A Multi-Resolution Algorithm to Solve Large Multi-Agent Path Finding Problems

Anonymous submission

Abstract

The adoption of automation solutions based on fleets of autonomously guided vehicles (AGVs) is a visible trend in industrial logistics. The problem of orchestrating the movements of each robotic agent in the fleet is referred to as Multi-Agent Path Finding (MAPF), and several solutions have been developed in recent years. In real-life scenarios, the application of such algorithms demands scalability (being able to deal with large-size warehouses) and efficiency (being able to quickly adapt to changes in the problems, e.g., new orders or changes in their priorities). The performance of such algorithms decreases in presence of conflicts (two agents occupying the same location at the same time) to the point they become unusable when the solution has to be found in a short time. In this work, we propose a new technique to solve conflicts between agents, exploiting a variant of the Large Neighborhood Search (LNS) to extract sub-problems that are then solved using optimal MAPF solvers. We carried out an experimental evaluation that demonstrates the benefits of this novel approach, considering a realistic number of AGVs operating in real-life environments.

Introduction

In recent years, robotic automation has been increasingly changing the workplace, especially in warehouses. In this context, the problem of coordinating multiple robots to complete a task in a short time becomes of central importance. In the literature, this problem is known as the multi-agent pathfinding problem, which amounts to finding the best feasible path for each of the k agents considered. As described in (Roni et al. 2019), the standard version of the problem considers k agents moving from an initial to a final position on a graph. The combination of the agents' paths is called the joint plan Π and it should move all the agents to their goal without conflicts while minimizing a collective cost function $f(\Pi)$. The two most common cost functions are the makespan and the sum of individual costs, later referred to as `MKS` and `SIC`, respectively. The former returns the length of the longest path $\pi_i \in \Pi$, whereas the latter returns the sum of the lengths.

A second variant of the problem is the MAPF problem with pick-up and delivery, in which an agent should move to an intermediate node to load up some goods before reaching its final destination (Minghua et al. 2019). Moreover, lifelong variants of the problem exist in which new tasks may arrive while the agents are moving (Ma et al. 2017). In this work, we are going to consider a modified version of the problem in which each agent has to reach some intermediate goals know a priori before heading towards its final position.

The standard problem and its variations have been proven to be NP-hard (Yu and LaValle 2013), hence the computation of an optimal solution is burdensome. Optimal algorithms return the best solution possible, but they may require too much time and space in large scenarios. Some examples of optimal approaches are:

i. Constraint-Based Search (CBS) (Sharon et al. 2015), which adds two or more constraints on the agents' movements every time there is a conflict in order to avoid it;

ii. Increasing Cost Tree Search (ICTS) (Sharon et al. 2013) that iteratively increases the cost of the agents' paths until a feasible solution is found; and

iii. any of solution based on constraint programming (Bartak et al. 2017; Barták and Švancara 2019; Saccon, Palopoli, and Roveri 2022), which enables the programmer to model the problem using mathematical constraints and then solve them to find the solution.

Other algorithms in the literature are sub-optimal ones that compromise the optimality of the solution and the time necessary to find it. It is possible to classify them into bounded sub-optimal algorithms, i.e. algorithms for which the computed solution is only partially skewed from the optimal solution, and unbounded ones, i.e., algorithms that privilege computation time over the quality of the solution. Some examples of bounded algorithms are: Extended ICTS (EICTS) (Walker, Sturtevant, and Felner 2018) and Bounded CBS (Barer et al. 2014). Although EICTS was thought for non-unit cost graphs, it can be generally applied as a bound sub-optimal algorithm since it considers an interval around the optimal solution and returns whenever it finds a feasible solution within said interval. The second one instead applies focal search (Pearl and Kim 1982) to guess which nodes to expand. As for unbounded algorithms, Greedy CBS (Barer et al. 2014) relaxes both the high-level search and the low-level search in order to expand those nodes that seem to be closer to the goal.

Anytime solvers are another important category that has recently emerged. Such solvers are especially important

in time-sensitive situations since they provide a first sub-optimal solution to the problem as quickly as possible, and then they enhance it in the remaining time available. A couple of examples of this technique can be found in (Li et al. 2021; Huang et al. 2022), in which a first solution is promptly returned and then a Large Neighborhood Search (LNS) is used to improve on the solution. (Li et al. 2021) proposes different ways of extracting the neighbourhood on which to focus the search, considering both subsets of agents and of nodes. Also, (Huang et al. 2022) shows how machine learning may be used to effectively select a subset of agents on which to focus in order to improve the solution.

While anytime solvers may appear to be the best compromise, they may provide a first solution which is too unreliable leading to a waste of execution time. In this paper, we propose a sub-optimal algorithm that solves conflicts locally, hence providing a solution quickly and reliably. We also present a new real-life challenging scenario in which we have carried out some tests, and finally, we provide a comparison of the developed algorithms in another real-life scenario already availablen (Saccon, Palopoli, and Roveri 2022).

This paper is organized as follows. First, we formally define the problem. Then we provide a description of the algorithms, followed by a description of the different real-life challenging scenarios we considered. Finally, we present and discuss the results of the experiments, we draw conclusions and outline future work.

Problem Definition

Our problem, a variant of the standard problem proposed in (Roni et al. 2019), can be defined as follows.

Definition 1 *Given an undirected graph $\mathcal{G} = (V, E)$, a set $\mathcal{A}$ of agents such that $|\mathcal{A}| = k$, a vector $S \in V^k$ such that $S[i]$ represents the initial position of agent i, a vector $G \in \{V^*, ..., V^*\}$, $|G| = k$ such that $G[i]$ consists of the positions the agent i should pass through in the order of increasing vector subscripting value (i.e., $G[i][1], G[i][2]...$), a vector $T \in V^k$ such that $T[i]$ represents the target destination of the i-th agent, and a cost function associated to the plan of each agent. A plan π_i for an agent $A_i \in \mathcal{A}$ is a series of actions $\pi_i = (m_1, ..., m_n)$ that guide the agent from its initial position $S[i]$ to its final destination $T[i]$ passing through $G[i][j]$ in the order dictated by $j = 1...|G[i]|$. Such a plan must be valid, i.e., the actions must be either waiting on the same node or moving along an edge $(v_k, v_l) \in E$. We consider time as discretized, i.e., each agent's move lasts the same amount of time, hence $\pi_i[t]$ corresponds to the action taken by agent i at time t, which will result in the agent being on a node $n \in V$. A joint plan $\Pi = \{\pi_1, ..., \pi_k\}$ is the set of all the single plans and is said to be feasible, or valid, if all the single plans in it are valid and if no conflict between two agents arises. The goal is to find a feasible joint plan Π such that each agent A_i starting from its initial position $S[i]$ passes through all the positions $G[i]$ and finishes in its final position $T[i]$, minimizing the cost function of the entire plan.*

We consider two cases of conflict defined in (Roni et al. 2019): vertex conflict, i.e., when two agents are on the same node at the same time t ($a_i, a_j : \pi_i[t] = \pi_j[t]$), or a swap conflict, i.e., when two agents use the same edge in opposite direction at the same time t ($a_i, a_j : \pi_i[t] = \pi_j[t+1] \wedge \pi_j[t] = \pi_i[t+1]$). Notice that by avoiding vertex conflicts and swap conflicts we are also indirectly avoiding edge conflicts since we assume an edge can be used only by one agent at a time.

In this work, we will consider the sum of individual costs (`SIC`) as the cost function, while other aggregate ones, such as the makespan, will be referred to in future works.

Algorithms Description

We implemented four different algorithms, namely, CBS, ICTS, Constraint Programming (CP) and our proposed approach ICR. ICTS and CBS use a two-level search: at the high level, they maintain a tree called Constraint Tree (CT) and Increase Cost Tree (ICT), respectively, while at the low level, they use a search over the graph.

As the name suggests, CBS works by adding constraints every time there is a conflict between two agents. For each new constraint, also a node in the CT is added. The nodes of the CT have a cost, which is the value returned by the cost function on the solution obtained by applying the constraints. The algorithm is optimal since the CT nodes are stored in a priority queue based on their cost, hence the first to have a feasible solution is also the optimal one. We implemented the low-level search for CBS using A* (Hart, Nilsson, and Raphael 1968), but we had to modify it in order to account for the constraints. In particular, there are some cases in which an agent in order to avoid a conflict has to step aside, i.e., move out of its optimal trajectory, and let another agent pass. To account for this, we had to allow A* to move over the same nodes multiple times.

ICTS uses the high-level search to check if there are conflicts in the solution returned by the low-level search: if there are, it creates k new nodes, each one increasing the cost of the path of one agent by 1. The low-level search instead uses A* and Multi-value Decision Diagrams (MDD) (Srinivasan et al. 1990) to compute the path of one agent with a certain cost C. ICTS is optimal since all the nodes at the same level of the ICT have the same cost, and being a breadth-first search used at the high-level, the first feasible solution is also the optimal one.

We implemented a constraint programming solution based on the work of (Bartak et al. 2017), using a three-dimensional boolean matrix (`X`) to keep track of where each agent is in each instant of time and hence to avoid vertex conflicts. Notice that, with constraint programming, the makespan of the solution $\mathcal{T} = \{1, ..., t_f\}$, i.e., the number of time-steps necessary to execute the longest path, must be known, otherwise it would not be possible to instantiate the position matrix. In (Barták and Švancara 2019) the authors improve on (Bartak et al. 2017) by adding constraints to avoid swap conflicts, using another three-dimensional boolean matrix `edge` to keep track of the usage of edges. In both works, the authors address the standard MAPF problem, hence they do not take into account the need for agents to move through a series of intermediate goals. To solve

this step, we added a new bi-dimensional matrix, called $\mathtt{gp}$, which stores the time stamps at which the agents reached the i-th goal. We had to add the following constraint to ensure that the agents would move through the goals:

$$\forall i \in \{1, ..., |\mathcal{A}|\}, \forall g \in G[i], \forall t \in \mathcal{T}, \\ \mathtt{X}[t][i][g] \Rightarrow \mathtt{gp}[i][g] = t$$

Moreover, the agents should take the goals in the order they were provided, hence:

$$\forall i \in \{1, ..., |\mathcal{A}|\}, \forall g \in 1, ..., |G[i]|, \ \mathtt{gp}[i][g] - 1 \leq \mathtt{gp}[i][g]$$

Then we had to make sure that an agent could reach the final position only if all the goals had been reached, hence we changed the final constraint as follows:

$$\forall i \in \{1, ..., |\mathcal{A}|\}, \\ X[t_f][E[i]][i] \Rightarrow \sum_{g \in G[i]} (\mathtt{gp}[i][g] \neq 0) = |G[i]|$$

Instead of using the logic programming language Picat, we decided to use the CPLEX solver (IBM 2023) in order to have better performance to solve this MILP problem.

The last algorithm we developed is a variant of the LNS approach. We called it Intersection Conflict Resolution (ICR) and the algorithm starts by solving the Single-Agent Path Finding (SAPF) problems first, i.e., by finding the optimal paths for each agent as if they were alone on the graph. Then for each conflict between agents, the algorithm:

i. extracts a piece of the graph corresponding to the conflict area;
ii. considers only the agents that are inside this sub-graph;
iii. solves the new MAPF problem on the sub-graph using constraint programming;
iv. merges the solution we obtained with the previous one.

If no solution could be computed within the sub-graph, then this is iteratively expanded until either a feasible solution is found, or the whole graph is considered, at which point the problem is directly solved using constraint programming.

The extraction of the sub-graph starts from the node of the conflict and expands iteratively to the neighbour in a breadth-first manner until it reaches an intersection, that is a node which has more than two neighbours (not counting itself), and then it adds also these neighbours to the sub-graph. The algorithm starts by considering only one intersection, but if the solution could not be computed within the sub-graph, the algorithm expands it to another intersection, and so on until a solution is actually found.

As for the agents that the algorithm should consider, we do not care for agents that are not traversing the sub-graph, as they are neither an active part of the conflict (i.e., are not one of the two agents causing the conflict), nor they could be part of the conflict in the future since they are not moving through that region. Moreover, those agents that exit the sub-graph before the time of conflict and do not enter back can be skipped, as they are not part of the conflict, neither active nor passive. In practice, we consider only those agents that are either an active part of the conflict or are passing in the region of the conflict. Once we know the agents, we need to correctly change their tasks: their initial position will be the node through which they enter the sub-graph, their end position will be the last node they pass through to exit, and finally, the new goals will be all the goals that were previously reached inside the sub-graph in that period of time.

Another clause for discarding a sub-graph is based on the new agents, indeed if two agents were to have the same final position, then they would generate a conflict when the local solution is computed. For this reason, the sub-graph can be immediately discarded and the number of considered intersections increased. In a similar way, if two agents enter the graph at the same time through the same node, then they should be discarded[1].

Once the sub-graph and agents have been correctly extracted, the constraint programming solver is called on the new MAPF problem. This solver had to be modified in order to account for the fact that agents may enter the sub-graph at different times. If this were to happen, then we cannot consider them as if they were starting at the same time in the new MAPF solver, otherwise, the solver may produce solutions that seem to be conflict-free, but are not, or vice-versa. The solution to this problem is to consider the first agent that enters the sub-graph as the initial time of the simulation and simulate waiting actions for all the agents that enter the sub-graph later.

If the constraint programming solver has not found a local solution, then the process is restarted by increasing the number of intersections by one. Instead, if the sub-solver has found a solution, then the algorithm has to merge the local solutions with the previous ones:

i. it keeps part of the path that led to the agent entering the sub-graph;
ii. it replaces all the nodes inside the sub-graph with the new solution;
iii. it keeps the remaining part of the old path that was not inside the conflict area.

Finally, we test again for conflicts, if there are, then we restart the whole process, if there are not, then we return the merged solution.

While this algorithm does not return optimal solutions, it allows for quickly fixing possible conflicts.

Tests

In order to carry out relevant tests, we considered a new real-world map composed of two buildings connected by a bridge, later referred to as "University". To test the scalability of the algorithms, we divided this environment into four scenarios, one for each building (with 472 and 657 nodes, respectively), one for the bridge with 54 nodes and one considering all the structures together for a total of 1183 nodes. We also tested the real warehouse initially presented in (Saccon, Palopoli, and Roveri 2022).

[1]It should be noted that two agents enter the sub-graph before the time at which the original conflict triggered the algorithm, hence this situation should not happen.

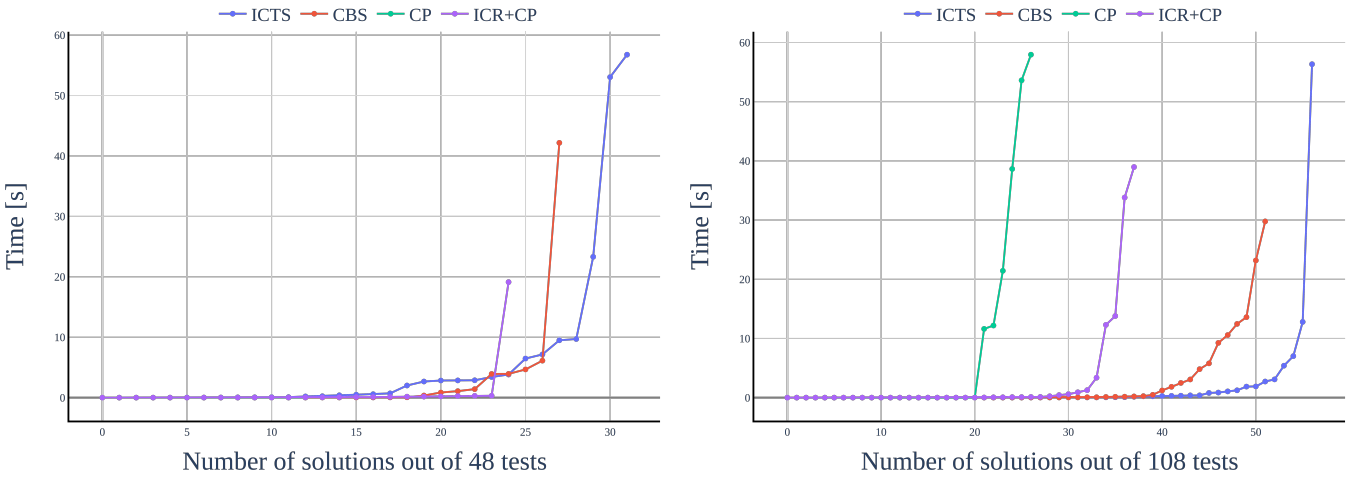

Figure 1: The survival plots show how the time to solve an increasing number of cases changes for the university scenario (left) and for the warehouse scenario (right), with a timeout of 60s.

	University				Warehouse			
Timeout[s]	**0.5**	**1**	**10**	**60**	**0.5**	**1**	**10**	**60**
ICTS	15	18	27	32	45	47	55	57
CBS	20	21	27	28	39	40	46	52
CP	13	13	13	13	21	21	21	27
ICR+CP	23	24	24	25	29	31	34	38

Table 1: The table containing the results for the university building (out of 48 tests) and for the warehouse (out of 108 tests). The best values are underlined.

For each scenario, we created random tests with an increasing number of agents $\{2, 4, 8\}$ and intermediate goals $\{0, 1, 5, 10\}$. The initial, the final positions and the goal points were randomly sampled from the set of nodes. We also made sure that not two consecutive goals were the same.

We then run the different tests with 4 timeouts $\{0.5s, 1s, 10s, 60s\}$ to understand how the algorithms behave during time.

Results and Discussion

We first focus on the University scenario. By starting from Table 1, it is possible to notice that ICR+CP solves a larger number of tests than ICTS and CBS, the real competitors, in the first two timeouts (0.5s, 1s), which are allegedly the most important ones when the system must compute the agents' paths on the fly. Within the 10s timeout, the algorithm is still competitive but starts to increase the required times as it is noticeable in Figure 1 on the left. It's interesting to notice that in this diagram, the constraint programming solver completely disappears: indeed, it solves too few tests to be noticeable.

Moving to the warehouse scenario, we can observe, both in Figure 1 and in Table 1, that ICTS and CBS provide better results than ICR+CP, solving not only more cases, but also in shorter times. This is due to the fact that i) the warehouse has many narrow aisles which are suitable to be used by only one agent at a time; ii) the possible intersection nodes where the conflicts may be resolved are only at the margins of the map. Instead, the university scenario has multiple lanes running along the corridors, hence it is easier to find an alternative route when two agents conflict.

One last remark should be done on the performance of ICR+CP. Given what has been said about the University scenario, we could expect the solver to solve many more cases than it does. The main reasons this does not happen can be found in how the constraint programming model works and in the different lengths of the paths. Indeed, as explained during the description of the algorithm, we need to know a priori the number of steps the solution should be found in. If the two agents in the conflict have paths much different in length, the number of steps to solve the problem increases, and as a result the constraint programming model becomes larger and takes longer to solve. For this reason, even providing more intersections may not lead to excellent results when using a constraint programming sub-solver.

Conclusions

In this study, we proposed a new approach to solving conflicts locally. While this provides sub-optimal solutions, the algorithm returns quickly and solving the conflicts locally allows not moving too much from the optimal path. Moreover, we propose a new and intriguing scenario where it's possible to test the performance of different algorithms.

In conclusion, we want to highlight how the algorithm[2] shown provides faster solutions when the timeout is shorter, especially in those cases where it is easy to find alternative routes in small spaces as in the University scenario. Yet it fails to outperform competitors due to the constraint programming model. For this reason, future work may focus on implementing sub-solvers using also other MAPF solvers and on exploring different neighborhood extraction processes as well.

[2]We will release the code upon acceptance.

References

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem.

Barták, R.; and Švancara, J. 2019. On SAT-based approaches for multi-agent path finding with the sum-of-costs objective. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, 10–17.

Bartak, R.; Zhou, N.-F.; Stern, R.; Boyarski, E.; and Surynek, P. 2017. Modeling and Solving the Multi-agent Pathfinding Problem in Picat. 959–966. IEEE. ISBN 978-1-5386-3876-7.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Huang, T.; Li, J.; Koenig, S.; and Dilkina, B. 2022. Anytime Multi-Agent Path Finding via Machine Learning-Guided Large Neighborhood Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36: 9368–9376.

IBM. 2023. ILOG CPLEX Optimization Studio. https://www.ibm.com/products/ilog-cplex-optimization-studio?mhsrc=ibmsearch_a&mhq=cplex. Accessed: 2023-03-07.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021. Anytime Multi-Agent Path Finding via Large Neighborhood Search. 4127–4135. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-9-6.

Ma, H.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks.

Minghua, L.; Hang, M.; Jiaoyang, L.; and Sven, K. 2019. Task and Path Planning for Multi-Agent Pickup and Delivery. *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*.

Pearl, J.; and Kim, J. H. 1982. Studies in Semi-Admissible Heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-4(4): 392–399.

Roni, S.; Nathan, S.; Ariel, F.; Sven, K.; Hang, M.; Thayne, W.; Jiaoyang, L.; Dor, A.; Liron, C.; Satish, K. T. K.; Eli, B.; and Roman, B. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *CoRR*, abs/1906.08291.

Saccon, E.; Palopoli, L.; and Roveri, M. 2022. Comparing Multi-Agent Path Finding Algorithms in a Real Industrial Scenario. In *AIxIA 2022 – Advances in Artificial Intelligence*, volume 13796. To be published.

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

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195: 470–495.

Srinivasan, A.; Ham, T.; Malik, S.; and Brayton, R. 1990. Algorithms for discrete function manipulation. In *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, 92–95.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2018. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. 534–540. International Joint Conferences on Artificial Intelligence Organization. ISBN 9780999241127.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. 1443–1449. AAAI Press.