Multi-agent Path Finding on Real Robots: First Experience with Ozobots

Roman Barták(✉), Jiří Švancara, Věra Škopková, and David Nohejl

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

Abstract. The problem of Multi-Agent Path Finding (MAPF) is to find paths for a fixed set of agents from their current locations to some desired locations in such a way that the agents do not collide with each other. This problem has been extensively theoretically studied, frequently using an abstract model, that expects uniform durations of moving primitives and perfect synchronization of agents/robots. In this paper we study the question of how the abstract plans generated by existing MAPF algorithms perform in practice when executed on real robots, namely Ozobots. In particular, we use several abstract models of MAPF, including a robust version and a version that assumes turning of a robot, we translate the abstract plans to sequences of motion primitives executable on Ozobots, and we empirically compare the quality of plan execution (real makespan, the number of collisions).

Keywords: Path planning · Multi-agent systems · Real robots

1 Introduction

Multi-agent path finding (MAPF) recently attracted a lot of attention of AI research community. It is a hard problem with practical applicability in areas such as warehousing and games. Frequently, an abstract version of the problem is solved, where a graph defines possible locations (vertices) and movements (edges) of agents and agents move synchronously. At any time, no two agents can stay in the same vertex to prevent collisions so the obtained plans are collision free and hence blindly executable. The plan of each agent consists of move (to a neighboring vertex) and wait (in the same vertex) actions. Makespan and sum-of-cost (plan lengths) are two frequently studied objectives.

In this paper, we focus on answering two questions: how to execute abstract plans obtained from existing MAPF algorithms and models on real robots and how the quality of abstract plans is reflected in the quality of executed plans. The goal is to verify if the abstract plans are practically relevant and, if the answer is no (as expected), to provide feedback to improve abstract models to be closer to reality. We use a fleet of Ozobot Evo robots to perform the plans. These robots provide motion primitives, for example, they can turn left/right, follow a line, and recognize line junction, so it is not necessary to solve classical

G. R. Simari et al. (Eds.): IBERAMIA 2018, LNAI 11238, pp. 290–301, 2018.
https://doi.org/10.1007/978-3-030-03928-8_24

robotics tasks such as localization. Though the robots have proximity sensors, the plans are executed blindly based on the MAPF setting as the plans should already be collision free.

Specifically, we explore the very classical MAPF setting as described above, the k-robust setting [1], where a gap is required between the robots to compensate possible delays during execution, and finally a model that directly encodes turning operations (the classical setting does not assume direction of movement). The abstract plans are then translated to motion primitives, which consist of forward movement, turning left/right, and waiting. We explore different durations of these primitives to see their effect on robot synchronization. As far as we know this is the first study of practical quality of plans obtained from abstract MAPF models.

The paper is organized as follows. We will first introduce the abstract MAPF problem formally and survey approaches for its solving. Then we will give more details on why it is important to look at the execution of abstract plans on real robots. After that, we will describe all the models used in this study and how they are translated to executable primitives of Ozobot Evo robots. Finally, we will describe our experimental setting and give results of an empirical evaluation.

2 The MAPF Problem

Formally, the MAPF problem is defined by a graph $G = (V, E)$ and a set of agents $a_1, \ldots, a_k$, where each agent a_i is associated with starting location $s_i \in V$ and goal location $g_i \in V$. The time is discrete and in every time step each agent can either move from its location to a neighboring location or wait in its current location. A grid map with a unit length of each edge is often used to represent the environment [10]. We will also be using this type of maps in this paper.

Let $\pi_i[t]$ denote the location (vertex of graph G) of agent a_i at time step t. Plan π_i is the sequence of locations for agent a_i. The MAPF task is to find a valid plan π that is a union of plans of all agents. We say that π is valid if (i) each agent starts and ends in its starting and goal location respectively, (ii) no two agents occupy the same vertex at the same time, and (iii) no two agents move along the same edge at the same time in opposite directions (they do not swap their positions). Formally this can be written as:

(i) $\forall i : \pi_i[0] = s_i \wedge \pi_i[T] = g_i$, where T is the last time step.
(ii) $\forall t, i \neq j : \pi_i[t] \neq \pi_j[t]$
(iii) $\forall t, i \neq j : \pi_i[t] \neq \pi_j[t+1] \vee \pi_i[t+1] \neq \pi_j[t]$.

We denote $|\pi_i|$ as the length of plan for agent a_i. Then we can define an objective function that measures the quality of the found valid plan π.

$$Makespan(\pi) = \max_i |\pi_i|$$

The makespan objective function is well known and often studied in the literature [15]. It can be shown that when we require the solution to be makespan optimal (i.e. a solution with minimal makespan), the problem is NP-hard [17].

To solve MAPF optimally, one can generally use algorithms from one of the following categories:

1. **Reduction-based solvers** are solvers that reduce MAPF to another known problem such as SAT [14], integer linear programming [16], and answer set programming [6]. These approaches are based on using fast solvers for given formalism and consist mainly of translating MAPF to that formalism.
2. **Search-based solvers** include variants of A* over a global search space – all possibilities how to place agents into the nodes of the graph [13]. Other make use of novel search trees [4,11,12] that search over some constraints put on the agents.

Though the plans obtained by different MAPF solvers might be different, the optimal plans are frequently similar and tight (no superfluous steps are used). As solving MAPF is not the topic of this paper (we focus on evaluating the practical relevance of obtained plans), any optimal MAPF solver can be used. We decided for the reduction-based solver implemented in the Picat programming language [3] that uses translation to SAT. This solver has performance comparable to state-of-the-art solvers and has the advantage of easy modification and extension of the core model, for example adding further constraints or using numerical constraints.

The Picat solver (like other reduction-based solvers) follows the planning-as-satisfiability framework [8], where a layered graph is used to encode the plans of a given length. Each layer describes positions of all agents in a given time step. As the plan length is unknown, the number of layers is incrementally increased until a solvable model is obtained. A Boolean variable B_{tav} indicates if agent a ($a = 1, 2, \ldots, k$) occupies vertex v ($v = 1, 2, \ldots, n$) at time t ($t = 0, 1, \ldots, m$). The following constraints ensure the validity of every state and every transition:

(1) Each agent occupies exactly one vertex at each time.

$\Sigma_{v=1}^{n} B_{tav} = 1$ for $t = 0, \ldots, m$, and $a = 1, \ldots, k$.

(2) No two agents occupy the same vertex at any time.

$\Sigma_{a=1}^{k} B_{tav} \leq 1$ for $t = 0, \ldots, m$, and $v = 1, \ldots, n$.

(3) If agent a occupies vertex v at time t, then a occupies a neighboring vertex at time $t + 1$.

$B_{tav} = 1 \Rightarrow \Sigma_{u \in neibs(v)}(B_{(t+1)au}) \geq 1$
for $t = 0, \ldots, m - 1$, $a = 1, \ldots, k$, and $v = 1, \ldots, n$.

The model consists of $k \times (m + 1) \times n$ Boolean variables, where k is the number of agents, m is the makespan, and n is the number of vertices in the graph. Further constraints can be added easily, for example, to prevent swaps or to introduce robustness. Figure 1 shows the executable Picat code with the core model to demonstrate how close the program is to the abstract model.

```
import sat.

path(N,As) =>
    K = len(As),
    lower_upper_bounds(As,LB,UB),
    between(LB,UB,M),
    B = new_array(M+1,K,N),
    B :: 0..1,

    % Initialize the first and last states
    foreach (A in 1..K)
        (V,FV) = As[A],
        B[1,A,V] = 1,
        B[M+1,A,FV] = 1
    end,

    % Each agent occupies exactly one vertex
    foreach (T in 1..M+1, A in 1..K)
        sum([B[T,A,V] : V in 1..N]) #= 1
    end,

    % No two agents occupy the same vertex
    foreach (T in 1..M+1, V in 1..N)
        sum([B[T,A,V] : A in 1..K]) #=< 1
    end,

    % Every transition is valid
    foreach (T in 1..M, A in 1..K, V in 1..N)
        neibs(V,Neibs),
        B[T,A,V] #=>
            sum([B[T+1,A,U] : U in Neibs]) #>= 1
    end,

    solve(B),
    output_plan(B).
```

Fig. 1. A program in Picat for MAPF.

3 Motivation and Contribution

The abstract plan outputted by MAPF solvers is, as defined, a sequence of locations that the agents visit. However, a physical agent has to translate these locations to a series of actions that the agent can perform. We assume that the agent can turn left and right and move forward. By concatenating these actions, the agent can perform all the required steps from the abstract plan (recall, that we are working with grid worlds). This translates to five possible actions at each time step - (1) wait, (2) move forward, (3, 4) turn left/right and move, and (5) turn back and move. As the mobile robot cannot move backward directly,

turning back is implemented as two turns right (or left). For example, an agent with starting location in v_1 and goal location in v_7 in Fig. 2 has an abstract plan of seven locations. However, the physical agent has to perform four additional turning actions that the classical MAPF solvers do not take into consideration.

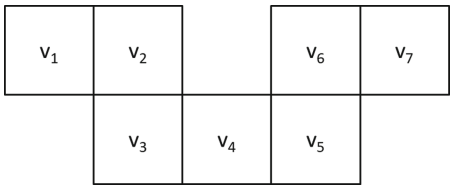

Fig. 2. Example of graph where an agent has to perform turning actions.

As the abstract steps may have durations different from the physical steps, the abstract plans, which are perfectly synchronized, may desynchronize when being executed, which may further lead to collisions. This is even more probable in dense and optimal plans, where agents often move close to each other.

The intuition says that such desynchronization will indeed happen. In the paper, we will empirically verify this hypothesis and we will explore several abstract models for MAPF and the output transformations to robot actions. These models not only try to keep the agent synchronous during the execution of the plan but also to avoid collisions caused by some small unforeseen flaw in the execution. We then compare and evaluate these models on an example grid using real robots. Note that the real robots only blindly follow the computed plan and cannot intervene if, for example, an obstacle is detected.

4 Models

In this section, we describe the studied abstract MAPF models and possible transformations of abstract plans to executable sequences of physical actions. Let t_t be the time needed by the robot to turn by 90° to either side and t_f be the time to move forward to the neighboring vertex in the grid. Both t_t and t_f are nonzero. The time spend while the agent is performing the wait operation t_w will depend on each model.

4.1 Classical Model

The first and most straightforward model is a direct translation of the abstract plan to the action sequence. We shall call this a *classic* model. At the end of each timestep, an agent is facing in a direction. Based on the next location, the agent picks one of the five actions described above and performs it. This means that all move actions consist of possible turning and then going forward. There are no independent turning moves. As the two most common actions in abstract

plans are (2) and (3, 4), we suggest to set the time t_w of waiting actions to be $t_f + 1/2 * t_t$ as the average of durations of actions (2) and (3, 4).

One can easily see that this simple model can be prone to desynchronization, as turning adds time over agents that just move forward. Recall Fig. 2 and suppose there is another agent with the same number of steps, but all of the actions are moving forward. This agent will reach its goal $4 * t_t$ sooner than the agent from the example.

To fix this synchronization issue, we introduce a *classic + wait* model. The basic idea is that each abstract action takes the same time, which is realized by adding some wait time to "fast" actions. The longest action is (5), therefore each action now takes $2 * t_t + t_f$ including the waiting action. The consequence is that plan execution takes longer time, which may not be desirable.

Note that both of these models do not require the MAPF algorithm and model to change. They only use different durations of abstract actions which are implemented in the translation of abstract plans to executable actions.

4.2 Robust Model

Another way to fix the synchronization problem is to create a plan π that is robust to possible delays during execution. The k-robust plan is a valid MAPF plan that in addition requires for each vertex of the graph to be unoccupied for at least k time steps before another agent can enter it [1]. In our experiments, we choose k to be 1. We presume that this is a good balance between keeping the agents from colliding with each other while not prolonging the plan too much. The 1-robust plan is then translated to executable actions using the same principle as the *classic* model. This yields a *1-robust* model.

The synchronization issue is not fixed in a guaranteed way, but hopefully, collisions are avoided as the agents tend to not move close to each other.

4.3 Split Actions Model

One may assume that executable actions might be directly represented in the abstract model. In particular, the need to turn can be represented by an abstract turning action. In the reduction-based solvers, this can be done by splitting each vertex v_i from the original graph G into four new vertices $v_i^{up}, v_i^{right}, v_i^{down}, v_i^{left}$ indicating directions where the agent is facing to. The new edges now represent the turn actions, while the original edges correspond to move only actions, see Fig. 3. Note that when an agent leaves a vertex facing some direction, it will arrive to the neighboring vertex also facing that direction. This change to the input graph also requires a change in the MAPF solver (constraints), because the split vertices need to be treated as one to avoid collisions of type (ii). This means that at any time there can be at most one agent in those four vertices representing a given location. The abstract plan is then translated to an executable plan in a direct way as the agent is given a sequence of individual actions wait, turn left/right, and move forward. The waiting time t_w is set as the bigger time of the remaining actions: $t_w = \max(t_t, t_f)$. We shall call this a *split* model.

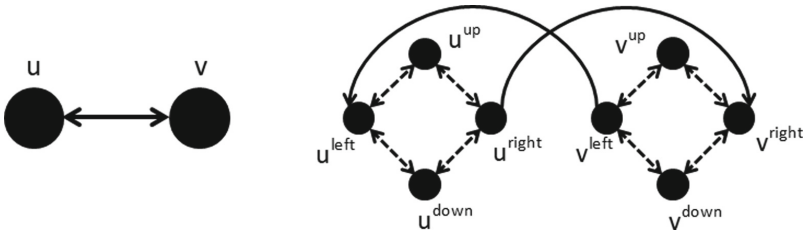

Fig. 3. Example of how two horizontally connected vertices (left) are split into new vertices (right) describing possible agent's orientations. The dotted edges correspond to turning actions.

A synchronization issue is still present in the *split* model, if the times t_t and t_f are not the same. Recall that the solvers assume equal durations of all actions. To fix this, we will use a notion from weighted MAPF [2]. Each edge in the graph is assigned an integer value that denotes its length. The weighted MAPF solver finds a plan that takes these lengths into account. Formally this can cause gaps in the plan of an agent as the agent may not be present in any vertex in the next step because the agent is still moving over an edge. This indeed does not break our definitions and the time is still discrete, only more finely divided. The lengths of turning edges are assigned a length of t_t and the other edges are assigned a length of t_f (or its scaled value to integers). The waiting time t_w is set as the smaller time of the remaining actions: $t_w = \min(t_t, t_f)$. We shall call this a *weighted-split* model or *w-split* for short.

A final enhancement to the *weighted-split* model is to introduce k-robustness there. This will again ensure that the agents do not tend to move close to each other to avoid undesirable collisions. In this case, however, it is not enough to use 1-robustness, as the plan is split into more time steps. Instead, we use $\max(t_t, t_f)$-robustness. We shall call this *robust-weighted-split* model or *rw-split* for short.

5 Experiments

The proposed models for MAPF were empirically evaluated on real robots and in this chapter we will present the obtained results. We shall first give some details on robots, that we used, and on the problem instance.

5.1 Ozobots

The robots used were Ozobot Evo from company Evollve [9]. These are small robots (about 3 cm in diameter) shown in Fig. 4. We have chosen them because their built–in actions are close to actions needed in the MAPF problems so there is no need to do low–level robotic programming. The robots are programmable through a programming language Ozoblockly [7] which is primarily meant as a

teaching tool for children. The program is uploaded to the robot and then the robot executes it. Most importantly, the robots have sensors underneath that allow the robot to follow a line and to detect intersection. An intersection is defined as at least two lines crossing each other. The robots also have forward and backward facing proximity sensors allowing them to detect obstacles. We used them to synchronize the start of robots (see further), but we did not exploit sensors further during plan execution. In addition, the robots have LED diodes and speakers that act as the robots output. We use them to indicate some states of the robot such as a finished plan. The moving speed and turning speed can be adjusted up to a speed limit of the robot.

Fig. 4. Ozobot Evo from Evollve used for the experiments. Picture is taken from [9].

There are some drawbacks in the simplicity of the robots. The main one is that there is currently no communication between multiple robots and therefore starting an instance of MAPF for all of the present robots at the same time is difficult. To solve this problem, we used the proximity sensors and forbid the agents to start performing the computed plan if an obstacle is present in front of them. An obstacle was placed in front of all of the agents and once they were ready to start executing the plan, all of the obstacles were removed. This ensured that the start time was identical and any desynchronization at the end of the plan was caused during the execution and not at the start.

5.2 Problem Instance

An instance was created to test the described models. It is a 5 by 8 grid map that was obtained by randomly removing vertices and edges in such a way that the rest of the graph still remained connected. This yielded map shown in Fig. 5. As opposed to the usual representation, where agents reside in the cells in between lines, here the agents follow the line and the vertex is represented as the crossing of two lines. This map was then printed on A3 paper in a scale such that each edge is 5 cm long and the line is 5 mm thick as per Ozobots recommended specification. The edge length was chosen to allow two robots to safely stay in neighboring nodes and to observe even minor desynchronization due to turning (if the edges are longer than the duration of moving is much bigger than the duration of turning and hence the effect of extra turning actions is less visible).

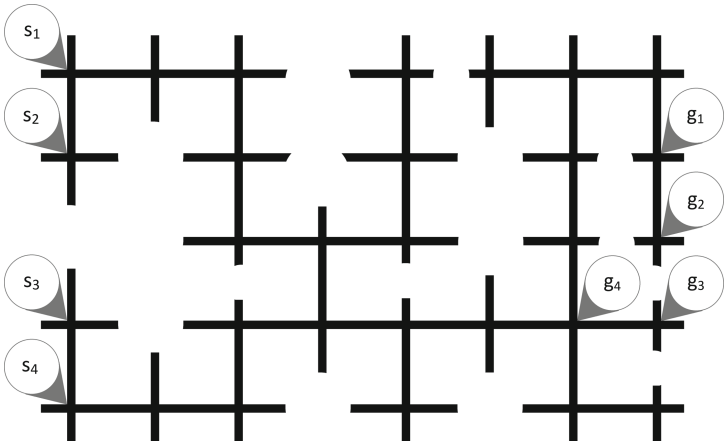

Fig. 5. Instance map for Ozobots. Ozobots follow the black line, the gray circles indicate starting and goal locations. They were not actually printed.

We used four robots; their initial and goal locations are also indicated in Fig. 5. These locations were chosen to ensure several bottlenecks in the map that will force the agents to navigate close to each other.

The speed of the robots was set in such a way that moving along a line takes 1600 ms and turning takes 800 ms. This means that $t_f = 1600$ and $t_t = 800$, however since the numbers are both divisible by 800, we can simplify the times for the MAPF solver to $t_f = 2$ and $t_t = 1$. This then gives us all required times for the models as described in the previous section.

5.3 Results

We generated plans using each MAPF model for the problem instance described above and then we executed the plans five times in total for each model. Several properties were measured with results shown in Table 1.

Table 1. Measured performance of Ozobots using each proposed model.

	Computed makespan	Failed runs	Number of collisions	Total time [s]	Max Δ time [s]
classic	17	5	4	NA	5
classic + wait	17	0	4.2	53	0
1-robust	19	0	0	41	4
split	27	0	2	36	3
w-split	45	0	2.6	39	0
rw-split	47	0	0	39	0

Computed makespan is the makespan of the plan returned by the MAPF solver. It is measured by the (weighted) number of abstract actions. Note that

the *split* models have larger makespan than the rest because the *split* models use a finer resolution of actions, namely turning actions are included in the makespan calculation. This is even more noticeable with *w-split* and *rw-split*, where the moving-forward action has a duration (weight) of two.

The number of failed runs is also shown. The only model that did not finish any run is the *classic* model while the rest managed to finish all of the runs. A run fails if there is a collision that throws any of the robots off the track so the plan cannot be finished. The average number of collisions per run shows how many collisions that did not ruin the plan occurred. These collisions can range from small one, where the robots only touched each other and did not affect the execution of the plan, to big collisions, where the agent was slightly delayed in their individual plan, but still managed to finish the plan. For the *classic* model, where no execution finished, we present the number of collisions occurring before the major collision that stopped the plan.

Since we are using the makespan objective function, all of the plans can have their length equal to the longest plan without worsening the objective function. Even if the agents reached their destinations sooner, their plans were prolonged by waiting actions to match the length of the longest plan. To visually observe this behavior, we used the LEDs on the robots. The LEDs were turned on during the execution of the whole plan (including wait actions) and they were turned off once the plan was finished. This helped us to measure the overall time of the plan execution as the time from start to the last robot turning LEDs off. For the *classic* model, there is no total time, since the agents did not finish at all.

Each individual agent was let to execute the plan without interference with other agents to measure the difference between the fastest and slowest agent as Max Δ time. If the agents are perfectly synchronized then this Δ should be zero. All of the times are rounded to seconds because the measurements were conducted by hand.

From the number of collisions and times, we can conclude some properties of the models. Indeed, models *classic + wait*, *w-split*, and *rw-split* keep the agents synchronous, while the other models do not (there is a gap between finishing the plans by different agents). From all models, the *classic + wait* model is the slowest one to perform the plan. This is expected as this model uses longest durations of actions. Further, we can see that even if the agents are synchronous, some collisions may appear, since the agents have a nonzero diameter and are moving close to each other. This issue is solved by making a k-robust plan, however, the simple *1-robust* model was not synchronous and this desynchronization could cause a collision eventually if the plan was long enough.

In general, the *split* models provide the fastest execution of plans. This is expected because these models optimize the makespan that is closer to the real makespan. From the results, we can also see that introduction of robustness and weighted edges to the classical MAPF is of practical use if we plan to use the computed plan for real robots.

6 Conclusion

In this paper, we studied the behavior of MAPF plans when executed on real robots. We defined several models that either take the classical plan and translate it into a sequence of robot actions or create a plan that already consists of the robot actions. This mainly included the need for turning of the robot.

In the experiments, we concluded that the classical plan produced by MAPF solvers is not suitable to be performed on robots. The introduction of splitting the position of the robot to include orientation proved to be useful as well as using weighted edges to correspond with travel time. Furthermore, introducing k-robustness forbid the agents to travel close to each other to prevent collisions.

Acknowledgement. Roman Barták is supported by the Czech Science Foundation under the project P202/12/G061 and together with Jiří Švancara by the Czech-Israeli Cooperative Scientific Research Project 8G15027. This research was also partially supported by SVV project number 260 453.

References

1. Atzmon, D., Felner, A., Stern, R., Wagner, G., Barták, R., Zhou, N.: k-robust multi-agent path finding. In: Fukunaga, A., Kishimoto, A. (eds.) Proceedings of the Tenth International Symposium on Combinatorial Search, 16–17 June 2017, Pittsburgh, Pennsylvania, USA, pp. 157–158. AAAI Press (2017). https://aaai.org/ocs/index.php/SOCS/SOCS17/paper/view/15797
2. Barták, R., Švancara, J., Vlk, M.: A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs. In: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, 11–13 July 2018, pp. 748–756. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2018). http://dl.acm.org/citation.cfm?id=3237383.3237494
3. Barták, R., Zhou, N.F., Stern, R., Boyarski, E., Surynek, P.: Modeling and solving the multi-agent pathfinding problem in picat. In: 29th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pp. 959–966. IEEE Computer Society (2017). https://doi.org/10.1109/ICTAI.2017.00147
4. Boyarski, E., et al.: ICBS: the improved conflict-based search algorithm for multi-agent pathfinding. In: Lelis, L., Stern, R. (eds.) Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, Ein Gedi, the Dead Sea, Israel, 11–13 June 2015, pp. 223–225. AAAI Press (2015). http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/10974
5. desJardins, M., Littman, M.L. (eds.): Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, Bellevue, Washington, USA, 14–18 July 2013. AAAI Press (2013). http://www.aaai.org/Library/AAAI/aaai13contents.php
6. Erdem, E., Kisa, D.G., Öztok, U., Schüller, P.: A general formal framework for pathfinding problems with multiple agents. In: desJardins, Littman [5]. http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6293
7. Evollve Inc., Ozobot & OzoBlockly: Welcome to OzoBlockly (2015). https://ozoblockly.com/

8. Kautz, H.A., Selman, B.: Planning as satisfiability. In: ECAI, pp. 359–363 (1992). https://dl.acm.org/citation.cfm?id=146725
9. Ozobot & Evollve Inc.: Ozobot—Robots to code, create, and connect with (2018). https://ozobot.com/
10. Ryan, M.R.K.: Exploiting subgraph structure in multi-robot path planning. J. Artif. Intell. Res. **31**, 497–542 (2008). https://doi.org/10.1613/jair.2408
11. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artif. Intell. **219**, 40–66 (2015). https://doi.org/10.1016/j.artint.2014.11.006
12. Sharon, G., Stern, R., Goldenberg, M., Felner, A.: The increasing cost tree search for optimal multi-agent pathfinding. Artif. Intell. **195**, 470–495 (2013). https://doi.org/10.1016/j.artint.2012.11.006
13. Standley, T.S.: Finding optimal solutions to cooperative pathfinding problems. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010. AAAI Press (2010). http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1926
14. Surynek, P.: On propositional encodings of cooperative path-finding. In: IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, 7–9 November 2012, pp. 524–531. IEEE Computer Society (2012). https://doi.org/10.1109/ICTAI.2012.77
15. Surynek, P.: Compact representations of cooperative path-finding as SAT based on matchings in bipartite graphs. In: 26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, 10–12 November 2014, pp. 875–882. IEEE Computer Society (2014). https://doi.org/10.1109/ICTAI.2014.134
16. Yu, J., LaValle, S.M.: Planning optimal paths for multiple robots on graphs. In: 2013 IEEE International Conference on Robotics and Automation, ICRA 2013, pp. 3612–3617, May 2013. https://doi.org/10.1109/ICRA.2013.6631084
17. Yu, J., LaValle, S.M.: Structure and intractability of optimal multi-robot path planning on graphs. In: desJardins, Littman [5]. http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6111