust solutions for this form of robustness is possible when we have some knowledge of the delay probability of the agents. We explained why CBS-based algorithms have to perform an additional split to obtain optimal p-robust solutions. We proposed a complete and optimal CBS-based algorithm, pR-CBS, and a greedy CBS-based algorithm for finding p-robust solutions, both with two possible verifiers. We showed that p-robust solutions typically cost more than non-robust solutions, but can decrease the number of conflicts that occur during execution. We showed that kR-CBS, an existing CBS-based algorithm, typically runs faster than pR-CBS but cannot guarantee any specific required value for p.

There are many possible lines of future work, **(1)** adapting other MAPF solvers, such as ICTS (Sharon et al. 2013), to find p-robust solutions; **(2)** integrating p-robust solutions with execution policies, as suggested by Ma et al. (2017); and **(3)** designing additional verifiers that approximate, in more efficient ways, the probability of no conflicts occurring when executing a given solution.

10 Acknowledgements

This research was supported by the Israel Ministry of Science, ISF grants #844/17 to Ariel Felner and #210/17 to Roni Stern, NSF grants 1815660 to Nathan R. Sturtevant and 1724392, 1409987, 1817189, 1837779, and 1935712 to Sven Koenig, an Amazon Research Award to Sven Koenig, and BSF grants #2017692 and #2018684.

References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N. 2018. Robust multi-agent path finding. In *the International Symposium on Combinatorial Search (SoCS)*, 2–9.

Felner, A.; Stern, R.; Shimony, S. E.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N. R.; Wagner, G.; and Surynek, P. 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *the International Symposium on Combinatorial Search (SoCS)*, 29–37.

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. In *the International Conference on Automated Planning and Scheduling (ICAPS)*, 83–87.

Lam, E.; Bodic, P. L.; Harabor, D.; and Stuckey, P. J. 2019. Branch-and-cut-and-price for multi-agent pathfinding. In *the International Joint Conference on Artificial Intelligence (IJCAI)*, 1289–1296.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2019. Disjoint splitting for multi-agent path finding with conflict-based search. In *the International Conference on Automated Planning and Scheduling (ICAPS)*, 279–283.

Ma, H., and Koenig, S. 2017. AI buzzwords explained: Multi-agent path finding (MAPF). *AI Matters* 3(3):15–19.

Ma, H.; Kumar, T. K. S.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *the AAAI Conference on Artificial Intelligence (AAAI)*, 3605–3612.

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.

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

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Barták, R.; and Boyarski, E. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *the International Symposium on Combinatorial Search (SoCS)*, 151–159.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games* 4(2):144–148.

Surynek, P. 2010. An optimization variant of multi-robot path planning is intractable. In *the AAAI Conference on Artificial Intelligence (AAAI)*, 1261–1263.

Surynek, P. 2012. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *the Pacific Rim International Conference on Artificial Intelligence (PRICAI)*, 564–576.

Wagner, G., and Choset, H. 2015. Subdimensional expansion for multirobot path planning. *Artificial Intelligence* 219:1–24.

Wagner, G., and Choset, H. 2017. Path planning for multiple agents under uncertainty. In *the International Conference on Automated Planning and Scheduling (ICAPS)*, 577–585.

Yu, J., and LaValle, S. M. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *the AAAI Conference on Artificial Intelligence (AAAI)*, 1444–1449.