Combining Strengths of Optimal Multi-Agent Path Finding Algorithms

Jiří Švancara and Roman Barták
Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic

Keywords: Multi-Agent Path Finding, SAT, Search Algorithm.

Abstract: The problem of multi-agent path finding (MAPF) is studied in this paper. Solving MAPF optimally is a computationally hard problem and many different optimal algorithms have been designed over the years. These algorithms have good runtimes for some problem instances, while performing badly for other instances. Interestingly, these hard instances are often different across the algorithms. This leads to an idea of combining the strengths of different algorithms in such a way that an input problem instance is split into disjoint subproblems and each subproblem is solved by appropriate algorithm resulting in faster computation than using either of the algorithms for the whole instance. By manual problem decomposition we will empirically show that the above idea is viable. We will also sketch a possible future work on automated problem decomposition.

1 INTRODUCTION

Multi-agent path finding (MAPF) is the task to navigate a set of homogeneous agents, located in a shared environment, from their initial locations to their desired goal locations in such a way that they do not collide with each other (Silver, 2005). An abstraction, where the shared environment is represented by a graph is often used (Ryan, 2008).

The MAPF task is strongly motivated in practice. The applications include traffic management, not only on roads but also air and sea traffic, warehouse management, robotics, and computer games. For a more in-depth analysis on the applications, we refer the reader to (Sharon et al., 2011).

Solving MAPF optimally in some manner (we will describe this more formally later) is NP-hard, and over the years, many optimal solvers were designed. In the given publications, pathological instances, where the solver is not performing well, are often identified. These instances can be often described by the size of the graph, the number of agents or initial and goal locations of the agents. We suspect that a randomly generated instance may contain some underlying structure (not necessarily the pathological instance) that negatively affects the solver performance.

In this paper, we selected two algorithms for which we identified the structure in instances that make them perform poorly. Based on this characterization, we suggest a way to exploit benefits of both solvers to solve general instances faster than either of the original algorithms. This is based on idea of splitting the problem instance into independent subproblems and solving each sub-problem by the best algorithm for it. We will show empirically that this idea is viable and we will suggest a method to obtain the decomposition automatically.

2 DEFINITIONS

An *instance of MAPF* can be formally defined as a pair (G,A), where $G=(V,E)$ is a graph representing the shared environment and A is a set of agents. Each agent $a_i \in A$ is then a pair $a=(a_0,a_+)$. a_0 represents the initial location (sometimes also called the start location) of agent a, and a_+ represents the goal location of agent a. In both cases, location corresponds to a vertex in the input graph.

Our task is to find a path for each agent that meets the following restrictions. The time is discretized and at each timestep, all agents move simultaneously. Each agent can either move to a neighboring vertex or not move (perform wait action). Let π_i denote a plan for agent i, where $\pi_i(j)=v$ represents that agent i at timestep j is present in vertex v. A valid *solution to MAPF problem* is a plan

$$\pi = \bigcup_{a_i \in A} \pi_i$$

satisfying the following conditions.

Švancara, J. and Barták, R.
Combining Strengths of Optimal Multi-Agent Path Finding Algorithms.
DOI: 10.5220/0007470002260231
In *Proceedings of the 11th International Conference on Agents and Artificial Intelligence (ICAART 2019)*, pages 226-231
ISBN: 978-989-758-350-6

1. Each agent is following a valid path: either $\pi_i(j) = \pi_i(j+1)$ or $(\pi_i(j), \pi_i(j+1)) \in E$.
2. No two agents occupy one vertex at the same time (vertex collision): for all pairs of different agents a_{i_1} and a_{i_2} at all time steps j, $\pi_{i_1}(j) \neq \pi_{i_2}(j)$ holds.
3. No two agents traverse the same edge at the same time (swap collision): for all pairs of different agents a_{i_1} and a_{i_2} at all time steps j it holds that $\pi_{i_1}(j) \neq \pi_{i_2}(j+1) \vee \pi_{i_1}(j+1) \neq \pi_{i_2}(j)$.

Note that these constraints allow agents to move on a fully occupied cycle in the graph, as long as the cycle consists of at least three vertices. There are other settings in regards to the allowed movements that are used. The first one requires the vertex an agent wants to enter to be empty before entering (sometimes called pebble motion) (Kornhauser et al., 1984) and the second allows agent to move into an occupied vertex, provided that the vertex will be empty by the time the agent arrives (this part is the same as the conditions for valid solution we are using in this paper), but forbidding movement on closed cycles (Surynek, 2010). Algorithms presented in this paper can be easily modified to work on either of those settings.

In addition to having a valid solution, it is often required to find a solution that is optimal (minimal) in some cost function. Two main functions used are *sum of cost* (Standley and Korf, 2011) and *makespan* (Surynek, 2014). Sum of cost adds the length of each agent's individual plan, while makespan is equal to the longest individual plan. It is worth noting that these functions are not equivalent and minimizing one or the other can result in different optimal solutions.

While there exist many polynomial time bounded solvers that find some solution (Kornhauser et al., 1984; de Wilde et al., 2014), to find an optimal solution in either makespan or sum of cost is NP-hard (Surynek, 2010; Yu and LaValle, 2013).

3 RELATED WORK

As indicated earlier, we picked two optimal MAPF algorithms for which we intend to combine their strengths. These two algorithms are CBS (Sharon et al., 2015), which is a representative of a search technique, and reduction of the MAPF problem to propositional satisfiability problem (SAT) (Surynek, 2014; Surynek et al., 2016; Barták et al., 2017), which is a representative of a reduction based approach. In this section, we shall describe them both in detail.

3.1 Reduction to SAT

The main idea of solving MAPF by reduction to SAT is to create propositional variables that describe the movement of each agent in the input graph in time. We define $\varphi^a_{v,t}$ to be *true* iff agent a is present in vertex v at time t and *false* otherwise. Now we can build constraints over these variables to ensure that the solution found by a SAT solver is indeed a valid solution to MAPF.

While it is not necessary, introducing similar variables for edges proves to create smaller and more efficient encoding of the problem (Surynek et al., 2016). Therefore, we define $\psi^a_{(v,u),t}$ to be *true* iff agent a is traversing edge (v,u) at time t and *false* otherwise. We also have to add loop edges for each vertex.

The formula that encodes the MAPF instance is then created by the following constraints for a fixed maximal time T:

1. Each agent a starts in its initial vertex v in timestep 0 and ends in its goal vertex u in timestep T.

$$\varphi^a_{v,0} = 1 \tag{1}$$

$$\varphi^a_{u,T} = 1 \tag{2}$$

2. Each agent a can occupy up to one vertex at a time.

$$\bigwedge_{\forall u,v \in V: u \neq v} \neg\varphi^a_{u,t} \vee \neg\varphi^a_{v,t} \tag{3}$$

3. If an agent a is present in vertex v, it must leave through exactly one connected edge. By adding the loop edge for each vertex, this also allows the agent to stay in the vertex.

$$\varphi^a_{v,t} \implies \bigvee_{(v,u) \in E} \psi^a_{(v,u),t} \tag{4}$$

$$\bigwedge_{\forall u,w \in V: u \neq w, (v,u),(v,w) \in E} \neg\psi^a_{(v,u),t} \vee \neg\psi^a_{(v,w),t} \tag{5}$$

4. If an agent a is using edge (v,u), it needs to enter the vertex u in the next timestep.

$$\psi^a_{(v,u),t} \implies \varphi^a_{u,t+1} \tag{6}$$

5. No two agents a_i, a_j can be present at the same vertex at the same time nor traverse the same edge at the same time.

$$\bigwedge_{\forall v \in V} \neg\varphi^{a_i}_{v,t} \vee \neg\varphi^{a_j}_{v,t} \tag{7}$$

$$\bigwedge_{\forall (v,u) \in E} \neg\psi^{a_i}_{(v,u),t} \vee \neg\psi^{a_j}_{(u,v),t} \tag{8}$$

Constraints 1 - 6 ensure that each agent follows correct path, while constraints 7 and 8 ensure that there are no vertex or swap collisions among the agents.

To find makespan optimal solution, we iteratively increase the maximal time limit T. The first T that produces a satisfiable formula is optimal makespan.

There is an encoding that produces also sum of cost optimal solution (Surynek et al., 2016), but its details are out of the scope of this paper. However, the variables and constraints described here are still used.

3.2 Conflict Based Search

Conflict Based Search is a two-level search algorithm that on top level searches a constraint tree containing a different set of constraints in each node; on low-level CBS finds a path for a single agent that is consistent with the given constraints.

A constraint is a triple (a, v, t) which states that agent a can not be present in vertex v in timestep t. We start the search with a root node of the constraint tree that contains no constraints. An optimal path for each agent is found and the cost of the solution is computed. All of these paths are examined and if there is a conflict we add new nodes to the constraint tree.

Let there be a collision between agents a_i and a_j in node v at time t. We want to avoid this conflict, which means that one of the agents can not use this node at that time. However, to ensure optimality, we need to investigate both options. We create two new nodes in the constraint tree as sons of the current node. Both of the two new nodes copy all of the parent's constraints and add one new constraint. One node adds (a_i, v, t) forbidding the conflicting location to agent a_i and similarly the second node adds (a_j, v, t). New single agent paths are found. Note that only path for the agent that has a new constraint needs to be found and other paths can be copied from the parent node.

The unexplored nodes in the conflict tree are ordered in ascending order by the cost of the solution. We continuously explore the node with the lowest cost until a solution with no conflict is found. This yields an optimal solution.

Note that the algorithm is not dependent on the cost function. It can be easily modified to compute either sum of cost or makespan. Both of these functions can be computed from the single agent paths.

Some improvements to this algorithm exist (Boyarski et al., 2015). The main improvement is that there may be many different optimal paths for single agent. Picking one may create conflict while picking other may not. In the improved version, it is first checked if there is some path that avoids current conflict. Another improvement is the ordering in which we examine the conflicts.

4 COMBINED APPROACH

As can be seen from the description of the algorithms, both are exponential in some value. In this section, we will identify the type of instances that are hard for each of the two algorithms respectively.

The satisfiability problem is exponential in the number of variables of the propositional formula. The reduction of MAPF to SAT produces one variable for each triplet (agent, vertex, time). If we increase the number of agents, the number of variables increases linearly. On the other hand, if we increase the size of the graph and assume that the starting and goal locations of the agents are randomly chosen, the length of the optimal path (path of an agent if no other agents are present in the graph) for each agent increases as well. This is important because the longest of the optimal paths is a lower bound for the makespan T. This means that increasing the graph size can make the problem harder than increasing the number of agents.

Of course, this does not hold true in every case since some formulas are harder for SAT solvers than others. However, it can be generally said that the reduction of MAPF problem to satisfiability problem is effective on smaller graphs, even with a high density of present agents.

On the other hand, CBS is exponential in the number of conflicts between agents in the planned paths. While it is fast (polynomial time) to find a path for a single agent that follows the constraints defined by the constraint tree, the binary constraint tree itself is exponentially big in the number of explored conflicts. This generally means that the fewer conflicts there are on the optimal path of each agent, the faster CBS is.

Increasing the graph size, while fixing the number of agents (again assuming that the starting and goal locations of the agents are randomly chosen), decreases the likelihood of conflict between the optimal path of each agent. Conversely, increasing the number of agents, while fixing the size of the graph, increases the probability of conflict. This means that CBS algorithm is more effective on graphs with a low density of present agents.

Combining these two observations, we see that there are instances where reduction to SAT outperforms CBS and vice versa. If we want to only choose the appropriate algorithm for the input instance, it is possible to just start both algorithms in parallel and see which one terminates and yields result faster.

However, we conjure that there are hidden structures in the input instance that can be separated and each can be solved by an appropriate algorithm. Indeed, assume that the set of agents A can be split into two disjoint sets A_1 and A_2, where any optimal solution for A_1 does not collide with any optimal solution for A_2. Furthermore A_1 forms dense instance, while A_2 forms sparse instance. Then it is more effective to solve for agents from A_1 by reduction to SAT and for agents from A_2 by CBS.

In general, the input instance can be split into many disjoint sets of agents and each set can be solved by a different algorithm. If the found optimal solutions are independent (i.e. there are no collisions between these solutions), we can merge them to form a valid optimal solution for the whole instance (Standley, 2010). This way, we are able to find the solution faster than by running either of the algorithms on the initial input instance.

The goal of this work is to validate the above idea by assuming that we are given the problem decomposition. The remaining challenge is how to identify such problem decoposition. We will provide some ideas on how to do this in the Future Work section.

5 PRELIMINARY RESULTS

5.1 Test Instances

To test our hypothesis about splitting an input instance into several subproblems, we created a set of instances. The instances share the same 4-connected grid graph pictured in Figure 1. We split the graph by hand into two disjoint section, each of which will fulfill one of the pathological instance described above.

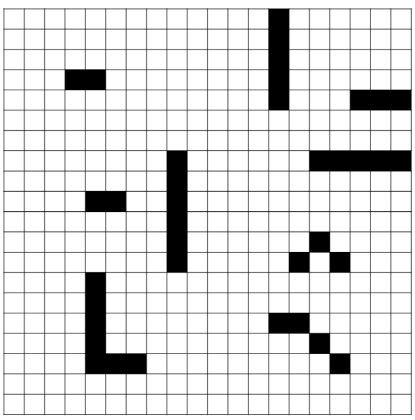


Figure 1: A grid map used in the experiments. Black vertices are impassable obstacles.

The section of 6 by 4 vertices in the top right corner shall play the role of small but dense part of the map (*Small*), while the rest of the grid shall play the role of large and sparse part of the map (*Large*). Note that the graph is connected, so the agents can move from one part to the other.

The number of agents in the *Large* part is fixed to be 10 and the number of agents in the *Small* part is increasing by one from 10 to 15. All of the agents have randomly chosen start and goal locations in their respective parts of the graph. Each of these settings was used to create 5 random instances. Together, this yields 30 instances.

We are also given the information about how to split the agents into the desired disjoint subsets. No automatic splitting is implemented at this time.

5.2 Measured Results

The two solvers used are CBS (Sharon et al., 2015) and reduction of MAPF to SAT via the Picat language and compiler (version 2.2#3) (Zhou et al., 2015).

Picat is a logic-based multi-paradigm language that integrates logic programming, functional programming, and constraint programming. Most importantly for our purposes, it contains a SAT compiler that translates constraints modeled by the Picat language into a logic formula in the conjunctive normal form (CNF) that is in turn solved by the built-in SAT solver. The main advantage of this approach is the simplicity of the code, while being comparable with the state of the art SAT-based MAPF solver (Barták et al., 2017).

The used cost function is sum of cost since this is the cost function already available in CBS implementation. However, it is easy to see that the experiments can be done with makespan cost function.

All experiments were conducted on a PC with an Intel® Core$^{\text{TM}}$ i7-2600K processor running at 3.40 GHz with 8 GB of RAM.

Given the two subsets of agents A_{Small}, A_{Large} and shared graph G, we let both of the solvers compute instance (G, A_{Small}) and (G, A_{Large}) with time limit of 600 seconds. Combination of these solutions is a solution to the initial instance (G, A). We measured the computation time for each of the subinstance, let name them $PicatSmall$, $PicatLarge$, $CBSSmall$, and $CBSLarge$.

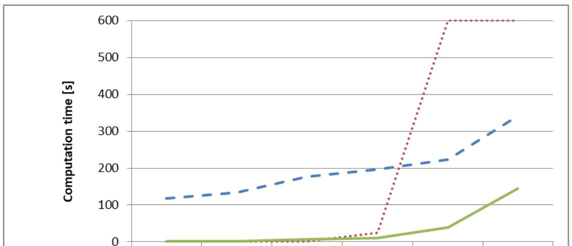
The times that we are interested in are

$$Picat = PicatSmall + PicatLarge \tag{9}$$

$$CBS = CBSSmall + CBSLarge \tag{10}$$

$$Combined = PicatSmall + CBSLarge \tag{11}$$

The averages of these times for each setting of the input instances are shown in Figure 2.



	10	11	12	13	14	15
Picat	118	134.8	176.8	197.2	222.9	341.3
CBS	0.2	0.2	1.9	25.2	600	600
Combined	2.3	2.5	6.5	10.2	38.7	145.4

Figure 2: Experimental results using translation to SAT (via Picat), CBS algorithm, and combination of both.

We can see that for the instances with fewer agents, the CBS solver is a clear winner. However, as the number of agents increase (and thus the *Small* part becomes denser), the CBS solver is not able to find a solution within the given time limit.

The computation time *PicatSmall* increases as well, but not as rapidly. We can see that the biggest portion of the computation time *Picat* is *PicatLarge*.

These results clearly support our theory that splitting the input instance and using different solvers for each part is beneficial in terms of total computation time.

6 FUTURE WORK

It is clear that an automatic way of splitting the agents into independent subsets is needed in order to create a useful algorithm. To do this, we propose to use Independence Detection (ID) algorithm (Standley, 2010). ID can be used to split an instance into independent subproblems such that combining their solutions yield a valid solution for the whole instance. Note that if the solutions for the independent subproblems are optimal then the combined solution for the whole instance is also optimal (Standley, 2010).

ID works as shown in Algorithm 1. We start by assigning each agent into a group and finding an optimal path for each group. This is a single agent path at the beginning. We then repeatedly check the solutions of all groups for conflicts. If there is no conflict, and each group has an optimal plan, these plans can be combined into an optimal plan for the whole instance.

Assume that there are two groups conflicting (G_1 and G_2). If we can replan one of the groups while avoiding conflict with the other, and keeping the same cost of the found plan, we will keep that solution (lines 9 – 14). If no such replanning is possible, we have to merge G_1 and G_2 into a single new group and

Algorithm 1: Independence Detection.

1: assign each agent to a group
2: compute plan for each group
3: **while** there is a conflict in plans **do**
4: $G_1, G_2 \leftarrow$ conflicting groups
5: **if** G_1 and G_2 conflicted before **then**
6: merge G_1 and G_2 into new group G
7: find plan for G
8: continue
9: **else if** can replan G_1 and avoid G_2 **then**
10: replan G_1 and avoid G_2
11: continue
12: **else if** can replan G_2 and avoid G_1 **then**
13: replan G_2 and avoid G_1
14: continue
15: **else**
16: merge G_1 and G_2 into new group G
17: find plan for G
18: **end if**
19: **end while**
20: solution $\leftarrow$ path of all groups combined

find a new optimal plan (line 16).

By changing the plan of one group, we can create a new conflict with some other group. This can lead to infinite cycles. To fix this problem, we have to keep track of groups that were already checked for conflicts. If we visit such a pair again, we are possibly in a cycle, therefore, we merge the groups immediately into a new group and find optimal paths for that group (line 5).

There exist some enhancements for ID algorithm (Standley, 2012). These enhancements include the way how to prioritize the replanning of the conflicting groups, what conflict to resolve first, and how to choose the initial path for singe-agent groups (if there are more possible paths).

Note that ID is independent of the solver used to compute plans for the groups. We can run both algorithms (CBS solver and Picat solver) simultaneously each time and use the result of the algorithm that terminates first. If it is not possible to run both algorithms in parallel, we can start with either one with given time limit. If no solution is found in that time, we run the other algorithm. Again if no solution is found, we increase the time limit and repeat.

We hypothesize that while the groups are small (and thus are not dense), CBS solver will outperform Picat solver. As the groups get bigger (indicating that there are many conflicts among the agents), Picat solver will outperform CBS solver.

7 CONCLUSION

In this paper, we studied the problem of multi-agent path finding (MAPF), namely, we suggested how to combine solvers with complementary strengths. We analyzed two state-of-the-art solvers, CBS and reduction-based Picat solver, used to solve MAPF optimally and for each of them, we identified what type of instances are hard for it. We tested this empirically and observed that dense instances are harder for CBS algorithm and instances on large graphs are hard for reduction based algorithms. By manual decomposition of an example problem, we showed that the proposed combination of solvers indeed improves runtime significantly. As a future (not-yet-implemented) work, we proposed how to automatically split instances into independent subproblems by using the Independence Detection algorithm.

ACKNOWLEDGEMENTS

This research is supported by the Czech Science Foundation under the project 19-02183S any by the SVV project number 260 453.

REFERENCES

Barták, R., Zhou, N., Stern, R., Boyarski, E., and Surynek, P. (2017). Modeling and solving the multi-agent pathfinding problem in picat. In *29th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2017, Boston, MA, USA, November 6-8, 2017*, pages 959–966.

Boyarski, E., Felner, A., Stern, R., Sharon, G., Betzalel, O., Tolpin, D., and Shimony, S. E. (2015). ICBS: the improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel.*, pages 223–225.

de Wilde, B., ter Mors, A., and Witteveen, C. (2014). Push and rotate: a complete multi-agent pathfinding algorithm. *J. Artif. Intell. Res.*, 51:443–492.

Kornhauser, D., Miller, G. L., and Spirakis, P. G. (1984). Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *25th Annual Symposium on Foundations of Computer Science, West Palm Beach, Florida, USA, 24-26 October 1984*, pages 241–250.

Ryan, M. R. K. (2008). Exploiting subgraph structure in multi-robot path planning. *J. Artif. Intell. Res.*, 31:497–542.

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

Sharon, G., Stern, R., Goldenberg, M., and Felner, A. (2011). The increasing cost tree search for optimal multi-agent pathfinding. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 662–667.

Silver, D. (2005). Cooperative pathfinding. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*, pages 117–122.

Standley, T. S. (2010). Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*.

Standley, T. S. (2012). Independence detection for multi-agent pathfinding problems. In *Multiagent Pathfinding, Papers from the 2012 AAAI Workshop, MAPF@AAAI 2012, Toronto, Ontario, Canada, July 22, 2012*.

Standley, T. S. and Korf, R. E. (2011). Complete algorithms for cooperative pathfinding problems. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 668–673.

Surynek, P. (2010). An optimization variant of multi-robot path planning is intractable. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*.

Surynek, P. (2014). 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, November 10-12, 2014*, pages 875–882.

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 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, pages 810–818.

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*.

Zhou, N., Kjellerstrand, H., and Fruhman, J. (2015). *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer.