Proceedings of the Twelfth International
Symposium on Combinatorial Search (SoCS 2019)

Multi-Train Path Finding

Dor Atzmon, Amit Diei, Daniel Rave
Ben-Gurion University, Israel
dorat@post.bgu.ac.il, diei@post.bgu.ac.il, ravedan@post.bgu.ac.il

Abstract

Multi-agent path finding (MAPF) is the problem of moving a set of agents from their individual start locations to their individual goal locations, without collisions. This problem has practical applications in video games, traffic control, robotics, and more. In MAPF we assume that agents occupy one location each time step. However, in real life some agents have different size or shape. Hence, a standard MAPF solution may be not suited in practice for some applications. In this paper, we describe a novel algorithm, based on the CBS algorithm, that finds a plan for moving a set of train-agents, i.e., agents that occupy a sequence of two or more locations, such as trains, buses, planes, or even snakes. We prove that our solution is optimal and show experimentally that indeed such a solution can be found. Finally, we explain how our solution can also apply to agents with any geometric shape.

1 Introduction

In the *Multi-Agent Path Finding* (MAPF) problem a plan is required for moving a set of agents from their start locations to their goal locations, without collisions. MAPF has practical applications in video games, traffic control, and robotics (see Felner et al. 2017 for a survey). Generally, a plan that minimizes some cost function is needed, e.g., the sum of the time required to get to the goal location over all agents (also known as the *sum-of-costs*). Even though finding such an optimal solution is NP-hard (Yu and LaValle 2013), some efficient algorithms manage to do so for more than a hundred agents (Wagner and Choset 2015; Boyarski et al. 2015; Surynek 2012; Felner et al. 2018).

Most MAPF algorithms assume that agents can occupy only a single location/cell/vertex at any single time step. However, in reality, agents normally can have different sizes and different shapes, and hence can occupy more than a single location per time step. In this paper, we investigate a new type of solution that is suitable for moving agents with the shape of a train, i.e., each agent occupies a sequence of locations. We define a problem, called: *Multi-Train Path Finding*, in which a plan is needed for moving a set of train-agents without collisions. A train-agent is an agent that each time step occupies its current location (by the head of the train) and the previous k locations on its path (by the tail of the train). Thus, even by waiting, it occupies $k+1$ locations each time. Note, that if $k = 0$ then the problem is similar to the standard MAPF problem. This problem is most suitable for cases where the agents have a long shape, e.g., trains, buses, planes, or even snakes, that can rotate and turn in different directions along their path.

Li et al. (2019) suggested a multi-agent path finding algorithm for large agents, and has shown that an optimal plan can be found for multiple large agents. However, it only fits agents with fixed shapes, e.g., circles or squares, that occupy the same locations after rotating them. XCBS (Thomas, Deodhare, and Murty 2015) is another MAPF algorithm in which agents occupy more than a single location each time step. This algorithm is designed for agents that function as convoys, and hence each agent occupies multiple consecutive edges. These convoys have been defined only to occupy edges and therefore the conflicts between agents were only on edges. Consequently, agents can cross the same location at the same time without causing collisions. Agents may also occupy more than one location in cases of uncertainty. In these cases, agents keep their distance from each other as their plan execution is inaccurate, and hence, they occupy multiple locations. In a k-robust plan (Atzmon et al. 2018), each agent can be delayed up to k times and no collision will occur. To avoid collisions that might occur from delays, agents cannot cross the same location in two closer than k times. Although each agent is at one location, no other agent can enter this location for the next k steps. This might seem similar to a case where each agent occupies $k+1$ locations (as a train); however, it is not the case if the agent waits in some location. Similarly, agents occupy multiple locations in MAPF under uncertainty (Wagner and Choset 2017), but not a fixed number of locations because agents can wait. A deeper exploration of the differences between this paper and the related researches is presented later in this paper, including adjusting our solution to any other type of agent.

Next, we define the problem and suggest an optimal algorithm for solving it. Experimentally, we show that finding an optimal solution to the problem can be done relatively fast. Then, we explain how to adjust the solution for agents with different shapes. Finally, we conclude this research and suggest directions for future work.

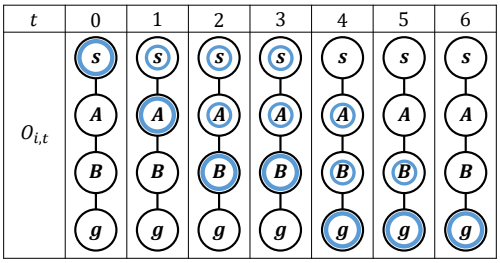

Figure 1: Agent occupation example.

2 Problem Definition

In the *Multi-Agent Path Finding* (MAPF) problem a plan π is required for moving a set of n agents. For each agent a_i, π contains a path π_i from its start location s_i to its goal location g_i. Each time an agent may either move to an adjacent location or wait in its current location. $\pi_i(t)$ denoted the location of agent a_i at time step t, hence $\pi_i(0) = s_i$ and $\pi_i(|\pi_i|) = g_i$. A conflict between a_i and a_j at time t can be either a vertex conflict ($\pi_i(t) = \pi_j(t)$) or an edge conflict ($\pi_i(t) = \pi_j(t+1) \wedge \pi_i(t+1) = \pi_j(t)$). A plan π is *valid* if there are no conflicts between all agents, and is *optimal* if it has the lowest cost, among all valid plans. There are a few commonly used cost functions. In this research we focused on a cost function called *sum-of-costs*, which is the sum of the path costs of all agents.

Agent occupation. While in standard MAPF each agent only occupies its current location, here each agent occupies a sequence of locations. Given a path π_i, we denote the list of locations that a_i occupies at time t by the list $O_i(t) = (l_1, \ldots, l_{k+1})$. $O_i(t)$ contains the location of agent a_i at time t (the head of the train) as well as its last k locations (the tail of the train). The start and goal locations represent stations. Hence, we define that agents start by occupying only their start location, growing to the size of $k+1$ as they leave their start locations, and shrinking back to the size of 1 as they enter their goal locations. Figure 1 shows the occupied locations of agent a_i at each time step for $k = 2$, while executing $\pi_i = (s, A, B, B, g)$. The locations occupied by the agent are represented by the blue circles on the figure, where the head is represented by a bigger circle. For instance, we can see that $O_i(2) = (B, A, s)$ and $O_i(\geq 6) = (g)$. Note that at times 2 and 3 the agent occupies the same locations as it performs a wait action. The cost of π_i is 4 as the agent arrives at g after performing 4 actions. We defined this occupation, in which the tail follows the head, and that all agents have the same k value (tail length) for simplicity. Later in this paper we show how $O_i(t)$ (this specific occupation) can be calculated and explain how it can be generalized for agents with different sizes and shapes.

Definition 1 (Self conflict) *A self conflict (a loop) $\langle a_i, l, t\rangle$ in a plan π occurs iff agent a_i occupies location l at time step t more than once (not by performing a wait action), i.e., $O_i(t)$ contains l more than once.*

Definition 2 (Occupation conflict) *An occupation conflict $\langle a_i, a_j, l, t\rangle$ in a plan π occurs iff agents a_i and a_j both occupy the same location l at time step t, i.e., both $O_i(t)$ and $O_j(t)$ contain l.*

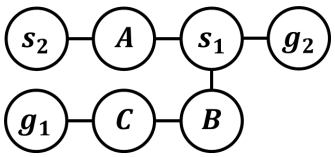

Figure 2: MTPF problem example.

Algorithm 1: *get-occupation* function

```
1  get-occupation(plan π_i, time t, size k)
2      Init empty list O
3      O.insert(π_i(t))
4      t ← t − 1
5      while (k ≥ O.size()) ∧ (t ≥ 0) do
6          if π_i(t) ≠ π_i(t + 1) then
7              O.insert(π_i(t))
8          else if StandingAtGoal(π_i, t) then
9              k ← k − 1
10         t ← t − 1
11     return O
```

Multi-Train Path Finding (MTPF) is a generalization of MAPF, in which each agent is a train-agent with a tail of k locations. Thus, MAPF is a special case of MTPF, where $k = 0$ (no tail). A plan π is a *multi-train valid solution* if there are no self conflicts and no occupation conflicts in π.

Example. Figure 2 presents an example of a MTPF problem with $k = 1$, in which a path is needed for agents a_1 and a_2, from s_1 and s_2 to g_1 and g_2, respectively. Assuming the path of agent a_1 is (s_1, B, C, g_1) and the path of agent a_2 is (s_2, A, s_1, g_2). There must be no self conflicts and no occupation conflicts so that the plan will be considered as a valid solution. By calculating O described earlier for both agents at each time step, we find that there are no conflicts. For instance, at time 2, $O_1(2) = (C, B)$ while $O_2(2) = (s_1, A)$, thus there is no conflict at time 2. Therefore, it is a valid solution. Assuming the path of agent a_1 is (s_1, B, B, C, g_1) and the path of agent a_2 is (s_2, A, s_1, g_2). In this case, $O_1(2) = (B, s_1)$ and $O_2(2) = (s_1, A)$. Both agents occupy location s_1 at time 2, and hence the plan contains an occupation conflict and is not a valid solution.

Calculating the occupation list. $O_i(t)$ can be calculated by the function *get-occupation* presented in Algorithm 1 as follows. First, we initialize an empty list O, insert the current location $\pi_i(t)$ into O (the train's head), and decrease t (lines 2-4). Then, we iterate as long as k is *greater than or equal to* the size of O as well as the current time t is *greater than or equal to* 0 (line 5). If the current location $\pi_i(t)$ does not equal to the next location $\pi_i(t+1)$, i.e, the agent preformed a move action, then insert the current location to O (the current location is occupied by the tail, lines 6-7). Otherwise, if the agent preformed a wait action and stands at the goal (and stays their), then we decrease k as the tail occupies one

less location (lines 8-9). We then decrease t and finish the iteration (line 10). Finally, we return the list O (line 11).

3 Conflict-Based Solution

In this section, we present a conflict-based solution for solving the MTPF problem.

Conflict-Based Search

Conflict-based search (CBS) (Sharon et al. 2015) is a state-of-the-art MAPF solver that has two levels (high-level and low-level). The high-level of CBS searches the binary *constraint tree* (CT). Each node $N \in CT$ contains: **(1)** $N.constraints$, a set of constraints imposed on the agents, where a *constraint* imposed on agent a_i is a tuple $\langle a_i, l, t\rangle$, meaning that agent a_i is prohibited from occupying location l at time t; **(2)** $N.\pi$, a single solution that is *consistent* with all constraints; and **(3)** $N.cost$, the cost of solution $N.\pi$, that is, the sum of the path costs of all agents. The root node contains an empty set of constraints. The high-level performs a best-first search on the CT, ordering the nodes by their costs.

Generating a node in the CT. Given a node N, the low-level of CBS finds a shortest path for each agent that satisfies all constraints in node N imposed on the agent.

Expanding a node in the CT. Once CBS has chosen node N for expansion, it checks the solution $N.\pi$ for conflicts. If it is conflict-free, then node N is a goal node and CBS returns its solution. Otherwise, CBS *splits* node N on one of the conflicts $\langle a_i, a_j, t\rangle$ ($\pi_i(t) = \pi_j(t) = l$) as follows. In any conflict-free solution, at most one of the conflicting agents a_i and a_j can occupy location l at time step t. Therefore, at least one of the constraints $\langle a_i, l, t\rangle$ or $\langle a_j, l, t\rangle$ must be satisfied. Consequently, CBS *splits* node N by generating two children of node N, each with a set of constraints that adds one of these two constraints to the set $N.constraints$.

MT-CBS

Multi-Train CBS (MT-CBS) is a CBS-based algorithm designed to return optimal MTPF solutions. MT-CBS differs from CBS in how the low-level calculates paths, and in how the high-level identifies and resolves conflicts.

Calculating paths. The low-level of CBS calculates a path for a given agent without violating any constraint from a given set of constraints. However, by using a standard low-level solver this path can be invalid, as a train-agent might have self conflicts. To overcome self conflicts, the low-level must be slightly modified. Before the low-level generates a new node, it performs a self conflict check. This can be done easily by calling the *get-occupation* function described in Algorithm 1. If O contains a location more than once, then a self conflict exists, and this node will not be generated.

Moreover, to ensure that the low-level returns an optimal path for a given agent, we cannot represent each low-level state only by the agent's current location. Two nodes with the same train's head can have a different sub-tree; thus, we represent each state by its current list of occupations. Additionally, representing nodes by a set of occupations instead of a list of occupations can also result in a non-optimal path.

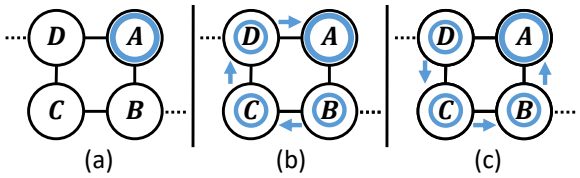

Figure 3: MT-CBS low-level example.

Figure 3 shows an example of a low-level search of MTPF with $k = 3$, in which the head of the agent is in location A. In Figure 3(a), each state represented only by the agent's head. Thus, it can either move to location D or to location B. However, if the agent arrived at A from D (as presented in Figure 3(b)) it cannot move to D as it results in a self conflict (similarly for moving to location B in Figure 3(c)). Therefore, we cannot represent states only by the head location. In addition, in both Figures 3(b) and 3(c), the agent occupies the same locations, for different paths. Although the same locations are being occupied, each results in different children (sub-trees). Hence, we represent states by a list of occupations and not by a set of them.

Identifying occupation conflicts. After the low-level has calculated a path for each agent for a CT node N, the high-level calls the *get-occupation* function described in Algorithm 1 for each agent, for each time. An occupation conflict is identified between a_i and a_j at time t, if for some location l, both $O_i(t)$ and $O_j(t)$ contain l. N is determined as a goal node iff no occupation conflicts were found in N.

Resolving occupation conflicts. Let N be a non-goal CT node selected to be expanded next, and let $C = \langle a_i, a_j, l, t\rangle$ be an occupation conflict in N. C occurred as both agents (a_i and a_j) occupy location l at time t (not necessarily by their heads). There is no valid solution in which both agents occupy l at time t. Therefore at least one the constraints $\langle a_i, l, t\rangle$ or $\langle a_j, l, t\rangle$ must be added to N. Note that in MTPF, a constraint $\langle a, l, t\rangle$ means that it is forbidden for agent a to occupy location l at time t, either by the head or by the tail. Consequently, we generate two children to N, each having one of these constraints.

MT-CBS is *sound*, as it halts only when expanding a CT node that has no occupation conflicts, and self conflicts are not being generated by the low-level. MT-CBS is *complete*, as it finds a solution if one exists, since splitting CT nodes never loses any valid solutions. MT-CBS is *optimal*, as it performs a best-first search on the CT (lowest cost first). Thus, expanding node N means that the cost of N is a lower bound on the cost of any other unexplored plan.

Example. Figure 4(a) presents an MTPF problem with $k = 2$. The shortest paths of agents a_1 and a_2 are $\pi_1 = (s_1, C, g_1)$ and $\pi_2 = (s_2, A, B, C, g_2)$, respectively. MT-CBS calculates these paths on the CT root node by the low-level (Figure 4(b)). Using *get-occupation* we get that $O_1(3) = (g_1, C)$ and $O_2(3) = (C, B, A)$. Thus, the agents collide by occupying C at time 3. Then, MT-CBS imposes the occupying constraints $\langle a_1, C, 3\rangle$ and $\langle a_2, C, 3\rangle$. After replaning for each agent, we expand the node with the lower cost (the right child) and get a solution with a cost of 7.

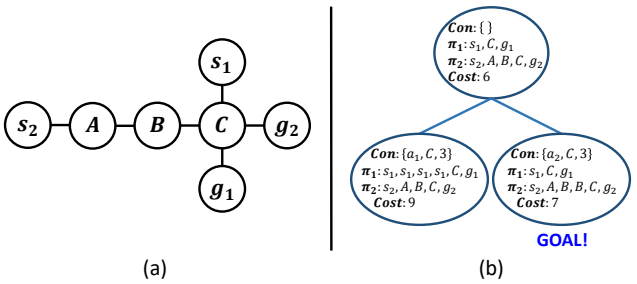

Figure 4: MT-CBS high-level example.

	Cost				Time (ms)			
#Agents	k=0	k=1	k=2	k=3	k=0	k=1	k=2	k=3
4	19.80	20.00	20.02	20.16	6	7	12	16
6	30.64	30.78	31.20	31.62	47	382	811	1,232
8	42.50	43.00	43.62	44.20	124	392	2,433	4,311
10	52.86	53.75	55.08	56.14	629	3,200	4,081	7,830

Table 1: MT-CBS on an 8x8 open grid.

4 Experimental Results

Next, we experiment MT-CBS on different grids, with a different number of agents, and different values of k.

Table 1 shows the average plan cost and planning runtime for 50 problem instances on an 8x8 grid with 4, 6, 8, and 10 randomly allocated agents, and $k = 0, 1, 2$, and 3. As expected, increasing the number of agents also increases the plan cost and the planning runtime. For example, for $k = 1$ the cost for 4 agents was 20.00 and the time was 7ms while for 10 agents the cost was 53.75 and the time was 3,200ms. For problems with more agents, the plan contains more paths and thus has a higher cost and it takes more time to find a conflict-free solution. Moreover, increasing k also results in higher cost and higher runtime. For 6 agents, for $k = 0$ the cost was 30.64 and the time was 47ms, and for $k = 3$ the cost increased to 31.62 and the time increased to 1,232ms. As k increases the agents occupy more locations and causes more conflicts. Therefore, greater k value increases the cost and required more time to plan.

We also experiment MT-CBS on a large map, called `brc202d`, from the Dragon Age Origins (DAO) video game, which is available in the `movingai` repository (Sturtevant 2012). We created 50 problem instances with 10, 20, and 30 randomly allocated agents, and $k = 0, 1, 2$, and 3. The average plan cost and runtime were measured and are presented in Table 2. As observed in the experiment of the smaller map, increasing the number of agents increases the cost and time. Also, to find a plan for greater k requires more time. The runtime for 20 agents was 395ms for $k = 0$ and 8,577ms for $k = 3$. However, this is hardly affected the cost of the plan. The average cost of both $k = 0$ and $k = 3$ for 20 agents were about 2,489. The larger the map, the less likely the agents will conflict. Therefore, the same plan for a lower k value or a plan with minor modifications may be conflict-free also for a higher k value.

	Cost				Time (ms)			
#Agents	k=0	k=1	k=2	k=3	k=0	k=1	k=2	k=3
10	1,294.8	1,294.8	1,294.8	1,294.8	25	59	110	164
20	2,489.0	2,489.1	2,489.1	2,489.1	395	1,013	3,681	8,577
30	3,851.3	3,851.3	3,848.8	3,848.9	1,010	1,907	5,220	17,159

Table 2: MT-CBS on the `brc202d` DAO map.

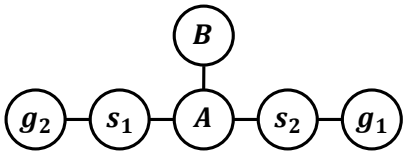

Figure 5: Difference between MTPF and k-robust example.

5 Different Types of Agents

As mentioned in the introduction, Li et al. (2019) addressed a MAPF problem that contains large agents that occupy more than one location each time step. Each agent was represented by a reference point. By using this point we could calculate the locations that this agent occupies. However, this can only be done for agents with specific shapes, and not for train-agents. Additionally, these agents cannot collide with themselves and cannot make turns, which needed a special consideration in this paper.

In k-robust plans (Atzmon et al. 2018) each agent can be delayed up to k times and no collision will occur. To avoid collisions that might occur from delays, agents cannot cross the same location in two closer than k times. This creates some kind of a train-agent occupation. However, by performing a wait action, the size of the occupation changes. Figure 5 describes this difference. A 1-robust plan exists, in which one of the agents waits in B and the other agent passes to the other side. In contrast, MTPF with $k = 1$ does not exist; while one of the agents waits in B it also occupies A. Moreover, in this example, a k-robust plan exists for any value of k.

In both papers above, the authors used range constraints to avoid collisions. These constraints were only imposed on the head of the train (or reference point) to prevent two agents from occupying the same location at the same time, which in some cases is similar to our solution. For example, imposing the range constraints $\langle a_1, C, [1, 3]\rangle$ and $\langle a_2, C, [1, 3]\rangle$ on the head of the agents in Figure 4 creates the same high-level tree. However, in some cases (as in Figure 5) the future tail is influenced by the current location of the head as well as by future actions (as wait actions), and hence, in contrast to the proposed solution, range constraints might not always work.

In fact, the proposed solution can be easily adjusted for agents with any size/shape/or inaccurate execution. The only modification that has to be made is creating for an agent a proper *get-occupation* function. Thereby, we can also apply this solution to problems in which each agent is different.

6 Conclusion and Future Work

In this paper we explored a new type of MAPF solution suited for long agents (train-agents). We defined an extension of MAPF for such agents, called: MTPF, and proposed

a CBS-based algorithm (MT-CBS) that finds optimal solutions to the extended problem. We have shown experimentally that it is possible to find such solutions, using MT-CBS, for both small and large domains. Possible directions of future work include compiling the problem into other known NP-hard problems (as done by Surynek et al. 2016) and adjusting the solution to agents with other shapes or sizes.

7 Acknowledgements

This work was supported by ISF grant 844/17, by BSF grant 2017692 and by NSF grant 1815660.

References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N. 2018. Robust multi-agent path finding. In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, 2–9.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Shimony, E.; Bezalel, O.; and Tolpin, D. 2015. Improved conflict-based search for optimal multi-agent path finding. In *IJCAI*.

Felner, A.; Stern, R.; Shimony, S. E.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N. R.; Wagner, G.; and Surynek, P. 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *the International Symposium on Combinatorial Search (SoCS)*, 29–37.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. K. S.; and Koenig, S. 2018. Adding heuristics to conflict-based search for multi-agent path finding. In *ICAPS*.

Li, J.; Surynek, P.; Felner, A.; Kumar, D. S.; and Koenig, S. 2019. Multi-agent path finding for large agents. In *AAAI 2019*.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.* 219:40–66.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *Computational Intelligence and AI in Games* 4(2):144–148.

Surynek, P.; Felner, A.; Stern, R.; and Boyarski, E. 2016. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*.

Surynek, P. 2012. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *Pacific Rim International Conference on Artificial Intelligence*, 564–576.

Thomas, S.; Deodhare, D.; and Murty, M. N. 2015. Extended conflict-based search for the convoy movement problem. *IEEE Intelligent Systems* 30:60–70.

Wagner, G., and Choset, H. 2015. Subdimensional expansion for multirobot path planning. *Artif. Intell.* 219:1–24.

Wagner, G., and Choset, H. 2017. Path planning for multiple agents under uncertainty. In *the International Conference on Automated Planning and Scheduling (ICAPS)*, 577–585.

Yu, J., and LaValle, S. M. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*.