To Wait, or Not to Wait, That Is the Question

Jiří Švancara,[1] Etienne Tignon,[2] Roman Barták,[1] Torsten Schaub,[2,3] Philipp Wanko,[2,3] Roland Kaminski[2,3]

[1] Charles University, Prague, Czech Republic
[2] University of Potsdam, Potsdam, Germany
[3] Potassco Solutions, Potsdam, Germany
svancara@ktiml.mff.cuni.cz, tignon@uni-potsdam.de, bartak@ktiml.mff.cuni.cz, torsten@cs.uni-potsdam.de, wanko@uni-potsdam.de, kaminski@cs.uni-potsdam.de, tignon@uni-potsdam.de

Abstract

Multi-agent pathfinding is the task of navigating a set of agents in a shared environment in such a way that they do not collide with each other. Finding an optimal plan in terms of the plan's length is a computationally hard problem, therefore, one may want to sacrifice optimality for faster computation time. In this paper, we present our preliminary work on finding a valid solution using only a predefined path for each agent with the possibility of adding wait actions. This restriction makes some instances unsolvable, however, we show cases where this approach is guaranteed to find a solution and the solution is found fast.

Introduction

Multi-agent pathfinding (MAPF) is the problem of navigating a fixed set of mobile agents in a shared environment (map) from their initial locations to target locations without any collisions among the agents (Silver 2005). This problem has numerous practical applications in robotics, logistics, digital entertainment, automatic warehousing, and more, and it has attracted significant research focus from various research communities in recent years (Li et al. 2021, 2020; Surynek 2019; Nguyen et al. 2017; Barták and Svancara 2019).

Finding an optimal solution in terms of the length of the found plan, being it the total length of the plan (i.e. *makespan*) or sum of individual lengths (i.e. *sum of costs*), is NP-Hard problem (Ratner and Warmuth 1990; Yu and LaValle 2013; Surynek 2015). However, in practical situations, it might be more important to find a path for each agent fast while sacrificing optimality.

In this paper, we try to answer the question of what type of instances of MAPF can be solved by only allowing each agent to move forward on a predefined path with the possibility of adding extra waiting actions. This of course means that some problems that are solvable under the classical setting of MAPF are no longer solvable. However, for instances that are solvable, we present a polynomial-time solver that is able to find a collision-free solution fast. [This work is preliminary, there is still one type of instances that is solvable under the proposed setting, but our solver is unable to decide that. We will identify this case in later sections.]

Definitions

In this work, we follow the usual definitions for multi-agent pathfinding from (Stern et al. 2019).

Definition 1. *An* instance of MAPF *is a pair (G, A), where G is a graph representing a shared environment and A is a set of agents. Each* agent $a_i \in A$ *is represented by a pair $a_i = (s_i, g_i)$, where s_i represents the start location (sometimes also called the initial location) of agent a_i, and g_i represents the goal location of agent a_i. In both cases, location corresponds to a vertex in the input graph.*

Definition 2. *A* solution to MAPF *is a plan $\Pi = \bigcup_{a_i \in A} \pi_i$, where π_i is a sequence of vertices that navigate agent a_i from s_i to g_i without any collisions with other agents. We use notation $\pi_i(t) = v$, meaning that agent a_i is present at time t in vertex v.*

As can be seen from the definition of the solution, the time is assumed to be discretized and at each timestep, all of the agents move at once into a neighboring vertex or they wait in their current vertex. The definition of collision is again taken from (Stern et al. 2019) in Figure 1. In our work, we assume that *vertex* and *swapping* conflicts are forbidden, but the algorithms do not depend on that and all of the conflicts in Figure 1 can be forbidden. Furthermore, we assume the setting where agents remain present in the graph after reaching their goal location.

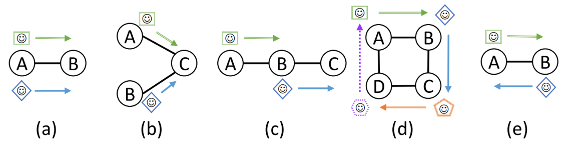


Figure 1: All possible MAPF conflict between two or more agents. From left to right *edge conflict*, *vertex conflict*, *following conflict*, *cycle conflict*, *swapping conflict*. Figure taken from (Stern et al. 2019).

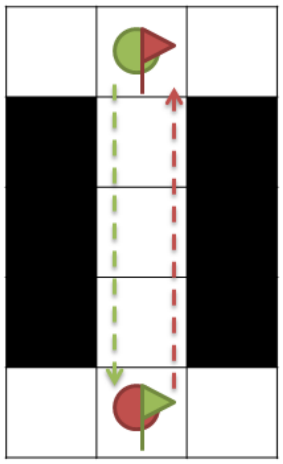
As opposed to the classical setting of MAPF above, we set an extra constraint on the movement of the agents. Assume that we are given a path $\mathcal{P}^i$ with non-repeating vertices in G from s_i to g_i, where $\mathcal{P}^i_k$ is the k-th vertex on that path. We restrict π_i such that $\pi_i(t) = \mathcal{P}^i_k \wedge \pi_i(t+1) = \mathcal{P}^i_{k+1}$

or $\pi_i(t) = \mathcal{P}^i_k \wedge \pi_i(t+1) = \mathcal{P}^i_k$. This means that at each timestep, each agent either moves forward on its predefined path or it waits. As a consequence, each agent can enter each vertex on $\mathcal{P}$ only once and wait there or move out. In our work, we assume that the predefined path is the shortest path from s_i to g_i (in case there are more different shortest paths, we choose one at random), therefore, we will use the notation $\mathcal{SP}$. However, it is not strictly necessary that the shortest path is used. We shall refer to this setting as SP+wait.

Problem Properties

Adding the extra restrictions on movement means that an instance that is solvable under the classical MAPF may have no solution. A simple example can be seen in Figure 2a[1]. In this example, there is no solution, as both agents are forced to move on their paths, and there is no room for them to avoid each other.

The example in Figure 2b has a solution both in classical MAPF and in SP+wait as the two agents can avoid each other by making one of them wait till the other moves to its goal.


(a) An example with no solution.

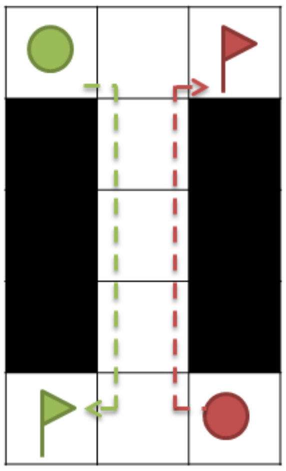
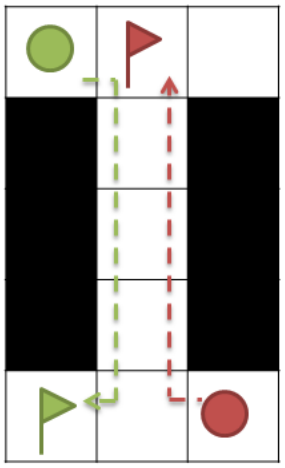

(b) An example with a solution, where one agent needs to wait.

Figure 2

The first idea on specifying when a solution exists is to forbid any intersection of $\mathcal{SP}^i$ with s_j or g_j, for $i \neq j$. This indeed produces instances that have a solution, but this restriction is too strict as can be seen in Figure 3a, where $\mathcal{SP}^{green}$ overlaps g_{red}, however, this example has a solution, where red agent waits for the green one to reach its goal.

Another idea is to consider only the interaction of pairs of agents. For example in Figure 3b, each pair of agents can be solved, but all three agents together produce an instance that has no solution (note that this instance has a solution in classical MAPF).

[1]In all of the examples, the dotted line with the corresponding color represents $\mathcal{SP}$ of that agent.


(a) An example with a solution, even though the goal of one agent intersects with $\mathcal{SP}$ of the other agent.

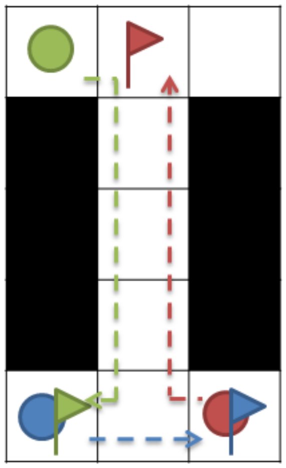

(b) An example with no solution when considering all 3 agents, but solvable with any pair of agents.

Figure 3

In the next two sections, we present our attempt at defining when an instance has a solution under SP+wait setting and, as a byproduct, we propose an algorithm to find plans for the agents.

The Wait-Graph Method

To overcome the problem presented in Figure 3b, we consider all agents and the interaction of their starting and goal locations. We define a waiting relation between two agents as follows.

Definition 3. *Agent a_i waits for agent a_j if:*

- *a_j's start s_j is on the shortest path $\mathcal{SP}^i$ of a_i, or*
- *a_i's goal g_i is on the shortest path $\mathcal{SP}^j$ of a_j*

We create a *waiting graph* $W = (V, E)$, where V represents agents and "a_i waiting for a_j" creates a directed edge (a_i, a_j).

Proposition 1. *If there are no cycles in the waiting graph W (i.e. W is a directed acyclic graph (DAG)) the instance can be solved by navigating one agent after each other on their $\mathcal{SP}$ while other agents wait. The order of the agents is given by the topological ordering of W.*

Proof. By induction, we take an agent a_i that is not waiting for anyone (such an agent must exist since W is DAG) and run it to its goal. Since a_i is not waiting for anyone, there is no other agent on its path and no other agent needs to go through a_i's goal. □

As a side result, based on Proposition 1, we can reason about the upper-bounds and lower-bounds of any solution to the MAPF problem under SP+wait.

Remark 2. *The lower-bound on the solution's makespan is the same as the lower-bound on the classical MAPF instance makespan (i.e. the maximum of the shortest paths).*

The upper-bound on the solution makespan is the sum of the shortest paths.

Proof. The lower-bound is trivial. The upper-bound follows from a solution found by Proposition 1. The longest solution is when the agents are moving one at a time giving the proposed upper-bound. If a solution would have a bigger makespan, at some point all of the agents are waiting which is unnecessary and such a step can be omitted. □

Remark 3. *The lower-bound on the solution's sum of costs is the same as the lower-bound on the classical MAPF instance's sum of costs (i.e. the sum of the shortest paths). The upper-bound on the solution's sum of costs is* $\sum_i |\mathcal{SP}^i| \times (|A| - i) \leq \max_i |\mathcal{SP}^i| \times |A|^2$.

Proof. The lower-bound is trivial. The upper-bound can be found in the same way as in Remark 2, only this time we need to count the cost of all of the waiting agents. The expression is maximized when the agents move in the descending order of the length of their respective $\mathcal{SP}$. □

Figures 4a – 4d show the wait graphs constructed based on the examples in Figures 2a–3b, respectively. As we can see, Figure 4a contains a cycle, since the two agents need to switch places, Figure 4b contains no edges as the two agents can go in any order (i.e. in terms of Proposition 1, there are 2 possible topological orders). On the other hand, Figure 4c has only a single topological order, which tells us in what order the agents need to move. Lastly, Figure 4d contains a cycle that could be broken if any of the agents were not present.

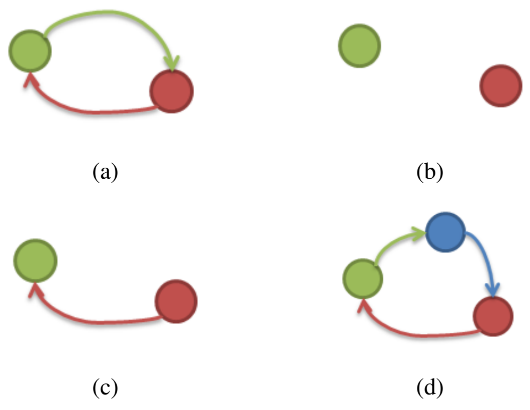

(a) (b) (c) (d)

Figure 4: Wait graph for examples in Figures 2a – 3b.

Following Proposition 1, if there is no cycle in the waiting graph, we are guaranteed that there is a solution under SP+wait. However, the reverse implication does not hold. Figure 5 shows an example where the agents form a cycle in the wait graph, yet it is easy to see that there exists a solution – for example, moving the red agent one step forward on its path leads to a position of the agents that does not create a cycle in the wait graph.

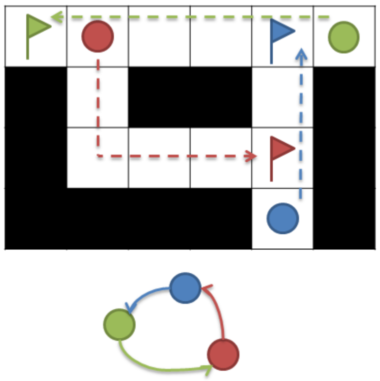


Figure 5: Example where the position of the agents creates a cycle in the wait graph, yet there is a solution under SP+wait.

As of right now, it remains an open problem how to deal with the cycles in the wait graph. Based on Proposition 1, we can show that if there is no solution under SP+wait, there has to be a cycle in the wait graph. On the other hand, we just showed that some cycles may be resolved. It is part of the ongoing research to find an easy method to decide when a cycle can be resolved. Furthermore, Figures 6a and 6b show examples where for two different instances, we get the exact same wait graph, yet one instance is unsolvable (the ones where the agents use the black paths) while the other has a solution (the ones where the agents use the paths corresponding to their colors).

Based on the previous examples, we can see that we need to look not only at the start and goal locations of the agent but also at the other vertices on the path. Unfortunately, simply checking if there is a vertex into which the agent may move, such that it does not obstruct any other agent while freeing the location that another agent is waiting for, is not enough. This approach would work for example in Figure 5 where any agent has such vertex, but we identified examples, where the only solution is to resolve a cycle by creating another cycle, which in turn may be resolved. Such complex interaction requires further research.

Related Work

We identified three related works that are similar to the SP+wait setting and the approach to solving it. We describe the differences between these works and our work.

Wait-for graph as detection of deadlocks in concurrent systems (Silberschatz, Galvin, and Gagne 2018). In this concept, a similar graph to our wait graph is built to represent agents waiting for resources used by other agents. If a cycle is detected, it means that there is a deadlock in the system. The difference to our work is that a cycle does not necessarily mean that there is a deadlock, or rather no solution in our case, as we showed in Figure 5.

MAPF as a scheduling problem (Barták, Svancara, and Vlk 2018). In this work, MAPF is modeled as a scheduling problem, where a layered graph is used (not the same layered graph as a time-expanded graph used in reduction-based MAPF solvers). If only a single layer is used, the

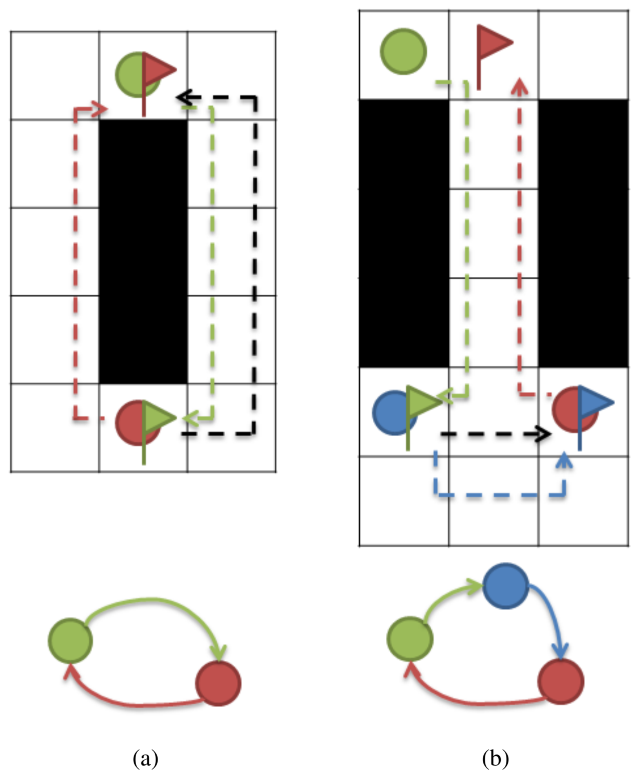


Figure 6: Examples, where the solvability depends on the paths the agents take (i.e. black or color), yet the wait graphs are the same.

agents are not allowed to return to a vertex and are allowed only to either move forward or wait. This is similar to our setting, however, in (Barták, Svancara, and Vlk 2018) the task is still to find a path for the agents, while we are asking if there is a solution given a path for each agent.

MAPF-POST (Hönig et al. 2016). In this work, the task is, for a given non-conflicting plan, to find an exact schedule of arrival to each vertex using STN (simple temporal network). The work deals with real robots, so the original non-conflicting plan is not assuming the motion constraints of the robot, hence they need to find a schedule. They start with a plan, we want to decide if there is one. An STN can decide this as well, however, STN needs ordering of visits of the agents in the vertices, and we do not know this in advance.

Experiments

We implemented the wait-graph method as was described in the previous section without resolving the cycles. This way we can report how many agents can be added to the MAPF instance before a cycle occurs. We used instances from the standard benchmark set (Stern et al. 2019) by starting with one agent and adding agents one by one until a cycle is detected in the wait graph. We report the average maximum number of agents before a cycle is detected in Table 1. We selected only a subset of maps provided in the benchmark set, however, we used both random and even distributions of the agents' start and goal locations. We can see that the number of agents highly depends on the size of the map (for example comparing the large game map *ost003d* with the small *empty-8* map) and its complexity (for example highly restricted big map *maze-128* reached similar numbers to much smaller open map *empty-32*). On the other hand, there is no significant difference between the random and even placement of the agents.

Map	even	random
empty-8	8,3	7,2
empty-16	10,5	10,4
empty-32	17,0	15,4
maze-128	13,4	13,9
ost003d	31,3	38,8
random-64	26,4	32,0
room-64	26,2	25,5
warehouse	21,4	22,2

Table 1: Average number of maximum agents before a cycle was detected in the wait graph.

As the algorithm is not yet complete, we do not report any other metrics. Each instance was computed in the order of milliseconds. So far, we also do not measure the length of the plan, as we let the agents travel their paths one by one. In the future version, a simple post-processing to parallelize the plan may be included.

The Extended Wait-Graph

To reduce the bridge between the wait-graph method and the solution to a SP+wait MAPF instance, we extended the concept of wait-graph to order not only the agents but every part of their shortest paths. This ordering can then be used to define a solution to the instance.

Definition 4. *Given an instance (G, A) and, for each agent $a_i \in A$, the given shortest path $\mathcal{SP}^i$, we define the* vertex occupations set $\mathcal{V}$. *For every vertex v in the shortest path of an agent a_i, $\mathcal{V}$ contains the tuple (a_i, v).*

Each tuple (a, v) in the vertex occupations set represents the presence of the agent a on the vertex v. We can now express every situation when an agent must wait for another, as well as other important constraints on the order of the occupations, as follows.

Definition 5. *Given an instance (G, A) and, for each agent $a_i \in A$, the given shortest path $\mathcal{SP}^i$, the waiting relations of the form "a_i on v waits for a_j on v'" are defined in the following situations:*

1. *"a_i on v' waits for a_i on v" if a vertex v is on the shortest path of a_i, and v' is the next vertex on the same path.*
2. *"a_i on v waits for a_j on v" or "a_j on v waits for a_i on v" if a vertex v is on the shortest paths of two agents a_i and a_j.*
3. *"a_i on s_j waits for a_j on s_j" if a_j's start s_j is on the shortest path of a_i.*

4. *"a_i on g_i waits for a_j on g_i" if a_i's goal g_i is on the shortest path of a_j.*
5. *"a_i on v waits for a_j on v" implies "a_i on v' waits for a_j on v'" if a vertex v is on the shortest paths of two agents a_i and a_j and, for both agents, the next vertex on their paths is v'.*
6. *"a_i on v waits for a_j on v" implies "a_i on v' waits for a_j on v'" if a vertex v is on the shortest paths of two agents a_i and a_j and the next vertex v' on the path of a_i is also the previous vertex on the path of a_j.*

Situation 1 maintains the ordering of vertices inside each path. Situations 2 and 6 address relationships between paths that could lead to conflicts (vertex and edge conflicts respectively). Situations 3 and 4 address the special cases when an agent starts or finishes its path on the path of another agent. Situation 5 expresses that one agent cannot pass another agent when they follow each other.

Definition 6. *An* ordering constraints set *is a set that contains an edge from (a_i, v) to (a_j, v') for each waiting relation of the form "a_i on v waits for a_j on v'".*

Note that because of the situation 2 of the definition 5, different ordering constraints sets may exist for the same instance.

Definition 7. *Given an instance (G, A) and, for each agent $a_i \in A$, the given shortest path $\mathcal{SP}^i$, an* extended waiting graph *$EW = (\mathcal{V}, E)$ is a graph composed of the vertex occupations set $\mathcal{V}$ and an ordering constraints set E.*

Proposition 4. *An instance (G, A) can be solved iff it has an extended waiting graph EW such that EW is a directed acyclic graph (DAG).*

Repair

From an acyclic extended waiting graph EW, we are able to define solution Π using the following property: The waiting relation "a_i on v waits for a_j on v'" implies that if $\pi_i(t) = v$ and $\pi_j(t') = v'$, then $t' < t$.

Proposition 5. Π *does not contain any collisions between agents.*

Proof. Using situation 2, we can affirm that of every π_i and π_j in Π, $\pi_i(t) = \pi_j(t')$ implies $t \neq t'$. Using situation 6, we can affirm that of every π_i and π_j in Π, $\pi_i(t) = \pi_j(t')$ implies $\pi_i(t+1) \neq \pi_j(t'-1)$. $\square$

Furthermore, getting a plan using the scheduling of a decentralized plan has already been addressed in the literature. For example, ASP with difference constraints is able to generate a plan by scheduling the path of different agents, as seen in (Abels et al. 2019).

Conclusion

In this paper, we presented our preliminary work on SP+wait setting of MAPF, where agents are able to move only on a predefined path with possibly added wait actions. We asked the question of how to decide when an instance has a solution under such setting.

To that extent, we proposed a wait-graph method that compares the predefined paths and starting and goal locations of all agents to create a wait graph. If the agents interact in such a way that there is no cycle in the wait graph, we are guaranteed to have a solution. On the other hand, some cycles can be also resolved. This decision process remains an open problem.

To improve the wait-graph method, we presented the extended wait-graph method, where all locations on the predefined paths are considered. Based on the paths, we get constraints on the ordering of agents' visits to the vertices, which are represented as edges in the extended wait graph. For some edges, we need to decide the orientation of the edge, meaning which agent visits the given vertex first. If we can find the orientation of the edges in such a way that the graph is without cycles, we can extrapolate a solution for the SP+wait MAPF instance. This approach remains to be implemented to empirically compare the results.

Acknowledgments

Research is supported by the Czech-Israeli Cooperative Scientific Research Project LTAIZ19014, and DFG grant SCHA 550/15, Germany.

References

Abels, D.; Jordi, J.; Ostrowski, M.; Schaub, T.; Toletti, A.; and Wanko, P. 2019. Train Scheduling with Hybrid ASP. In Balduccini, M.; Lierler, Y.; and Woltran, S., eds., *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, volume 11481 of *Lecture Notes in Computer Science*, 3–17. Springer.

Barták, R.; and Svancara, J. 2019. On SAT-Based Approaches for Multi-Agent Path Finding with the Sum-of-Costs Objective. In Surynek, P.; and Yeoh, W., eds., *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, 10–17. AAAI Press.

Barták, R.; Svancara, J.; and Vlk, M. 2018. A Scheduling-Based Approach to Multi-Agent Path Finding with Weighted and Capacitated Arcs. In André, E.; Koenig, S.; Dastani, M.; and Sukthankar, G., eds., *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2018, Stockholm, Sweden, July 10-15, 2018*, 748–756. International Foundation for Autonomous Agents and Multiagent Systems Richland, SC, USA / ACM.

Hönig, W.; Kumar, T. K. S.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Multi-Agent Path Finding with Kinematic Constraints. In Coles, A. J.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*, 477–485. AAAI Press.

Li, J.; Chen, Z.; Zheng, Y.; Chan, S.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2021. Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge.

In Biundo, S.; Do, M.; Goldman, R.; Katz, M.; Yang, Q.; and Zhuo, H. H., eds., *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, 477–485. AAAI Press.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. K. S.; and Koenig, S. 2020. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. In Seghrouchni, A. E. F.; Sukthankar, G.; An, B.; and Yorke-Smith, N., eds., *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems, AAMAS '20, Auckland, New Zealand, May 9-13, 2020*, 1898–1900. International Foundation for Autonomous Agents and Multiagent Systems.

Nguyen, V.; Obermeier, P.; Son, T. C.; Schaub, T.; and Yeoh, W. 2017. Generalized Target Assignment and Path Finding Using Answer Set Programming. In Sierra, C., ed., *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 1216–1223. ijcai.org.

Ratner, D.; and Warmuth, M. K. 1990. NxN Puzzle and Related Relocation Problem. *J. Symb. Comput.*, 10(2): 111–138.

Silberschatz, A.; Galvin, P. B.; and Gagne, G. 2018. *Operating System Concepts, 10th Edition*. Wiley. ISBN 978-1-118-06333-0.

Silver, D. 2005. Cooperative Pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In Surynek, P.; and Yeoh, W., eds., *Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, California, 16-17 July 2019*, 151–159. AAAI Press.

Surynek, P. 2015. On the Complexity of Optimal Parallel Cooperative Path-Finding. *Fundam. Inform.*, 137(4): 517–548.

Surynek, P. 2019. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. In Kraus, S., ed., *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 1177–1183. ijcai.org.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*.