Independence Detection, Hierarchical Collaborative Algorithm and Alternate Paths for MAPF resolution

Alexandra Kathova and Andrés Córdova

Potsdam University

Abstract. Multi-Agent Pathfinding (MAPF) is the problem of assigning a route for several agents between their initial positions and its correspondent goals. The different approaches are a trade-off between optimality, completeness, and computational efficiency. The solutions created in this work are a combination of sub-optimal algorithms, OD+ID, HCA* and MAPP, that result in low computational cost, and a middle-optimal solution for a wide range of instances, even large ones.

Keywords: MAPF · OD+ID · HCA* · MAPP · Clingo · ASP.

1 Introduction

The aim of this project was to implement a solution which would solve MAPF problem by merging individual agents' plans. There are many variations and relaxations of the classical MAPF problem, and many different approaches are used to solve it [9]. The formalization of the MAPF problem and of the subset of it that we have worked with in this project is given in the Section 2.

During the course of the project, we've attempted two different approaches: (i) ID+HCA*, a combination of two algorithms described in [6] and [7], and (ii) Multi-Agent Path Planning (MAPP), as described in [10]. The detailed description of the implemented algorithms is given in the Section 3. Both algorithms work best with the warehouse-like instances. We decided not to focus on more complex instances since they are rarely seen in warehouse layouts.

To assess our solutions, we have run them on the diverse set of randomly generated instances. The comparison of the two solutions and the evaluation of the benchmarking results can be found in the Section 4.

We also compared our solution with the solutions of other groups participating in the same course. The results can be found in the Section 4.1.2.

2 Background

The Multi-Agent Pathfinding (MAPF) is an important problem in logistics. The goal of the pathfinding is to plan a route from starting position of an agent to its destination. When several agents are present in the same instance, the plans should be built in a way that prevents collisions [7].

There are three classes of MAPF centralized approaches, each one with sub-classes [5]. A brief taxonomy, useful for this project, can be presented as follows:

- **Reduction-based solvers**. These methods try to solve the problem by reducing it to some well-known problems with known solutions. They return the optimal solution, but unfortunately are only efficient for small problems. They are very costly in time and therefore not applicable for large instances. Prominent examples include reducing to Answer Set Programming (ASP) [2], as well the Conflict-Based Search (CBS) [5].
- **MAPF-specific sub-optimal solvers**. Highly efficient but not always gives optimal or even complete solutions.
  - Search-based sub-optimal solvers. They treat the agents in a different way. The best example of this case is Hierarchical Cooperative A* (HCA*), which look the best way for each of the agents in an ordered list, blocking the nodes used for an agent for the next agents of the list.
  - Rule-based sub-optimal solvers. There is determined behaviour for each scenario, like traffic rules (the one going straight have preference over the one turning, for example). That decrease the computational cost but results are very sub-optimal. Examples of this are TASS [1] and Push and Swap [3].
  - Hybrid solvers. These are a combination of the above, searching but also considering some rules for some scenarios.
- **Optimal MAPF solvers**. They search a global space combining the states of all agents. The operations are actions for the robot in such a way that they do not conflict.

The goal is solve the MAPF problem for instances like real warehouses where robots have to reach some shelves. In this sense, complex instances with artificial and intricate obstacles and shapes are avoided. Considering those constrains, there are two solutions proposed in this work, both belonging to the second class of solvers, particularly to the sub-class of Search-based sub-optimal solvers. The first approach combines the idea of Independence Detector [7] and HCA* [6], and the second follows the rules of Multi-Agent Path Planning [10].

3 Algorithms

In this section there are the 2 algorithms worked in this project, the ID+HCA*, and MAPP. Both search-based sub-optimal solvers.

3.1 ID+HCA*

The independence detection comes from the idea of Standley [7] [8] of Operator Decomposition, which instead of creating a wide group of large branches of cells from the starting point to the goal, it analizes the the possibilities that each cell has for the next move. For decrease the cost of this search, it first

select the agents which are not conflicting with others so the nodes used for them are not included in the search of the others. This method is Independence detection. The idea of HCA* is to add an heuristic to Cooperative A* (CA*) [6]. In short, the novelties of CA* were to add a wait move (in the encoding of this project a direction: $direction((0,0))$.) and a reservation table (or illegal table). The layer of the heuristic consist in prioritize the agents according to the distance between initial position and goal. With those two main ideas in mind, the method developed in this work is called ID+HCA*. ID+HCA* has as input all the single paths for each agent. Among them there are usually several conflicts. But not necessarily all of the agents are conflicted. Those paths who do not conflict, are called independent, and the rest (the dependent agents) are clustered. The cells used by the independent agents are included in a table of illegal nodes. In Clingo there are therefore a predicate called *illegal*. But that is sufficient only to avoid in the future node conflicts, avoiding that two agents share a cell at the same timestep. For that reason, a new predicate is added called $illegal\_from$ prohibiting an agent B in cell Y to come to cell X in time T where agent A is located in time $T-1$ but wanting to go to cell Y at time T. The code shows briefly how this is done in Clingo:

```
dependent(R) :- conflict(R,T).
independent(R) :- not dependent(R), robot(R).
illegal(R,(X,Y),T) :- position(robot(R),(X,Y),T),
                      independent(R), robot(R), time(T).
illegal_from(R,(X',Y'),(X,Y),T) :- position(robot(R),(X',Y'),T),
                      position(robot(R),(X,Y),T-1), independent(R).
```

Once the dependent robots have been identified and clustered, they are prioritized, being the agent with lowest cost the first to be processed. The cost is defined as the number of movements needed for each agent to reach the goal. A new encoding verify then for each agent if the given path is compatible with the illegal table and therefore arrive to the shelf position, as seen in code:

```
shelf(R,D) :- init(object(shelf,R),value(at,D)).
arrived(R,T) :- new_position(robot(R),C,T), shelf(R,C).
last_position(R,C,T) :- new_position(robot(R),C,T),
          not new_position(robot(R),_,T+1), not arrived(R,T).
```

If the agent can reach the goal without interfering with the illegal nodes, then its own used cells are written on the illegal table, and the the next agent is processed. Else, the path of the agent is truncated exactly one node before the conflict. A new path is then calculated for this agent, without colliding with the nodes on the illegal table. Them, reaching its goal, its new cells are also included in the illegal table. After completing all this process for all agents, all of them have reached their goal if they had a free cell at each time step. The algorithm is listed in 1.

Algorithm 1 ID+HCA* in pseudocode

```
Agents = [a_1, ..., a_n]
Path = [p_a1, ..., p_an]
C = GetDependents(Agents)
Illegal = {}
for a in Agents do
    if a not in C then
        Illegal ← p_a
    end if
    for c in C do
        if Path(c) does not conflict with Illegal then
            Illegal ← p_c
        else
            PWC_c = PartOfPathWithoutConflicts(p_c)
            Pos_c = GetLastPosition(c)
            NP_c = CreateNewPath(c, Pos_c)
            p_c = PWC_C + NP_c
            Illegal ← p_c
        end if
    end for
end for
```

3.2 MAPP

MAPP algorithm is incomplete for a general MAPF problem, but is proven to be complete on a subset of so-called slidable instances. We will explain this concept in subsection 3.2.2. To understand the notion of slidability, the reader should understand what an alternate path is. This concept is explained in subsection 3.2.1. The MAPP algorithm is explained in subsection 3.2.3. The details on our implementation are in subsection 3.2.4. MAPP algorithm requires some intial path $\pi(u)$ for each unit (also referred as robot) u. The collection of units U with their initial plans $\pi(u)$, and the graph representation of the map, represent an input instance for MAPP. The initial plans can be obtained by using Asprilo path generator [4] modified in such a way that there's no constraints regarding conflicts between robots. No replanning happens while running MAPP. Instead, the units proceed along their initial paths as long as possible. When a robot with a higher priority has a conflit with a robot with the lower priority, the unit with the lower priority (and maybe some other robots, for details see bring blank algorithm in section 3.2.3 is pushed away. Afterwards, all the pushed robots return to their initial plans and continue progressing.

3.2.1 Alternate paths If a robot is moving along its initial plan $\pi(u)$ from the position l_i^u to the position l_{i+1}^u, but the position l_{i+1}^u is occupied by another unit with lower priority, then MAPP tries to bring blank to the position l_{i+1}^u by pushing robots alongside the alternate path Ω_i, connecting locations l_{i-1}^u and

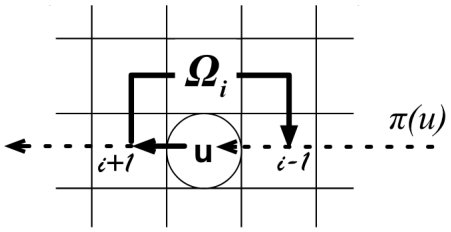


Fig. 1: Alternate path Ω_i for the triplet $(l^u_{i-1}, l^u_i, l^u_{i+1})$, which does not go through the node l^u_i. Image source: [10]

l^u_{i+1}. The important property of Ω_i is the fact that $l^u_i \notin \Omega_i$. Figure 1 represents such an alternate path.

3.2.2 Slidable unit and slidable instance An instance is called slidable, if all units in that instance are slidable. A slidable unit must fullfill the following criteria:

1. Alternate connectivity: there must be an alternate path for each triplet $(l^u_{i-1}$, l^u_i, $l^u_{i+1})$ of three consecutive cells on the initial paths of every unit u.
2. Initial blank: nor each $uinU$ l^u_1 is not occupied by any other robot.
3. Target isolation: no initial paths or alternate paths go through cells occupied by targets (shelves).

If all units in an instance are slidable, then the instance is also called slidable. MAPP algorithm is proven to be complete on all slidable instances. The detailed formal proof can be found in [10].

3.2.3 MAPP algorithm As an input, the MAPP algorithm receives an instance, which contains the grid, the initial positions of robots, the positions of shelves, and the initial paths for each robot starting from its initial position and ending at the position of the corresponding shelf. The upper-level algorithm is listed in Algorithm 2.

Here, inside the for loop we check if the instance is slidable. All three conditions described in subsection 3.2.2 can be verified in polynomial time. After that loop we can tell if the instance is solvable using basic MAPP algorithm. If the instance is solvable, then MAPP merger executes progression steps (and repositioning steps) until all units of the given instance are solved.

The algorithm for a single progression step in a procedural programming language paradigm is listed in Algorithm 3.

It's worth mentioning that in this algorithm a sequence of actions inside the while loop can be viewed as a procedure representing one substep of a given progression step. We will use this for the implementation in Clingo. Another important thing is the fact that each substep, in fact, consists of a set of rules.

Algorithm 2 High-level MAPP

for $u \in U$ **do**
  compute alternate paths from s_u to t_u
  if $SLIDABLE$ conditions hold **then**
    mark u as $SLIDABLE$
  end if
end for
if not all units are marked as $SLIDABLE$ **then**
  Return False
else
  do progression step
  do repositioning step if needed
end if

Algorithm 3 Progression step of MAPP

while changes occur **do**
  for each u **do**
    if pos(u) $\notin \pi(u)$ **then**
      wait {u has been pushed off his initial plan as a result of blank travel}
    else if u has already visited l^u_{i+1} in the current progression step **then**
      wait
    else if the next position of u belongs to the private zone of a higher priority unit **then**
      wait until l^u_{i+1} is released by v
    else if l^u_{i+1} is blank **then**
      move u to l^u_{i+1}
    else if can bring blank to l^u_{i+1} **then**
      bring blank to l^u_{i+1}
      move u to l^u_{i+1}
    else
      wait
    end if
  end for
end while

This is also very useful for translating the procedural algorithm into a logic program.

Another algorithm that needs to be explained is the bring blank algorithm which is used as a sub-procedure in the progression step algorithm. Bring blank algorithm is executed when the next position on the initial path of a robot is occupied by a robot with lower priority. Figure 2 demonstrates the steps of bring blank procedure. In the bring blank procedure, we are looking for a cell on the

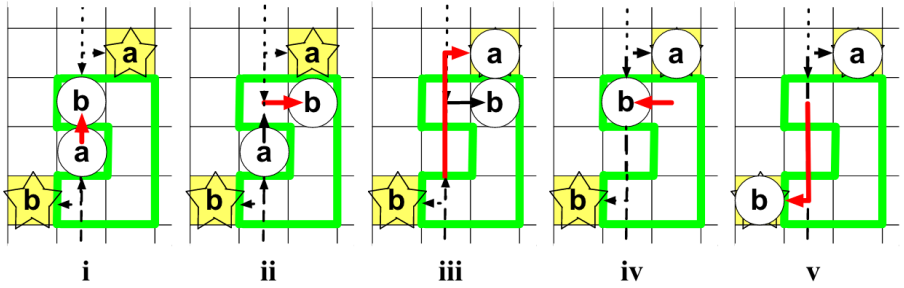


Fig. 2: Two progression steps of MAPP for a small instance containing two robots. Steps (i) - (iii) represent the first progression step with bring blank algorithm, step (iv) shows repositioning of robot b, and step (v) shows the second progression step. Image source: [**wangMAPPScalableMultiAgent2014**]

alternate path for the triplet consisting of previous, current and next positions of the robot. This cell has to fulfill the following criteria: (1) it has to be blank, (2) none of the cells between the current robot position and this cell (along the alternate path) belongs to a private zone of a robot with higher priority, and (3) this cell is the closest to the current robot position cell with this properties. If such a cell is found, then we slide all the robots between this cell and the current robot position along the alternate path. The bring blank procedure can be successfully executed for at least one robot - the robot with the highest priority - within each progression step, because the previous position is marked as private zone and can't hold a robot with a lower priority, i.e. it can't hold any robot. This means that during each progression step at least one robot will be able to arrive to its goal.

3.2.4 MAPP implementation in Clingo The high-level MAPP procedure and the iteration through progression steps is implemented in python because of the pure iterative structure. It was already mentioned in subsection 3.2.3, that each substep is in fact a set of rules controlling which of the robots can progress to the next step of the initial plan. Because of that, we have implemented a substep as a logic program in Clingo.

The explained above set of rules can be written (simplified to make it more intuitive for the reader) in Clingo language as a following logic program:

```
% get the possible next position
want(R,C,S) :- next_initial(R,C,C_prev), pos(R,C_prev,S-1).
% wait if the current position is not on the initial path
wait(R,S) :- pos(R,C,S-1), not initial_pos(R,C).
% wait if the next position belongs to a private zone of a
%robot with higher priority
wait(R,S) :- want(R,C,S), private(R1,C,S), r1_higher_R2(R1,R).
% wait if the next position already was visited
wait(R,S) :- want(R,C,S), visited(R,C).
% move if the next position is blank
move(R,C,S) :- want(R,C,S), not wait(R,S),
               #false: pos(R1,C,S), R != R1.
bring_blank(R,C,S) :- want(R,C,S), not move(R,C,S).
%....
% do bring blank which yields brought_blank(R,C,S)
% in case of success.
%....
% wait if could not bring blank
wait(R,S) :- bring_blank(R,C,S), not brought_blank(R,C,S).
% move if successfully brought blank
move(R,C,S) :- bring_blank(R,C,S), brought_blank(R,C,S).
```

Here, bring blank is omitted, but it is implied that in case of its success there will be a blank cell in front of the robot R.

4 Experimental Evaluation

There are two kind of experiments to evaluate the created algorithms. The first group of experiments is directed to observe the potency of the two algorithms, their evolution through the enlarging of size of the input instances and the quality of their results. Assessing this aspects, one algorithm can be selected for competing. The second group of experiments is among our algorithms and the ones of other groups.

4.1 Assessment of algorithms

For the first round of experiments, a series of 12 instances are created, all square grids of side size of 5, 8, 10 and 12. For each size s there are 3 instances with density $\frac{1}{s}$. All of those are slidable, assuring that both algorithms can solve them.

4.1.1 Internal Comparison The metrics to be evaluated are computational cost, makespan and sum of costs. Makespan here is defined as the length of the longest individual plan after the merging, and sum of costs is the sum of all movements done by all robots. The results are summarized in the Table 1 and Figure 4c. The density of each instance is given for the number of nodes occupied by shelves. The time is in seconds. From the Figure 4c it is clear that the ID+HCA* method is much faster computing a solution than MAPP. We assume that the main reason for the poor performance of MAPP is the implementation, which was not very straightforward to do in Clingo. The MAPP algorithm has iterative nature, and every progression step is purely deterministic, which led to the rapidly increasing size of the ground instance. This behaviour definitely can be somewhat improved via code optimization. On the other hand, MAPP always results in slightly lower sum of costs. Using the time steps as decision criteria does not help since the they are similar for both methods. However, ID+HCA* is selected as the method to use in the comparison with other groups, because the computational cost of MAPP could be a problem for larger instances and, more important, can not compete for non-slidable instances, while the former is more versatile.

Table 1: Results of the internal comparison.

Instances		ID+HCA*			MAPP		
Size	Density	Time	Makespan	Sum of costs	Time	Makespan	Sum of costs
5	0.20	0.54	8.00	22.00	16.83	6.67	17.00
8	0.13	1.06	12.33	53.00	24.75	16.00	44.00
10	0.10	1.79	13.00	82.67	28.31	13.33	79.33
12	0.08	1.95	10.67	57.33	24.76	12.67	54.33

ID+HCA* was afterwards tried in non-slidable instances, showing its efficiency for large and somehow dense instances. In the image 3 is an instance with 24 robots, with density of 16.7%, which was solved in 17.63 seconds. That one was one of the 2 instances offered for benchmarking to other groups. In the table 2 of next section, benchmarking with other groups, the result of ID+HCA* for the two own instances are in the rows 9 and 10.

The results are satisfactory, because the method developed in this project can solve relatively big instances in a very short time, and the sub-optimality is difficult to see, for each agents it takes in average only 10 moves to reach their goal, even with 144 or 225 nodes of freedom.

4.1.2 Comparison with other groups Based on the results of benchmarking, described in the Section 4.1.1, we decided to compete only using the ID+HCA* algorithm.

Though ID+HCA* has shown a decent performance and completion on the set of instances generated by our group, the results on instances provided by

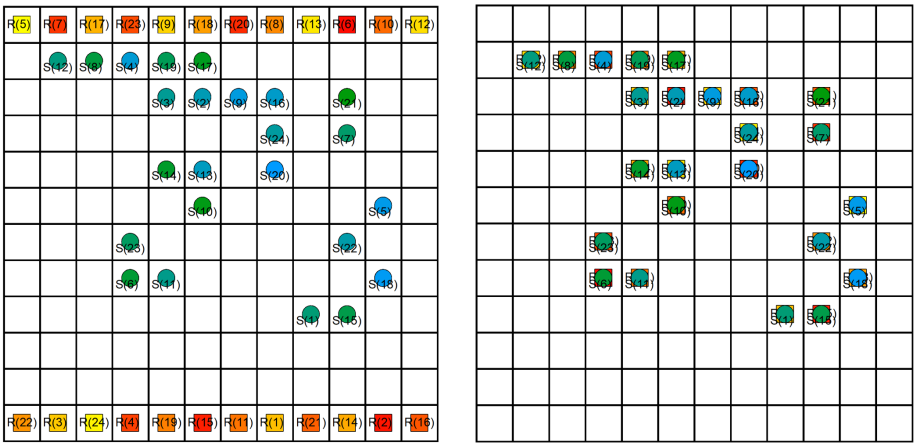
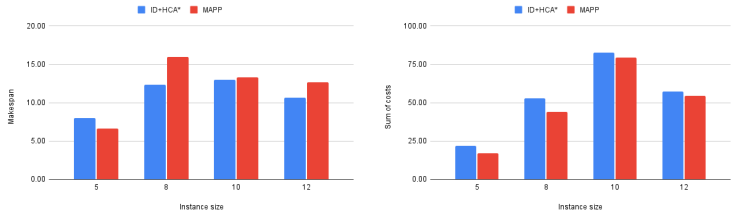
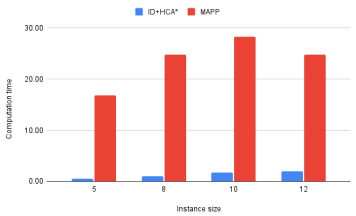


Fig. 3: Instance of 12x12 with 24 agents solved by ID+HCA*.



(a) Makespan (b) Sum of costs



(c) Computation time

Fig. 4: Benchmarking results for MAPP and ID+HCA*.

other groups are poor. The results are summarized in the Table 2. The success column shows the percentage of agents for which ID+HCA* was able to build a new plan without any conflicts with other agents. No instances were completed at 100%, though the vast majority has more than 50% completion. On the more positive side, there were no timeouts, and the longest computation time to obtain the result was 23.59 s for the instance Sauer1. These results are of no surprise, because our solution never aimed to be complete, whereas most of the mergers

Table 2: Benchmarking results of ID+HCA* on the instances provided by other groups

Instances				ID+HCA*				
Bench	Name	Agents	Size	Time[s]	Makespan	Sum of costs	Avg Steps	Success
1	mini_maze	4	10x10	0.36	21	71	17.75	0.50
2	warehouse	4	10x10	4.51	17	51	12.75	0.25
3	mix	26	15x15	7.49	13	166	6.38	0.40
4	small_room	3	7x7	0.76	10	26	8.67	0.66
5	14robs	14	7x7	2.64	10	61	4.36	0.79
6	ex8	19	7x7	3.20	10	95	5.00	0.74
7	Sauer1	9	11x11	23.59	11	37	4.11	0.56
8	Sauer2	7	2x10	2.20	5	28	4.00	0.86
9	24_dense	24	12x12	17.63	19	248	10.33	1
10	30_dense	30	15x15	35.85	21	309	10.3	1
11	x6_y11	16	6x11	21.07	11	50	3.13	0.56
12	x6_y11_2	32	6x11	45.55	3	51	1.59	0.81

implemented by other groups are modifications of CBS (constraint-based search) algorithm, which is complete [5].

Table 3: Benchmarking with the methods of other groups. 'A' = Agents, 'TO' = Timeout, 'x' = not solved.

Instances				ID+HCA*	bglaetzer	J-Behrens	Steven	Andreev
B	Name	A	Size	Time[s] (Succ)	Time[s]	Time[s] (Succ)	Time[s]	Time[s]
1	mini_maze	4	10x10	0.36 (0.50)	0.79	0.15 (0.50)	0.27	x
2	warehouse	4	10x10	4.51 (0.25)	0.75	0.73 (0.50)	0.18	x
3	mix	26	15x15	7.49 (0.40)	TO	TO (0.0)	2.48	x
4	small_room	3	7x7	0.76 (0.66)	x	x	0.95	x
5	14robs	14	7x7	2.64 (0.79)	TO	TO	0.52	8.17
6	ex8	19	7x7	3.20 (0.74)	TO	x	0.98	7.18
7	Sauer1	9	11x11	23.59 (0.56)	TO	0.14 (0.0)	3.94	x
8	Sauer2	7	2x10	2.20 (0.86)	x	0.10 (0.0)	0.29	x
9	24_dense	24	12x12	17.63 (1)	TO	TO	4.14	5.44
10	30_dense	30	15x15	35.85 (1)	TO	TO	6.81	7.18
11	x6_y11	16	6x11	21.07 (0.56)	TO	0.24 (0.0)	12.88	x
12	x6_y11_2	32	6x11	45.55 (0.81)	TO	0.55 (0.0)	0.49	x

Comparing the same instances with the projects of other groups which could run, it is clear that, even if ID+HCA* is not always the fastest, it gives always an answer for each instance, even if not complete. In table 3 only the solver of Steven could perform better than ID+HCA* in term of time, but unfortunately, there is no way to asses the completeness for this benchmarking, and probably is nearer to the solution of J-Behrens, which it returns no solution when computing less than one second.

5 Conclusion

This project present two solutions for MAPF. The first solution, ID+HCA*, shows that it is possibly to reach high computational efficiency without sacrificing much optimality, at least for not so dense instances. This advantage of time is also seen using the second solution, MAPP, when the optimal solution is more far away, but the solving is even faster when using a map with pre-calculated alternate paths.

We see that adapting algorithms to ASP is not always an easy task, since the reasoning is different for logic programming. The mixture between ASP and Python helps with this algorithmic way of thinking, resulting in a powerful final product, which can apply rule-based solutions sequentially for each agent.

We learnt also, that, as rich the literature is, is not necessary to create something completely new to contribute with a novelty. We took some strong ideas and adapt them to our particular problem, improving the previous results for some instances using only one approach.

References

[1] Mokhtar M Khorshid, Robert C Holte, and Nathan Sturtevant. "A Polynomial-Time Algorithm for Non-Optimal Multi-Agent Pathfinding". In: *International Symposium on Combinatorial Search* (2011), p. 8.

[2] Vladimir Lifschitz. "Answer Set Programming and Plan Generation". In: *Artificial Intelligence* 138 (2002), pp. 39–54. DOI: 10.1016/S0004-3702(02)00186-8.

[3] Ryan Luna and Kostas E. Bekris. "Efficient and Complete Centralized Multi-Robot Path Planning". In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems. Sept. 2011, pp. 3268–3275. DOI: 10.1109/IROS.2011.6095085.

[4] Van Nguyen et al. "Generalized Target Assignment and Path Finding Using Answer Set Programming". In: *IJCAI*. 2017. URL: https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/ijcai/NguyenOSS017.pdf (visited on 09/07/2022).

[5] Guni Sharon et al. "Conflict-Based Search for Optimal Multi-Agent Pathfinding". In: *Artificial Intelligence* 219 (Feb. 1, 2015), pp. 40–66. ISSN: 0004-3702. DOI: 10.1016/j.artint.2014.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0004370214001386 (visited on 05/22/2022).

[6] David Silver. "Cooperative Pathfinding". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 1.1 (1 2005), pp. 117–122. ISSN: 2334-0924. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/18726 (visited on 07/08/2022).

[7] Trevor Standley. "Finding Optimal Solutions to Cooperative Pathfinding Problems". In: *AAAI Conference on Artificial Intelligence* (2010), p. 6.

[8] Trevor Standley and Richard Korf. "Complete Algorithms for Cooperative Pathfinding Problems". In: *International Joint Conference on Artificial Intelligence* (2011), p. 6.

[9] Roni Stern et al. "Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks". June 19, 2019. arXiv: 1906.08291 [cs]. URL: http://arxiv.org/abs/1906.08291 (visited on 05/18/2022).

[10] Ko-Hsin Cindy Wang and Adi Botea. "MAPP: A Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees". In: *Journal of Artificial Intelligence Research* 42 (Sept. 2011), pp. 55–90. DOI: 10.1613/jair.3370. arXiv: 1401.3905 [cs]. URL: http://arxiv.org/abs/1401.3905 (visited on 05/28/2022).