On Modelling Multi-Agent Path Finding as a Classical Planning Problem

Jindřich Vodrážka
Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic
vodrazka@ktiml.mff.cuni.cz

Roman Barták
Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic
bartak@ktiml.mff.cuni.cz

Jiří Švancara
Charles University
Faculty of Mathematics and Physics
Prague, Czech Republic
Jiri.Svancara@mff.cuni.cz

***Abstract*—Multi-Agent Path Finding (MAPF) deals with finding collision-free paths for a set of agents. It is a special case of a planning problem with only two actions – move and wait – and with one major constraint of no collision between the agents. The paper addresses the question of how to model MAPF as a classical sequential planning problem. Several models in the PDDL language are proposed and empirically compared.**

***Index Terms*—Multi-Agent Path Finding, Classical Planning, Modeling**

I. Introduction

With the increasing development of autonomous agent technologies, the need to coordinate the movement of many agents is even more important. The application areas range from video games to automated warehousing, and other areas, where automation is expected soon such as automated traffic control, taxiing, and road junction control.

There exists an abstract framework of multi-agent path finding (MAPF) dealing with finding collision-free paths for a set of agents. Most ad-hoc solvers that have been developed to solve MAPF problems are based either on search techniques or problem reformulation (for example to the problem of Boolean satisfiability [1]). Nevertheless, MAPF is also a special kind of a planning problem with two types of actions – move to a neighboring node or stay at the current node.

There are several reasons to undergo an endeavour of encoding MAPF as a classical planning problem. It will provide another problem compilation method to solve MAPF problems. Also, the compilation of MAPF to classical planning will provide some guidelines on how to encode specific problems as planning problems. Note that despite advances in automated planning and the importance of problem formulation for the efficiency of planning [2] there are still no guidelines for encoding planning problems in a way efficient for automated planners. We believe that the modeling ideas presented in the paper can be exploited in other coordination problems modelled as sequential planning problems.

We will present how an inherently parallel problem of MAPF can be encoded as a sequential planning problem. We will also show how the sequential plans obtained from automated planners can be parallelised to obtain MAPF plans with agents that are moving at the same time. Several models in the PDDL modeling language [3] will be presented and empirically compared. These models and the techniques behind them are the major novel contribution of the paper.

II. Background on classical planning

We work with classical STRIPS planning [4] that deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal conditions. World states are modelled as sets of propositions that are true in those states, and actions are modelled to change the validity of certain propositions.

Let P be a set of all propositions modelling properties of world states. Then a state $S \subseteq P$ is a set of propositions that are true in that state (other propositions are false).

Each action a is described by three sets of propositions (B_a^+, A_a^+, A_a^-), where $B_a^+, A_a^+, A_a^- \subseteq P, A_a^+ \cap A_a^- = \emptyset$. Set B_a^+ describes positive preconditions of action a, i.e., propositions that must be true right before the action a. Some modeling approaches allow also negative preconditions, but these preconditions can be compiled away. Action a is applicable to state S iff $B_a^+ \subseteq S$. Sets A_a^+ and A_a^- describe positive and negative effects of action a, i.e., propositions that will become true and false in the state right after executing the action a. If an action a is applicable to state S then the state right after the action a is $\gamma(S, a) = (S \setminus A_a^-) \cup A_a^+$, while $\gamma(S, a)$ is undefined if an action a is not applicable to S.

The classical planning problem, also called a STRIPS problem, consists of a set of actions A, a set of propositions S_0 called an initial state, and a set of goal propositions G^+ describing the propositions required to be true in the goal state (again, the negative goal is not assumed as it can be compiled away). A solution to the planning problem is a sequence of actions $a_1, a_2, \ldots, a_n$ such that $S = \gamma(...\gamma(\gamma(S_0, a_1), a_2), ..., a_n)$ and $G^+ \subseteq S$. This sequence of actions is called a *plan*.

III. Background on MAPF

An *instance of MAPF* is defined as a pair (G, A), where $G = (V, E)$ is a graph representing the shared environment and A is a set of agents. Each agent $a_i \in A$ is a pair $a = (a_0, a_+)$, where a_0 represents the initial location (or start

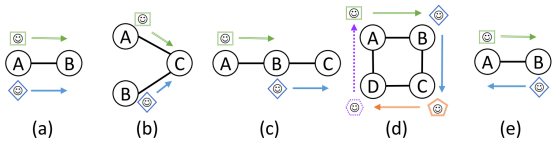

Fig. 1: Types of possible conflicts. From left to right: an edge conflict, a vertex conflict, a following conflict, a cycle conflict, and a swapping conflict. Figure taken from [5]

location), and a_+ represents the goal location of agent a. In both cases, location corresponds to a node in the input graph.

Time is assumed to be discretized and at each timestep, each agent can either move to a neighboring node or not move – perform wait action. All agents are moving simultaneously.

A solution to MAPF is a sequence of nodes π_i for each agent $a_i \in A$ such that $\pi_i(0)$ is the start location of a_i, $\pi_i(|\pi_i|)$ is the goal location of a_i and all of the nodes in between form a valid path. Furthermore, we request that there are no conflicts among the agents. Using MAPF terminology [5], we distinguish 5 possible conflicts (see Figure 1).

For most realistic applications, where the moving agents are physical entities (cars, ships, planes, etc.) the edge, vertex, and swapping conflict are forbidden, since in those cases, the agents share the same physical location. As for the following and cycle conflict, it depends on the intended application.

In this paper we will focus on two possible settings: (I) *pebble-motion* – all of the defined conflicts are forbidden and (II) *parallel-motion* – edge, vertex, and swapping conflict are forbidden while following and cycle conflicts are allowed.

IV. Problem Modeling

The problem that we address in this paper, is how to represent valid MAPF plans as classical sequential plans. Recall that the agents in the MAPF plan move in parallel, while the classical plan is a sequence of actions.

A. Sequential models

A straightforward sequential planning model for MAPF uses only `move` actions without explicit waiting actions – when one agent moves, all other agents wait. We can model the location of each agent via predicates `at/2` (the number two indicates the number of attributes of the predicate; in the case of `at/2`, the first attribute indicates the robot and the second attribute the location of that robot). For the `move` action we need to ensure that the node, to which an agent moves, is not occupied by another agent. As we do not use negative preconditions, we can use another predicate `free/1` to describe that a node is not occupied. At the initial state, each node is either free or there is exactly one agent at the node. This invariant is kept in every state – if an agent moves away from some node, the node becomes free, and vice versa, when an agent moves to some node, that node becomes occupied by that agent and it is no longer free. The planning goal is specified as goal locations of individual agents. The model of the `move` action in PDDL may then look like this (we believe that PDDL syntax is straightforward and it is easy to map to the formal definition of a planning action):

```
(:action move
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex)
    :precondition (and
        (at ?A ?ORIG)
        (connected ?ORIG ?DEST)
        (free ?DEST) )
    :effect (and
        (at ?A ?DEST)
        (free ?ORIG)
        (not (at ?A ?ORIG))
        (not (free ?DEST)) ) )
```

The only additional predicate used by the above model is `connected/2`, which models the graph. In this straightforward model exactly one agent moves to an empty node at any time. Clearly, the sequential plan is a valid plan for MAPF and it corresponds to the pebble-motion – there is no conflict from those shown in Figure 1 because there are no parallel actions. As this plan uses exactly one action at each time step, it has a larger makespan than plans with parallel actions.

1) Plan parallelisation: We can parallelise the sequential plan by cutting the original plan into sub-plans, where each sub-plan is a continuous sequence of `move` actions of different agents. Actions from each sub-plan are then executed in parallel and other agents are waiting during that parallel step.

To prove that this parallelisation produces a valid MAPF plan, we examine each of the 5 possible conflicts. At each parallel step, every agent moves at most once, we assume that the sequential plan is valid, and we do not change the order of the move actions.

Vertex conflict occurs when 2 agents move into the same node before one of them moves out, however, this would cause a vertex conflict in the sequential plan as well. Therefore the parallelisation can not cause a *vertex* conflict. If there are no *vertex* conflict then *edge* conflicts are also impossible, since there can not be 2 agents present in the same node to move over the same edge.

Assume that agents $a_1, \ldots, a_k$ move on a cycle and agent a_1 moves first in the sequential plan. Then a_1 cannot move to its destination node, as this node is still occupied by agent a_2 that will move later in the sequential plan. Hence no such cycle can be obtained by parallelisation as it would violate the preconditions of actions. For this reason both *cycle* and *swapping* conflicts are not possible. Notice that due to the above reasons, from pebble-motion plans we can never obtain parallel plans where cycles are present in a parallel step.

Finally, the *following* conflict may appear in the parallel plan. Assume a simple sequential plan `((move a1 n1 n2) (move a2 n0 n1))`. During parallelisation, these two actions are put to a single parallel step as they deal with different agents. The agents move like a train at this step – there is the *following* conflict. To prevent the *following* conflicts, we can check them during parallelisation so we cut the sequential plan to smaller sub-plans (in the worst case, we got the original sequential plan with no parallel steps).

In summary, the pebble-motion model describes MAPF problems where the edge, vertex, swap, and cycle conflicts are forbidden. The *following* conflict may or may not be

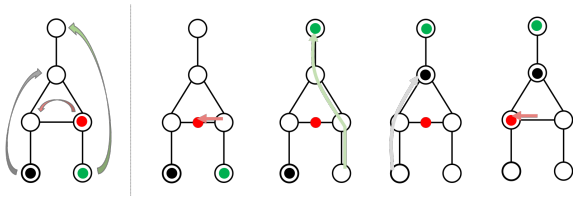

Fig. 2: Invalid MAPF plan if one agent (red) stays at the edge while other agents go through the end nodes of that edge.

allowed depending on how parallelisation is realized. The open question now is if it is possible to design a sequential planning model where the *cycle* conflict is also allowed when we parallelise the sequential plan.

2) Modeling cycles: To model the cyclic movement, we utilize two ideas. First, instead of doing a complete move to the next node, an agent `a1` moves only to the edge, which must be free at the time of the move. We call it opening the move. Agent `a1` completes (closes) the move to its destination node later, when that node is empty, which may happen immediately after or later after another agent `a2` staying at the node leaves it. The danger here is that while agent `a1` stays on the edge, other agents may use the end nodes of that edge, which may give invalid MAPF plans (see Figure 2). Hence the second idea of the model is using a freeze mechanism to disallow usage of these nodes by other agents.

Each original move action is now modelled using two actions – an opening action moves an agent to the edge and a closing action moves the agent from the edge to the destination node. When an agent is on the edge, the edge needs to be marked as occupied in both directions so no other agent moves to the same edge (predicate `busyLink/3`). We distinguish two situations there – if the destination node is already free, the agent freezes that node so no other agent can use it before the agent completes its move; if the destination node is occupied, we switch to a special mode `lock`, where the agent pushes the other agent out of the destination node. The destination cannot be frozen yet, but it is required from the other agent and when the other agent leaves it, the node will be frozen. The following two actions `freeze` and `require` show these possible openings of movement. Notice that instead of the predicate `connected/2` , we now use `freeLink/2` and `busyLink/3` to model the graph and the agent that occupies the edge.

```
(:action freeze
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex)
    :precondition (and
        (normal)
        (at ?A ?ORIG)
        (freeLink ?DEST ?ORIG)
        (freeLink ?ORIG ?DEST)
        (free ?DEST) )
    :effect (and
        (free ?ORIG)
        (frozen ?A ?DEST)
        (busyLink ?DEST ?ORIG ?A)
        (busyLink ?ORIG ?DEST ?A)
        (not (free ?DEST))
        (not (at ?A ?ORIG))
        (not (freeLink ?DEST ?ORIG))
        (not (freeLink ?ORIG ?DEST)) ) )
```

```
(:action require
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex
     ?AN - agent)
    :precondition (and
        (normal)
        (at ?A ?ORIG)
        (freeLink ?DEST ?ORIG)
        (freeLink ?ORIG ?DEST)
        (at ?AN ?DEST) )
    :effect (and
        (lock)
        (free ?ORIG)
        (require ?A ?DEST)
        (blocking ?AN ?DEST)
        (busyLink ?DEST ?ORIG ?A)
        (busyLink ?ORIG ?DEST ?A)
        (not (normal))
        (not (at ?A ?ORIG))
        (not (at ?AN ?DEST))
        (not (freeLink ?DEST ?ORIG))
        (not (freeLink ?ORIG ?DEST)) ) )
```

As soon as the blocking agent moves away it can leave the node frozen for the agent that required the node. The information about the agent who made the request is stored in the predicate `require/2`. Note that only one agent is allowed to require a certain node because the predicate `(at ?AN ?DEST)` is replaced by `(blocking ?AN ?DEST)` in the state. If the destination node has already been free, the action `freeze` marks the node with the predicate `frozen/2` so that the agent can immediately complete its move using the following action `finishMove`:

```
(:action finishMove
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex)
    :precondition (and
        (normal)
        (frozen ?A ?DEST)
        (busyLink ?DEST ?ORIG ?A)
        (busyLink ?ORIG ?DEST ?A) )
    :effect (and
        (at ?A ?DEST)
        (freeLink ?ORIG ?DEST)
        (freeLink ?DEST ?ORIG)
        (not (frozen ?A ?DEST))
        (not (busyLink ?DEST ?ORIG ?A))
        (not (busyLink ?ORIG ?DEST ?A)) ) )
```

The action `finishMove` is used for all agents to close their move to the destination node. However, if the agent starts moving using the action `required` we still need to freeze its destination node, since it is occupied. In this situation, we temporarily forbid all other movements, by using the predicate `lock`. In the `normal` mode the agents can open and close movements, while in the `lock` mode, the cycles and trains are being resolved by moving all of the required agents to the edges and freezing their destination nodes. This is done by action `passRequire` that moves the blocking agent to an edge, while requiring the destination node from another agent. Notice that this action also freezes the origin node for the previous agent.

```
(:action passRequire
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex
     ?AN - agent ?AP - agent)
    :precondition (and
        (lock)
        (require ?AP ?ORIG)
        (blocking ?A ?ORIG)
        (freeLink ?DEST ?ORIG)
        (freeLink ?ORIG ?DEST)
        (at ?AN ?DEST) )
    :effect (and
```

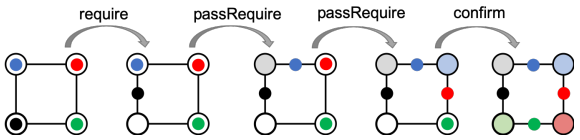

Fig. 3: Describing cycles in the sequential model, the shadow nodes are frozen (the color indicates for which agent) so other agents cannot used them.

```
    (frozen ?AP ?ORIG)
    (require ?A ?DEST)
    (blocking ?AN ?DEST)
    (busyLink ?DEST ?ORIG ?A)
    (busyLink ?ORIG ?DEST ?A)
    (not (require ?AP ?ORIG))
    (not (blocking ?A ?ORIG))
    (not (at ?AN ?DEST))
    (not (freeLink ?DEST ?ORIG))
    (not (freeLink ?ORIG ?DEST)) ) )
```

The `lock` mode is abandoned when the last agent moves to the edge and its destination node is free. This agent can be the head of the train or it could be the last agent in the cycle. Notice that the agent that started the `lock` mode via action `require` actually left its origin node and made that node free. This allows the last agent to use it and close the cycle. The following action does this resolving move (i.e. close the loop or move train head) and returns to the `normal` mode, where the agents at edges can close their moves.

```
(:action confirm
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex
         ?AP - agent)
    :precondition (and
        (lock)
        (require ?AP ?ORIG)
        (blocking ?A ?ORIG)
        (freeLink ?DEST ?ORIG)
        (freeLink ?ORIG ?DEST)
        (free ?DEST) )
    :effect (and
        (normal)
        (frozen ?A ?DEST)
        (frozen ?AP ?ORIG)
        (busyLink ?DEST ?ORIG ?A)
        (busyLink ?ORIG ?DEST ?A)
        (not (lock))
        (not (free ?DEST))
        (not (require ?AP ?ORIG))
        (not (blocking ?A ?ORIG))
        (not (freeLink ?DEST ?ORIG))
        (not (freeLink ?ORIG ?DEST)) ) )
```

After switching from `lock` mode to `normal` mode, all participating agents are sitting at edges and their destination nodes are frozen so no other agent can use them (see Figure 3). In this situation, other agents can start moving, other loops or trains may be formed or agents close their moves from the edge to the next node.

To prove that the valid sequential plan represents a valid MAPF plan, we show how to parallelise it. In any valid sequential plan, we can reshuffle the actions as follows. For each opening action `freeze` we find the corresponding closing action `finishMove` and move it backward in the plan to be right after `freeze`. This is possible because all preconditions of `finishMove` are already provided by `freeze` and no action between their original positions in the plan altered these preconditions. Actions `require`, `passRequire`, and `confirm` will always be grouped to a sequence starting with `require`, followed by a (possibly empty) sequence of actions `passRequire`, and concluded by the action `confirm`. We can move their corresponding closing actions `finishMove` right after the action `confirm` finishing the block. This is again possible due to the same reasons as above. After reshuffling, the sequential plan has a specific structure consisting of pairs of actions `freeze` and `finishMove` for the same agent – they represent move of that agent – and a group of actions `require`, `passRequire`, and `confirm` followed immediately by the closing actions `finishMove` for all the agents in that group – this group represents a parallel move of all agents in the group, it could be a train or a cycle.

The obtained sequential plan is a valid MAPF plan with no edge, vertex, and swapping conflicts. The following and cycle conflicts are allowed. Let us just highlight, that a cycle with only two agents (the swapping conflict) is still not allowed, as the second agent would need to use the same edge as the first agent, just in the opposite direction, but this edge has already been occupied by the first agent (`busyLink`). Note finally, that we can use the same parallelisation as we discussed for the pebble-motion model, only the groups of actions in the `lock` mode must always be part of a single parallel step. This is possible since in each such group each agent does exactly one move which follows the principle of parallelisation – no agent appears in more than one move action.

B. Layered model

We will now describe yet another model that forces the planner to produce plans with a specific layered structure that can be easily translated into a parallel plan. The main idea is to make all agents act, either move or wait, by passing a token until all the agents take a turn. This is realized through predicate `token/1` that marks the active agent and the predicate `next/2` that marks the next agent to take an action. The `next/2` predicate is defined in the initial state and it defines a loop over all agents in the particular planning task. Every action in this model passes the token on to the next agent. In this way, the actions in the resulting plan are always structured into layers where each agent takes a turn exactly once. Two such layers contain one valid parallel step. Following the same principle as in the model described above, agents first move from nodes to edges using action `startMove`:

```
(:action startMove
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex
                 ?NA - agent)
    :precondition (and
        (token ?A)
        (next ?A ?NA)
        (ready ?A)
        (at ?A ?ORIG)
        (freeLink ?DEST ?ORIG)
        (freeLink ?ORIG ?DEST) )
    :effect (and
        (token ?NA)
        (free ?ORIG)
        (moving ?A ?DEST)
        (busyLink ?ORIG ?DEST ?A)
        (busyLink ?DEST ?ORIG ?A)
```

```
        (not (token ?A))
        (not (ready ?A))
        (not (at ?A ?ORIG))
        (not (freeLink ?DEST ?ORIG))
        (not (freeLink ?ORIG ?DEST)) ) )
```

Then, agents move from edges to their destination nodes using action `finishMove`. The predicate `moving/2` prevents the agent from moving from node to an edge in one layer and returning back to the same node in the next layer.

```
(:action finishMove
    :parameters (?A - agent ?ORIG - vertex ?DEST - vertex
                 ?NA - agent)
    :precondition (and
        (token ?A)
        (next ?A ?NA)
        (moving ?A ?DEST)
        (free ?DEST)
        (busyLink ?ORIG ?DEST ?A)
        (busyLink ?DEST ?ORIG ?A) )
    :effect (and
        (token ?NA)
        (ready ?A)
        (at ?A ?DEST)
        (freeLink ?ORIG ?DEST)
        (freeLink ?DEST ?ORIG)
        (not (token ?A))
        (not (free ?DEST))
        (not (moving ?A ?DEST))
        (not (busyLink ?ORIG ?DEST ?A))
        (not (busyLink ?DEST ?ORIG ?A)) ) )
```

Splitting the movement into two parts is necessary when we need a group of agents to move in parallel along the cycle. Because the agents move to the edge first and in the second round, they complete the move to the next node, the cyclic movement can be directly realized. Notice also that the action `finishMove` ensures that the destination node is free. This prevents the edge and vertex conflicts. Freeze mechanism, as used by the sequential model, is not needed there because, in one round, all acting agents move to the edge and in the next round, they move to the free node. It is not possible for an agent to stay at the edge, while other agents use the nodes of that edge (the invalid MAPF plan as depicted in Figure 2 cannot be realized there).

Due to the token-passing mechanism, each agent must act in each round, so now we need to explicitly model the wait actions. Again, a pair of actions `startWait` and `finishWait` is used there. As the token is moving between the agents in a loop, the agent waiting in a node cannot distinguish the first and second round within a single layer just from the token. That is why we use predicate `ready` that basically expresses that the agent is neither moving nor waiting (recall that negative preconditions are not allowed).

```
(:action startWait
    :parameters (?A - agent ?V - vertex ?NA - agent)
    :precondition (and
        (token ?A)
        (next ?A ?NA)
        (ready ?A)
        (at ?A ?V) )
    :effect (and
        (token ?NA)
        (waiting ?A)
        (not (ready ?A))
        (not (token ?A)) ) )

(:action finishWait
    :parameters (?A - agent ?V - vertex ?NA - agent)
    :precondition (and
        (token ?A)
        (next ?A ?NA)
        (waiting ?A)
        (at ?A ?V) )
    :effect (and
        (token ?NA)
        (ready ?A)
        (not (waiting ?A))
        (not (token ?A)) ) )
```

The proof that any valid sequential plan is a valid MAPF plan is now straightforward as the sequence of actions can be naturally cut to parallel steps. As already mentioned, edge and vertex conflicts are forbidden there. The swapping conflict is also prevented as the agents block the edge using predicate `busyLink`. Trains and cycles are allowed as agents first move to the edge and then complete the move to the next node which was made free in the meantime. Notice also that the fixed order of agents in the token-passing mechanism does not clash with the train and cycle movement as agents can move to the edge in an arbitrary order.

V. Empirical evaluation

To compare the proposed models, we used MAPF instances from a benchmark set [5]. The maps selected were *empty8*, *empty16*, *random32*, *room32*, and *maze32*. The first two are maps with no obstacles and dimensions 8 by 8 and 16 by 16, respectively. The rest of the maps are of dimensions 32 by 32 with obstacles either randomly placed or placed to represent some structure. The number of agents for each map increases from 1 to 20. For each of the settings, 5 different instances were used. This gives us 500 instances in total.

We also executed the same set of instances on a makespan optimal SAT-based solver written in Picat programming language that automatically translates constraints into a proposition formulae [6]. Note that for the proposed PDDL models we used a solver that does not guarantee optimality with respect to any cost function – FF solver [7]. Comparing with an optimal solver will indicate how close to the optimum the plans are.

For each problem instance, the runtime was limited to 30 minutes and the used RAM was limited to 8GB. The computer used was equipped with Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz. For the planning-based models, we used FF solver [7] version 2.3, and for the SAT-based solvers, we used Picat language and compiler version 2.6#2.

To recapitulate, we will be comparing the following models:

- *Pebble* - The basic PDDL pebble motion model, i.e. no following or cycle conflicts.
- *Picat-Pebble* - SAT-based optimal solver that models pebble motion.
- *Layered* - The layered PDDL model with token that allows following conflicts and cycle conflicts.
- *Sequential* - The sequential PDDL model that allows following conflicts and cycle conflicts.
- *Picat* - SAT-based optimal solver that allows following conflicts and cycle conflicts.

To compare the efficiency of the models, we compute a PAR10 score (see Table I). This score reflects the runtime and adds an extra penalty for each timeouted instance. We

Pebble	Picat-Pebble	Layered	Sequential	Picat
13.6	40.0	5767.5	12226.3	289.2

TABLE I: PAR10 scores for all of the compared models.

compute the PAR10 as (*sum of all runtimes + penalty for each timeouted instance*) / *number of instances*. The penalty is defined as $TimeLimit \times 10$, therefore the total penalty is 18000. From the definition of the PAR10, we can see that the lower the score, the better the model performed.

Since modeling different constraints can lead to different complexities, we shall compare the models in two groups - the solvers that model pebble-motion (*Pebble* and *Picat-Pebble*) and the models that allow following and cycle conflicts (*Layered*, *Sequential* and *Picat*). We start with the latter.

From the obtained scores, we can see that there is a clear ordering in performance for the models. The optimal SAT-based solver achieved the best results with only 5 timeouted instance. The models proposed in this paper are ordered *Layered* with 144 and *Sequential* with 308 timeouted instances. The result clearly shows the advantage of a rather simple model *Layered* over carefully crafted representation *Sequential*.

The ordering in efficiency of the models can be further seen in Figure 4, which shows how many instances (x-axis) were solved in a given time limit (y-axis). Again, the lower the line, the better the model performed. The ordering in performance remains the same as with the PAR10 scores. This is still true when comparing results for individual types of maps.

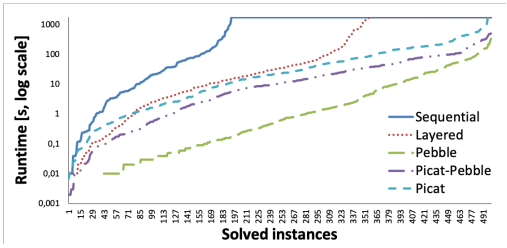

Fig. 4: Number of instances each of the described model solved in a given time limit.

The order in which the agents pass the token in the Layered model is not strictly defined. We ran experiments with different orderings – random order, and ascending and descending order by the length of the shortest path between agents start and goal node. The experiments did not prove that there is any significant difference in the order of the agents.

When comparing the quality of the plan, the simple model *Layered* is not so bad in comparison with the optimal solver. Out of the 356 instances, the model *Layered* solved, 232 were solved optimally. Those that were not solved optimally had a solution on average 6.2% worse than optimum.

For the layered models, it is simple to obtain the length of MAPF plan from the number of actions as $|\pi| = \left\lceil \frac{\texttt{number of actions}}{2 \times \texttt{number of agents}} \right\rceil$

Finding a parallel MAPF plan from the actions of model *Sequential* is not as straight forward, so we do not include this measure.

As for the solvers using the pebble-motion model variants, the planning-based solver outperformed the optimal SAT-based solver. This is another example that models with simple actions can be very efficient. Note, however, that the planning-based solver is not optimal with regards to any cost function.

VI. Conclusions

In the paper, we proposed PDDL models to describe MAPF problems. The straightforward pebble-motion model cannot describe plans with cycles, but it achieved very good performance. We extended the model to cover cycles, but the performance degraded significantly. Nevertheless, this sequential model gives the core idea of splitting actions into opening actions moving the agent to the edge and closing actions moving the agent from the edge to the destination node. By forcing agents to do these actions in a synchronous way using the token-passing mechanism we naturally define the parallel MAPF steps in the classical sequential plan. This layered model also achieved very good performance.

The paper also gives some modelling ideas that could be applicable to other problems, especially, when synchronization is needed. The experiments showed that simpler planning domain models (pebble-motion and layered models) can achieve much better efficiency in comparison with carefully designed but complex models (sequential model).

Acknowledgment: Research is supported by the project P103-19-02183S of the Czech Science Foundation, and by the project 90119 of the Charles University Grant Agency.

References

[1] P. Surynek, "Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, S. Kraus, Ed. ijcai.org, 2019, pp. 1177–1183. [Online]. Available: https://doi.org/10.24963/ijcai.2019/164

[2] R. Barták and J. Vodrážka, "An experimental study of influence of modeling and solving techniques on performance of a tabled logic programming planner," *Fundam. Inform.*, vol. 149, no. 1-2, pp. 35–60, 2016. [Online]. Available: https://doi.org/10.3233/FI-2016-1442

[3] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "Pddl - the planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep. CVC TR98003/DCS TR1165, 1998.

[4] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," in *Proceedings of the 2nd international joint conference on Artificial intelligence*, ser. IJCAI'71, San Francisco, CA, USA, 1971, pp. 608–620. [Online]. Available: http://dl.acm.org/citation.cfm?id=1622876.1622939

[5] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. T. Walker, J. Li, D. Atzmon, L. C. adn T. K. Satish Kumar adn Eli Boyarski, and R. Barták, "Multi-agent pathfinding: Definitions, variants, and benchmarks," in *the International Symposium on Combinatorial Search (SoCS)*, 2019.

[6] R. Barták, N. Zhou, R. Stern, E. Boyarski, and P. Surynek, "Modeling and solving the multi-agent pathfinding problem in picat," in *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*. IEEE Computer Society, 2017, pp. 959–966. [Online]. Available: https://doi.org/10.1109/ICTAI.2017.00147

[7] J. Hoffmann and B. Nebel, "The FF planning system: Fast plan generation through heuristic search," *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, 2001.