Multi-Agent Path Finding: Policies Instead of Plans

Jakub Mestek[a] and Roman Barták[b]
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic

Keywords: Multi-Agent Path Finding, Non-Deterministic Environment, Policies.

Abstract: The task of Multi-Agent Path Finding (MAPF) problem is to find collision-free plans for a set of agents moving from their starting locations to their destinations. In the classical variant of MAPF, a plan for an agent is a sequence of actions. In this paper, we suggest a novel approach to solving this problem in a non-deterministic environment – constructing a solution in the form of policies (one for each agent). The policy prescribes the agent which action it should take in a given situation described by a location and a timestep.

1 INTRODUCTION

Multi-Agent Pathfinding is a problem of constructing collision-free plans for a group of agents. The problem has many applications in various areas, including automated warehouses, aircraft taxiing on airfields, optimal crossroads controlling, and units navigation in video games.

Classical MAPF instance (Stern et al., 2019) consists of a graph $G = (V, E)$ and a set of agents A. For each agent $a \in A$, its source and destination (goal) vertices are given. We denoted them as $s(a)$ and $g(a)$, respectively. Time is discretized into unit-long timesteps. During every timestep, each agent can either move to an adjacent vertex or stay in its current vertex. A plan for an agent is a sequence of actions that leads the agent from its source to its destination. A solution of the MAPF instance is then a set of plans for individual agents. The main requirement for a solution is the nonexistence of conflicts (collisions) during execution. We use the common three types of conflicts which are vertex, edge, and swapping conflict (Stern et al., 2019). A vertex conflict is a situation when two agents are located in the same vertex at the same timestep. An edge or swapping conflict happens when two agents travel the same edge at the same timestep in the same or opposite direction, respectively.

In many real-world environments, agents are not able to follow the plans precisely – they might get delayed or even perform a wrong action (e.g., do not turn) (Barták et al., 2019), (Li et al., 2021a). Therefore, some kind of robustness is required – a (minor) deviation from the plan should not cause a collision with another agent.

This paper introduces a new method of achieving robustness via looking for a solution in the form of a policy that allows the agent to take different actions based on the current time. This approach enables a delayed agent both to avoid a location that is already occupied and to use a shorter path that has become free (due to arriving later than expected).

1.1 Related Work

Probably the simplest form of non-determinism in an environment is the possibility of agents being delayed. Such delay might then cause a collision, even though the original solution is conflict-free. Therefore, the problem of finding a *robust* solution has been already quite extensively studied.

Atzmon et al. (Atzmon et al., 2018) formalized the concept of delay as an insertion of unexpected wait action into the agent's plan and introduced the notion of k-robustness. A solution of the MAPF problem is k-robust if any set of at most k delays (for each agent) does not cause a collision. The approach is quite successful in avoiding collisions ((Atzmon et al., 2018), (Barták et al., 2019)). However, the main drawback is the prolongation of agents' plans – the condition basically requires each vertex to be free for at least k timesteps before another agent is allowed to enter it.

Nekvinda and Barták (Nekvinda and Barták, 2021) suggested achieving k-robustness by using al-

[a] https://orcid.org/0009-0003-8765-5092
[b] https://orcid.org/0000-0002-6717-8175

Mestek, J. and Barták, R.
Multi-Agent Path Finding: Policies Instead of Plans.
DOI: 10.5220/0012397200003636
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024) - Volume 1*, pages 95-104
ISBN: 978-989-758-680-4; ISSN: 2184-433X
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

ternative plans, instead of just time-separating the agents (i.e., adding wait actions). Plan with alternatives has a tree-like structure with one main branch – the original classical plan – and several other branches (alternatives) rooting from the main branch. During execution, each agent starts by following its main plan. In case of experiencing a delay (big enough to cause a collision), an agent might be detoured to one of the alternative branches. Mainly due to computability reasons, their method only provides alternatives for the main plan, not alternatives for alternatives. Furthermore, no collisions are guaranteed only among pairs of agents both following the main plan and among pairs of agents, where one follows its main plan and the second one its alternative plan. There is no guarantee for agents who both follow an alternative plan.

Another approach to combat unexpected delays is p-robustness (Atzmon et al., 2020) which aims to find a solution that will be executed successfully (i.e., without collisions) with a probability at least p.

Shahar et al. (Shahar et al., 2021) worked with a different model of delays, called MAPF with Time Uncertainty (MAPF-TU). In a MAPF-TU instance, an interval of possible duration (in timesteps) is assigned to each action (edge in the graph). The real duration of the action during execution is then a number from the interval. The task of MAPF-TU is to find a *safe* solution. A solution is *safe* if it guarantees no collisions during execution for any possible set of real action durations.

2 PRELIMINARIES

2.1 Conflict Based Search

Conflict Based Search (CBS), introduced by (Sharon et al., 2012), is one of the most widely-used algorithms for solving MAPF. CBS is a two-level algorithm. The upper level solves coordination of agents, while the lower level searches for plans (shortest paths) for each agent individually. This modularity makes CBS easily modifiable for different variants of MAPF, including ours. In this section, we describe the basic variant of CBS in detail.

Firstly, let us define some basic notions. A conflict of agents a_i and a_j who should be (given current solution) simultaneously in a timestep t located in a vertex v is denoted by (a_i, a_j, t, v). A constraint for an agent a_i is a tuple (a_i, v, t) denoting that the agent is prohibited from being in vertex v in timestep t. A solution (set of plans) is consistent with a given set of constraints C if all plans respect given constraints. A solution is valid if there are no conflicts among agents.

The idea of CBS is as follows. We maintain a set of constraints, initially empty. We find the cheapest consistent solution and check it for validity. If there is any conflict, the conflict is resolved by adding a constraint that prevents that conflict. We iterate this step until a valid solution is found. A conflict (a_i, a_j, t, v) can be resolved by adding one of the constraints (a_i, v, t) or (a_j, v, t). In order to obtain an optimal solution, we need to explore both options. Ultimately, this leads to the so-called Constraint Tree. Optimization is handled by the higher level of CBS that searches the Constraint Tree using best-first search which ensures finding an optimal solution.

The task of the low-level algorithm is to compute a consistent solution in each step. This can be done individually for each agent – basically, it is a shortest-path problem, that just has to avoid the prohibited states.

Over recent years, CBS has received several improvements that make it a state-of-the-art algorithm for classical MAPF, such as Conflict Avoidance Table (CAT) (Sharon et al., 2012), Priority Conflict (PC) and Conflict Bypassing (Li et al., 2019), or Corridor Reasoning (Li et al., 2021b).

2.2 Markov Decision Process

Markov Decision Process is a sequential decision process in a non-deterministic (stochastic) environment. Following Sutton and Barto (Sutton and Barto, 2018), MDP is a tuple (S, A, R, p). S denotes a finite set of states, including starting state s_0, A a finite set of actions, R is a set of possible rewards and a function $p\colon S \times R \times S \times A \to [0,1]$ describes dynamics of the system – the value $p(s', r|s, a)$ is the probability of getting to state s' and obtaining reward r when the agent executes action a in state s. Therefore, p is required to satisfy the following property:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1, \text{ for all } s \in S, a \in A.$$

In other words, for each choice of s and a, p specifies a probability distribution over $\{(s', r)\}$.

A solution to the MDP is a policy $\pi\colon S \to A$, i.e., a function that for each state s recommends an action $\pi(s)$ that the agent should perform in that state.

Quality of a policy π is measured using *return* – the cumulative sum of rewards obtained by the agent in the environment, possibly discounted by a discount factor γ (real number between 0 and 1). The purpose of the discount factor is to decrease the effect of distant actions on the actual value of return and to ensure that the return is a finite number even in the case of an infinite horizon.

Due to the stochasticity of the environment, the return might differ each time the agent runs in the environment. We define a *value function* $v_\pi(s)$ of a state s under a policy π as the expected return when starting in s and following π.

The optimal policy $\pi^\star$ is then such a policy that maximizes the value function $v_{\pi^\star}(s_0)$ of the starting state s_0. The value function of the optimal policy is called *utility*, denoted $U(s) = v_{\pi^\star}(s)$.

The relation between rewards and value functions is called the Bellman equation. For the utilities (of optimal policy), it holds

$$U(s) = \max_{a \in A(s)} \sum_{s'} \sum_{r} p(s', r | s, a) \left(r + \gamma U(s') \right), \qquad (1)$$

which we can rewrite using expectation over p to

$$U(s) = \max_{a \in A(s)} \mathsf{E}_{s', r \sim \mathsf{P}(s,a)} \left(r + \gamma U(s') \right). \qquad (2)$$

Using the Bellman equation, we can compute the optimal policy, for example, by the value iteration algorithm (Howard, 1960).

For details on MDP, see, e.g., Sutton and Barto (Sutton and Barto, 2018), or Russell and Norvig (Russell and Norvig, 2010).

3 OUR APPROACH

3.1 Non-Deterministic MAPF

Environment. We use the following model of a non-deterministic environment, inspired by the model used in MAPF-TU (Shahar et al., 2021).

The environment is represented in a standard way as graph $G = (V, E)$. Each edge (u, v), representing an action available in vertex u, is associated with a set of possible outcomes $\{(p_i, v_i, l_i)\}_i$. Outcome (p_i, v_i, l_i) means that with probability p_i, the agent ends up in vertex v_i, possibly different from v, and the real duration (length) of the action is l_i timesteps.

This representation enables us to express both the stochasticity of real durations of actions (modeling delays) and non-determinism in real action outcomes (modeling wrong turn). In the former case, the resulting vertex v_i is v for all outcomes of (u, v); while in the latter, resulting vertices of some outcomes are set to other vertices than v – e.g., other neighbors of u.

Furthermore, as the explicit distribution of possible action results is given, we can directly compute (and optimize) the expected length of the given plan.

Collision-Free Solution. A solution of nondeterministic MAPF is a set of policies $\{\pi_a\}$. A (single-agent) policy π_a is a function that for current timestep t and current location v of the agent a outputs an action $\pi_a(v, t)$. The solution is required to be collision-free. In order to formalize this property, we adopt a notion of *safe* solution (Shahar et al., 2021). A solution is called safe if there exists no such combination of real outcomes of executed actions that would lead to a vertex, edge, or swapping conflict.

Cost of Solution. *Real cost* of a single-agent policy π_a is the least timestep t when the agent arrives at its destination and will never leave it – stay-on-target scenario (Stern et al., 2019). Because of a non-deterministic environment, the real cost depends on the actual trajectory of the agent (actual outcomes of performed actions) and thus might differ if the policy is executed multiple times. Therefore we define also the *expected cost* of the policy as the expected value of real cost over all possible agent's trajectories.

For a complete solution of a given nondeterministic MAPF instance (set of policies), we then define *expected SoC* of the solution to be the sum of expected costs of individual policies.

3.2 DeltaPolicyCBS

We present a DeltaPolicyCBS, an algorithm that finds a safe optimal (in terms of expected SoC) solution of a given instance of nondeterministic MAPF. DeltaPolicyCBS is a modification of classical Conflict-Based Search that outputs single agent plans in the form of policy instead of action sequence.

The most important modification is the lower level of CBS – instead of the shortest path we need to output a minimum-expected-cost policy that respects given constraints. Another necessary modification is the addition of a procedure that computes *potential presence* of an agent under the given policy. The high level of DeltaPolicyCBS then looks for *potential conflicts* among agents and resolves them in a standard way by adding constraints. A potential conflict is an element of a non-empty intersection of the potential presences of two agents.

3.2.1 Computing Minimum Cost Policy

In case of no constraints, the problem of computing the minimum expected cost policy for a given agent a can be easily formulated as a Markov Decision Process with an infinite horizon.

We set the set of states to be V (vertices of the graph), set of available actions in vertex v to $A_v = \{(v, w) \mid w \in V, (v, w) \in E\}$ (all neighbors, including a loop (v, v) that represents wait action) and the probability $P(v'_i, l_i | v, a)$ of ending up in vertex v'_i and obtaining reward l_i when action a is performed in vertex

v to be equal to p_i if $a \in A_v$ and (p_i, v'_i, l_i) is a possible outcome of the action. Otherwise, $P(v'_i, l_i | v, a) = 0$.

We consider wait actions to be always deterministic, therefore for each vertex v the wait action (v, v) has exactly one possible outcome $(1, v, 1)$ – with probability 1, the agent will stay in vertex v and the duration of the action is 1 timestep. So, $P(v, 1 | v, (v, v)) = 1$.

The only exception is the wait action in agent's destination vertex $g = g(a)$. We set the reward for such action to be 0, i.e., $P(g, 0 | g, (g, g)) = 1$ – when the agent executes wait action in g, it stays in g but receives reward 0 (instead of 1). Effectively, this means that waiting at the destination does not increase the total return and makes the destination an absorbing state of the MDP – note, that we minimize the return (utility), see next paragraph.

For given destination vertex g $(= g(a))$ and policy $\pi\colon V \to A$, the expected travel time $d_\pi(v)$ from a vertex v to g is equal to $\mathsf{E}_{v',l \sim P_{\pi(v)}}\, l + d_\pi(v')$, i.e., an expectation over possible action outcomes of the sum of the action duration and expected travel time from next vertex. Therefore, $d_\pi(v)$ is equal to the expected utility of state v in the above-defined MDP if one is minimizing utility. Note that minimization of utility in MDP is equivalent to maximization of utility in MDP with negative rewards.

The optimal policy for an agent can therefore be computed as the optimal policy of above mentioned MDP. We can use standard algorithms, such as value iteration.

In the case of a non-empty set of constraints, the computation of the policy is a bit complicated. We have to ensure that an agent a following its policy π_a will never be in a vertex or edge x in timestep t if (a, t, x) is one of the constraints.

We can still formulate the problem as MDP, as follows: The set of states is $V \times T$, initial state is $(s(a), 0)$. We require the optimal policy to bring the agent to a state $(g(a), t)$ (for any t) as soon as possible (and then keep the agent in the vertex $g(a)$). Therefore, we set the rewards in the same way (to be the real duration of the action), except for waiting in the agent's destination vertex, where the reward (penalization) will be 0 only for $t \geq T_{goal}$ (until that it will be 1 as for other wait actions). T_{goal} is a timestep since which the agent can stay at its destination forever, i.e., it is the timestep of the last constraint denying the destination vertex.

Formally,

- set of states is $V \times T$,
- set of available actions in given state is $A_{(v,t)} \subseteq A_v$ – given the constraints, an action is not available at given timestep t if any of its possible outcomes would lead to banned state (v', t'),
- probability of ending in state $(v'_i, t + l_i)$ and obtaining reward l_i (when executing an action a in state (v,t)) is $P((v'_i, t + l_i), l_i | (v, t), a) = p_i$, if $a \in A_{(v,t)}$ and (p_i, v'_i, l_i) is a possible outcome of a; 0 otherwise.

Still, we need to work with infinite horizon MDP. Hence, the presented formulation would lead to an infinite set of timesteps T and to an infinite set of MDP states. However, we need the timesteps in states just to be able to express variable (un)availability of actions. An action a from vertex v is unavailable only if using it would lead to a violation of any constraint. Thus, $A_{(v,t)}$ depends on t only for t less than or equal to the timestep of the last constraint, let us denote it t_{max}. Later (for $t > t_{max}$), all actions are available, i.e., $A_{(v,t)} = A_v$.

Therefore, for any $t > t_{max}$, the expected travel time $d_\pi(v,t)$ is equal to the expected travel time from the version of MDP without constraints. Thus, the optimal policy is the same as the optimal policy for MDP without constraints (for $t > t_{max}$).

Provided that the action durations are positive, an optimal policy can be computed as follows: For $t > t_{max}$ we use the precomputed policy from the version without constraints. For $t \leq t_{max}$ we compute the policy using the Bellman equation. We use dynamic programming and compute the policy "backwards" – successively for t_{max}, $t_{max} - 1$, …, 0). Due to the positivity of action duration, all required values $d_{\pi^\star}(v', t + d)$ at each step have already been computed.

The method is summarized in algorithm 1. To prove correctness, let us note that the algorithm is in fact value iteration where the individual states are iterated in topological order. Thus, one iteration is sufficient to compute the exact utilities of states (and optimal policy as well).

3.2.2 Incremental Approach

After adding a new constraint $c = (a_c, t_c, v_c)$ to C, computation of a new policy π' for agent a_c satisfying all constraints in C is required. However, it is not necessary to compute the policy from scratch, it is sufficient just to modify the existing policy π. Specifically, it is sufficient to recompute $\pi'(v,t)$ for such states (v,t) in which the optimal action might change.

Let us define a set of successor states $N(v,t)$ as a set of states (v', t') such that (v', t') is reachable from (v,t) using an action. Therefore, according to the Bellman equation, the value $d_\pi(v', t')$ influences the choice of optimal action in (v,t).

The optimal action is chosen as $\arg\min_{a \in A_{(v,t)}} \mathsf{E}_{v',l \sim P_{\pi(v,t)}}\, l + d_\pi(v', t + l)$. There

```
Data: Graph $G = (V, E)$, goal vertex, set of
      constraints $C$
Result: $\pi$, $d_\pi$ – optimal policy and
        corresponding expected travel times
        to goal
$\phi, d_\phi \leftarrow$ optimal policy and corresponding
  expected travel times to goal in case
  constraints are ignored;
$t_{max} \leftarrow \max\{constraint.t \mid constraint \in C\}$;
for $t > t_{max}$ do
    for $v \in V$ do
        $\pi(v,t), d_\pi(v,t) \leftarrow \phi(v), d_\phi(v)$;
    end
end
for $t$ in max_dynamic_t, $\dots, 0$ do
    for vertex $v \in V$ do
        $d_\pi(v,t) \leftarrow$
          $\min_{a \in A_{(v,t)}} \mathsf{E}_{v',l \sim P_{\pi(v,t)}} l + d_\pi(v', t+l)$;
        $\pi(v,t) \leftarrow$ corresponding action;
    end
end
return $\pi$, $d_\pi$
```

Algorithm 1: Computation of optimal policy.

are two reasons why the value might change. Firstly, if the set of allowed actions $A_{(v,t)}$ changes. That happens if $(v_c, t_c) \in N(v,t)$, i.e. one or more actions lead to the newly prohibited state. Secondly, if the value of $d_\pi(v', t+l)$ changes. That means, there exists a $(v',t') \in N(v,t)$ such that $d'_\pi(v',t') \neq d_\pi(v',t')$.

Due to this fact, we may compute the new policy π' more efficiently using the algorithm 2.

Recall, for each successor $(v',t') \in N(v,t)$ holds $t' > t$ (assuming a positive duration of actions). Hence, the timestep t' of each state in N^{-1} is less than the timestep t of the currently processed state (v,t). Therefore, each state (v,t) is processed at most once.

All states in which the new policy π' or distance $d_{\pi'}$ might differ from π or d_π, respectively, are recomputed. Hence, the algorithm 2 returns the same result as the original algorithm 1.

3.2.3 Conflict Detection

To be able to implement the DeltaPolicyCBS algorithm, we need a method to detect possible conflicts between agents (following their policies). We propose using a concept of *potential presence* (Shahar et al., 2021). The idea is to "simulate" the agent following its policy and mark the visited timestep-vertex and timestep-edge pairs.

The method is shown in algorithm 3. We show directly a version extended by also marking the probability of presence. The pseudocode uses the following

```
Data: Graph $G = (V, E)$, destination goal, set
      of constraints $C$, new constraint $c$,
      previous optimal policy $\pi$ and
      corresponding expected travel times $d_\pi$
      ($\pi$ respects $C \setminus c$)
Result: $\pi'$, $d_{\pi'}$ – optimal policy and
        corresponding expected travel times
        to goal respecting all constraints
        (including $c$)
$\pi', d_{\pi'} \leftarrow \pi, d_\pi$;
$q \leftarrow$ empty queue // states to recompute;
$N^{-1} = \{(v',t') \mid (c.v, c.t) \in N(v',t')\}$;
for $(v',t') \in N^{-1}$ do
    if $(v',t') \notin q$ then
        $q.Enqueue(v',t')$;
    end
end
while $q$ not empty do
    $v,t \leftarrow q.Dequeue()$;
    $d_{\pi'}(v,t) \leftarrow$
      $\min_{a \in A_{(v,t)}} \mathsf{E}_{v',l \sim P_{\pi(v,t)}} l + d_\pi(v', t+l)$;
    $\pi'(v,t) \leftarrow$ corresponding action;
    if $d_{\pi'}(v,t) \neq d_{\pi'}(v,t)$ then
        $N^{-1} = \{(v',t') \mid (v,t) \in N(v',t')\}$;
        for $(v',t') \in N^{-1}$ do
            if $(v',t') \notin q$ then
                $q.Enqueue(v',t')$;
            end
        end
    end
end
return $\pi'$, $d'_\pi$
```

Algorithm 2: More effective computation of optimal policy using incremental approach.

notation:

- *PP* potential presence in vertices, *PPE* potential presence in edges,
- $PP[t] = \{(v_i, p_i)\}$ is a list of vertices where the agent might be situated in timestep t, including probability,
- method $PP.add((v,t,p))$ marks new option how the agent might arrive to state (v,t) with probability p – if v is not in $PP[t]$, it is added with probability p; otherwise, probability is increased by p (probability of independent ways how to arrive to (v,t) is summed),
- $PPE[t,(v,v')]$ probability of agent being situated on edge (v,v') in timestep t
- method $PPE.add((v,v'),(t,t'),p)$ similarily marks new presence of the agent on the edge (v,v') between timesteps t and t' (during travel

over the edge), with probability p – again, p is added to $PPE[\bar{t},(v,v')]$ for all $\bar{t} = t,\dots t'$.

```
Data: Graph $G = (V,E)$, agent's starting
      vertex $start$, policy $\pi$
Result: set of states (and probability
        distribution), where the agent might
        locate during execution of $\pi$
$PP.add((start,0,1))$ // in timestep 0, agent is
  in its source vertex (with probability 1);
$t = 0$;
while $PP$ not stable do
    for $(v,p) \in PP[t]$ do
        for possible outcome $(p',v',l')$ of the
          action $\pi(v,t)$ do
            $PP.add((v',t+l',p \cdot p'))$;
            $PPE.add((v,v'),(t,t+l'),p \cdot p')$;
        end
    end
    $t \leftarrow t+1$;
end
return $PP$, $PPE$
```
Algorithm 3: PotentialPresence.

The simulation runs until PP stabilizes, i.e., until $PP[t'] = PP[t'+1]$ is quarantied for all $t' > t$. Sufficient condition is that the agent is with probability 1 in its destination ($PP[t] = (goal,t,1)$) and will not leave it ($t > t_{max}$).

Once the potential presence of all agents is computed, potential conflicts can be found easily as an intersection of the potential presences of two agents (see algorithm 4). Similarly to other algorithms, we show a version extended by outputting the probability of each potential conflict (in addition to the conflict itself). The probability of the conflict is equal to $p_i \cdot p_j$ because both agents need to be situated in the given state (v,t) simultaneously.

3.2.4 Complete Algorithm

The complete high-level algorithm of DeltaPolicy-CBS is shown in algorithm 5.

4 EXPERIMENTS

We conduct several experiments to compare the proposed policy approach to MAPF to classical k-robustness and to explore the influence of pruning (ignoring not much probable conflicts) on runtime and quality of solution.

By the quality of a solution, we refer to

- expected cost (expected SOC),

```
Data: Potential presence $PP_a, PPE_a$ for each
      agent $a \in Ag$
Result: List of potential conflicts (optionally
        including their probability)
forall $a_i \in Ag$ do
    forall $a_j \in Ag; j < i$ do
        forall $t$ do
            forall $v$: $(v,p_i) \in PP_{a_i}[t]$ and
              $(v,p_j) \in PP_{a_j}[t]$ do
                Add $(a_i,a_j,v,t,p_i \cdot p_j)$ (or
                  only $(a_i,a_j,v,t)$) to $C$;
            end
            forall
            $(v,v')$: $PPE_{a_i}[t,(v,v')] = p_i >$
              $0 \wedge PPE_{a_i}[t,(v,v')] = p_j > 0$ do
                Add $(a_i,a_j,(v,v'),t,p_i \cdot p_j)$
                  (or only $(a_i,a_j,v,t)$) to $C$;
            end
        end
    end
end
return $C$
```
Algorithm 4: FindConflicts.

```
Data: MAPF instance
$Root.constraints \leftarrow \emptyset$;
$Root.solution \leftarrow$ find individual policies by
  the low level;
$Root.cost \leftarrow Cost(Root.solution)$;
insert $Root$ to $OPEN$;
while $OPEN$ not empty do
    $N \leftarrow$ best node from $OPEN$;
    $pp \leftarrow PotentialPresence(N.solution)$;
    $conflicts \leftarrow FindConflicts(pp)$;
    if no conflict then
        return $N.solution$
    end
    $C \leftarrow$ a conflict $(a_i,a_j,x,t)$ from $conflicts$
     forall agent $a$ in $C$ do
        $NN \leftarrow$ new node;
        $NN.constraints \leftarrow$
          $N.constraints + (a,v,t)$;
        $NN.solution \leftarrow N.solution$;
        Update $NN.solution$ by low level –
          update $a$'s policy;
        $NN.cost \leftarrow Cost(NN.solution)$;
        if $NN.cost < \infty$ then
            Insert $NN$ to $OPEN$;
        end
    end
end
```
Algorithm 5: PolicyCBS – high level – pseudocode.

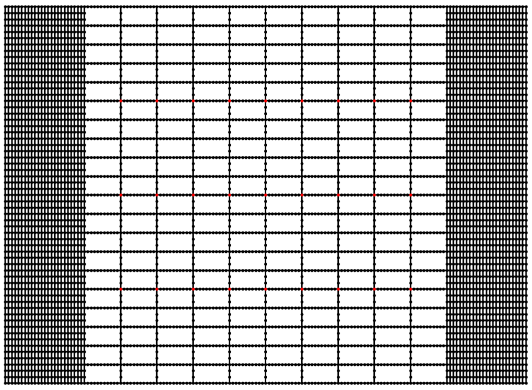
Figure 1: Warehouse map.

- real cost (evaluated by simulation),
- success rate of execution – whether any collision happened during simulation.

Experiments were run on a computer with AMD Ryzen 5 3600 CPU (frequency 3.9 GHz) and 32 GB RAM. We used our implementation of DeltaPolicyCBS in Python 3.11.0. To obtain *k*-robust solutions, we used Improved k-Robust CBS (Atzmon et al., 2018), implemented by Nekvinda and Barták (Nekvinda and Barták, 2021).

In our experiments, we abbreviate DeltaPolicyCBS as DPCBS. DPCBS-0.001 then denotes the pruned variant of DeltaPolicyCBS that ignores conflicts whose probability is less than 0.001. Of course, the pruned variant guarantees neither optimality nor safeness of the solution.

4.1 Instances

MAPF Benchmark (Stern et al., 2019) contains several different kinds of maps commonly used for testing MAPF algorithms. For our experiments, we use the empty grid map 8×8 and a warehouse map (namely, warehouse-10-20-10-2-1). The warehouse map is shown in Figure 1. Instances are created by taking the first *n* agents from each of 25 random scenarios for a given map. Solutions of successfully solved instances are then simulated on 50 presampled sets of real action results. During each simulation, real SOC cost and number of collisions are observed.

We used two modes of incorporating nondeterminism. In the first set of instances, we set two possible outcomes to every non-deterministic edge – length 1 with probability $1-p$ and length 2 with probability p. The parameter p can be seen as the probability of delay.

In the second set of instances, edges (actions) from selected vertices only are set as nondeterministic, we refer to those vertices as *locations with higher delay probability*. In particular, in the map Grid, all vertices in the 3rd and 5th rows were selected. In the map Warehouse, inner crossroads in the 5th, 10th, and 15th rows were selected. Other actions are deterministic – have one and only possible outcome with travel time 1. We refer to this set of instances as *mixed* – as there are both deterministic and nondeterministic actions.

4.2 Results

For each map, number of agents, and value of p (delay probability) we present, for each algorithm, the ratio of solved instances (out of the 25), the success rate of executions (ratio of simulations that ended without any collision) and the mean of measured real costs (in terms of SOC). In order not to penalize an algorithm that solved more instances by including real costs of those, possibly more difficult, instances, the execution success rate and real costs are computed using only instances solved by all algorithms.

Table 1 shows the results on the map Grid with 10 agents. DPCBS achieves a 100% success rate, similarly to 1- and 2-robust classic plans. However, the real cost of DPCBS solutions is on average lower, especially compared to 2-robust plans.

Tables 2 and 3 show the results on the map Grid–mixed with 10 and 13 agents, respectively. Similarly as in the previous case, DPCBS is able to reach a success rate of 100% while achieving low real costs – similar to the 0-robust solution. *k*-robust solutions are either not successful on all instances (for $k \leq 1$) or have significantly higher real costs ($k = 2$). The major drawback of DPCBS is not being able to solve all instances in the given time limit, as opposed to *k*-robust solutions (at least for smaller *k*).

The same observations hold also for the bigger map, Warehouse–mixed, as shown in the table 4.

Apart from the optimal DPCBS, tables contain results of its pruned variant DPCBS-0.001. Recall, that DPCBS-0.001 ignores conflicts whose probability is less than 0.001 and therefore is not optimal and does not guarantee the safety of the solution. Notwithstanding, no collisions occurred during simulations and the real costs of the solutions were almost identical to those of DPCBS (on instances solved by both algorithms. On the other hand, due to ignoring some conflicts, DPCBS-0.001 was able to solve more instances than DPCBS, especially when the delay probability p was rather small.

Table 1: Quality of solutions, instances on the map Grid, 10 agents.

Algorithm	**Instances solved**			**Execution success rate**			**Real cost**		
	$p = 0.1$	0.2	0.5	0.1	0.2	0.5	0.1	0.2	0.5
DPCBS	0.52	0.56	0.68	1.00	1.00	1.00	50.22	54.82	69.79
DPCBS-0.001	0.72	0.64	0.68	1.00	1.00	1.00	50.21	54.81	69.84
0r-plan	1.00	1.00	1.00	0.72	0.61	0.64	49.31	53.84	68.91
1r-plan	1.00	1.00	1.00	0.98	0.97	0.97	50.41	55.03	70.40
2r-plan	1.00	1.00	1.00	1.00	1.00	1.00	52.78	57.49	73.14

Table 2: Quality of solutions, instances on the map Grid–mixed, 10 agents.

Algorithm	**Instances solved**			**Execution success rate**			**Real cost**		
	$p = 0.1$	0.2	0.5	0.1	0.2	0.5	0.1	0.2	0.5
DPCBS	0.88	0.88	1.0	1.00	1.00	1.00	49.67	51.27	57.05
DPCBS-0.001	1.00	1.00	1.0	1.00	1.00	1.00	49.65	51.27	57.05
0r-plan	1.00	1.00	1.0	0.78	0.66	0.53	49.59	51.70	59.14
1r-plan	1.00	1.00	1.0	0.99	0.97	0.92	51.06	53.14	60.66
2r-plan	1.00	1.00	1.0	1.00	1.00	0.99	53.83	55.95	63.93

Table 3: Quality of solutions, instances on the map Grid–mixed, 13 agents.

Algorithm	**Instances solved**			**Execution success rate**			**Real cost**		
	$p = 0.1$	0.2	0.5	0.1	0.2	0.5	0.1	0.2	0.5
DPCBS	0.68	0.68	0.8	1.00	1.00	1.00	62.29	64.38	70.81
DPCBS-0.001	0.76	0.72	0.8	1.00	1.00	1.00	62.29	64.38	70.81
0r-plan	1.00	1.00	1.0	0.69	0.49	0.35	62.03	64.80	72.97
1r-plan	1.00	1.00	1.0	0.99	0.95	0.85	65.03	67.69	75.86
2r-plan	0.60	0.60	0.6	1.00	1.00	0.97	70.57	73.37	81.69

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 4: Quality of solutions, instances on the map Warehouse–mixed, 20 agents.

Algorithm	**Instances solved**			**Execution success rate**			**Real cost**		
	$p = 0.1$	0.2	0.5	0.1	0.2	0.5	0.1	0.2	0.5
DPCBS	0.72	0.76	0.80	1.00	1.00	1.00	1537.67	1539.99	1555.67
DPCBS-0.001	0.80	0.84	0.80	1.00	1.00	1.00	1537.67	1539.99	1555.68
0r-plan	0.88	0.88	0.88	0.84	0.76	0.70	1540.51	1545.94	1571.55
1r-plan	0.80	0.80	0.80	1.00	0.99	0.92	1541.16	1546.77	1572.55
2r-plan	0.72	0.72	0.72	1.00	0.99	0.93	1541.17	1546.66	1572.17

Table 5: Quality of solutions, instances on the map Grid–mixed with non-deterministic resulting vertex, 7 agents.

Algorithm	**Success rate**		**Real costs (average)**	
	$p = 0.1$	$p = 0.5$	$p = 0.1$	$p = 0.5$
DPCBS	0.00	0.21	—	43.57
DPCBS-0.001	0.36	0.36	36.58	40.72

Table 6: Quality of solutions, instances on the map Warehouse–mixed with non-deterministic resulting vertex, 10 agents.

Algorithm	**Success rate**		**Real costs (average)**	
	$p = 0.1$	$p = 0.5$	$p = 0.1$	$p = 0.5$
DPCBS	0.52	0.72	671.72	661.09
DPCBS-0.001	0.96	0.96	660.42	667.47

4.3 Non-Deterministic Resulting Vertex

Additionally, we conduct another set of experiments in environments with non-deterministic results of actions. Recall, that in our model of environment, a set of possible outcomes $\{(p_i, v_i, l_i)\}_i$ is assigned to each action. In this set of experiments, outcomes might have different vertex v_i in (which agent ends up).

This set of experiments is conducted on the maps Grid–mixed and Warehouse–mixed. Possible outcomes of a move action (u, v) from vertex u are set as follows. Duration l_i is always set to 1. If u is not marked (as a location with higher delay probability), the resulting vertex v_i is equal to v. If u is marked, the action has 3 possible resulting vertices (3 possible outcomes): v with probability $1 - 2p$, v_{-1} with probability p, and v_{+1} with probability p. Vertices v_{+1} and v_{-1} are neighbors of u located clockwise, or counterclockwise, from v.

Possible outcomes correspond to the possibility that an agent traveling through a marked vertex turns 90° left or right from the desired direction. Wait actions are always deterministic.

Results on the map Grid–mixed with 7 agents are shown in Table 5. Table 6 shows results on Warehouse–mixed with 10 agents. Contrary to the previous section, we present a success rate which is a ratio of simulations in which both the instance was solved by the algorithm and the simulation ended without any collisions. Both success rate and real costs are computed from all instances and simulation, for each algorithm independently (i.e., it is not limited to instances solved by both algorithms only). The reason is the ratio of solved instances by one of the algorithms.

Results show that with $p = 0.5$, algorithms are more successful but the cost of solutions is higher. This is expected because, with $p = 0.5$, non-deterministic actions have in fact two possible results only. On the other hand, they lead to opposite vertices.

DPCBS (without pruning) is not very successful, especially on the smaller map with more agent interactions and a higher density of non-deterministic actions. We assume it might be caused simply by the non-existence of a safe solution. The existence of a safe solution in non-deterministic MAPF remains an open problem.

Nevertheless, results illustrate that by using our approach (policies instead of plans) it is possible to solve at least some instances of MAPF in an environment with non-deterministic resulting vertices. Let us recall that classical plans (sequences of actions) cannot be used in such environments.

5 CONCLUSIONS

In this paper, we introduced a novel idea of representing MAPF solution by policies, instead of sequences of actions or sequences of visited vertices. Policies allow the agent to take different actions based on actual timesteps of arrival into a vertex. Therefore, agents are able to react to experienced delay during execution while we are still in an area of *offline* MAPF – the solution (policies) is computed completely before the execution.

The use of policies is beneficial only in non-deterministic environments where agents might experience some delays or other unpredicted events. We formalized such a non-deterministic environment in a way similar to MDP – a set of possible outcomes is assigned to each action (*move* to a neighbor vertex). The possible outcome consists of the duration of the action (which allows us to model a delay) and the resulting vertex (to model for example a wrong turn).

The main part of this paper presents an algorithm, called DeltaPolicyCBS (DPCBS), that optimally (in terms of expected cost) solves MAPF in the above-mentioned version of a non-deterministic environment and returns the solution in the form of policies. The algorithm is a modification of a well-known Conflict Based Search in which individual shortest-path problems are formulated as Markov Decision Processes. A solution of the MAPF problem then consists of (optimal) policies for those MDPs.

In the last section, DPCBS was experimentally evaluated on several types of MAPF instances. Solutions were executed in a simulation and compared to classical k-robust plans. We showed that usage of policy-based solutions found by DPCBS leads to no collisions during execution while keeping low real costs of the solutions.

The intent of our work was to explore a novel approach to solving MAPF in a non-deterministic environment. We have shown that using policies is beneficial not only in environments in which the results of actions are nondeterministic – as those environments cannot be solved using standard plans – but also in environments in which the only possible non-determinism is delays – using policies reduces real costs while ensuring no collisions. Our work opened several directions for future research. For the approach to be usable in practice, a more efficient method (than our current version of DPCBS) for finding a MAPF solution in the form of policies is needed, possibly accompanied by some clever efficient data structure for storing policies in a compact way. The incapability of DPCBS to solve even some rather small instances opens the question of the actual existence of a safe policy solution of a given MAPF instance, especially in highly nondeterministic environments, such as our *not-mixed* Grid map with some delay probability on each edge.

ACKNOWLEDGEMENTS

Research is supported by project No 23-05104S of Czech Science Foundation.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

REFERENCES

Atzmon, D., Stern, R., Felner, A., Sturtevant, N. R., and Koenig, S. (2020). Probabilistic Robust Multi-Agent Path Finding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1):29–37.

Atzmon, D., Stern, R., Felner, A., Wagner, G., Barták, R., and Zhou, N.-F. (2018). Robust Multi-Agent Path Finding. *Symposium on Combinatorial Search (SoCS)*, pages 2–9.

Barták, R., Švancara, J., Škopková, V., Nohejl, D., and Krasičenko, I. (2019). Multi-agent path finding on real robots. *AI Communications*, 32(3):175–189.

Howard, R. (1960). *Dynamic programming and Markov processes*. Technology Press of Massachusetts Institute of Technology.

Li, J., Chen, Z., Zheng, Y., Chan, S.-H., Harabor, D., Stuckey, P. J., Ma, H., and Koenig, S. (2021a). Scalable rail planning and replanning: Winning the 2020 flatland challenge. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1):477–485.

Li, J., Felner, A., Boyarski, E., Ma, H., and Koenig, S. (2019). Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 442–449. International Joint Conferences on Artificial Intelligence Organization.

Li, J., Harabor, D., Stuckey, P. J., Ma, H., Gange, G., and Koenig, S. (2021b). Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 301:103574.

Nekvinda, M. and Barták, R. (2021). Contingent Planning for Robust Multi-Agent Path Finding. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 487–492.

Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Shahar, T., Shekhar, S., Atzmon, D., Saffidine, A., Juba, B., and Stern, R. (2021). Safe Multi-Agent Pathfinding with Time Uncertainty. *J. Artif. Int. Res.*, 70:923–954.

Sharon, G., Stern, R., Felner, A., and Sturtevant, N. (2012). Conflict-Based Search For Optimal Multi-Agent Path Finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 26(1):563–569.

Stern, R., Sturtevant, N., Felner, A., Koenig, S., Ma, H., Walker, 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 (SoCS)*, pages 151–158.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, 2 edition.