Comparing Multi-Agent Path Finding Algorithms in a Real Industrial Scenario

Enrico Saccon[0000−0002−2418−6618], Luigi Palopoli[0000−0001−8813−8685], and Marco Roveri[0000−0001−9483−3940]⋆

University of Trento
{enrico.saccon,luigi.palopoli,marco.roveri}@unitn.it

Abstract. There is an increasing trend for automating warehouses and factories leveraging on teams of autonomous agents. The orchestration problem for a fleet of autonomous robotic cooperating agents has been tackled in the literature as Multi-Agent Path Finding (MAPF), for which several algorithms have been proposed. However, these algorithms have been only applied to synthetic randomly generated scenarios. The application in real scenarios demands scalability (being able to deal with realistic size warehouses) and efficiency (being able to quickly adapt to changes in the problems, e.g., new orders or change in their priorities).
In this work we perform an analysis of the MAPF literature, we selected the most effective algorithms, we implemented them and we carried out an experimental analysis on a real scalable warehouse of a large distribution company to evaluate their applicability in such scenarios.
The results show that a) no algorithm prevails on the others; b) there are difficult (realistic) cases out of the scope of all the algorithms.

1 Introduction

Robots are becoming a familiar presence in the daily life of people, helping them in different application domains: industry, warehouse, healthcare, search and rescue, and office automation. Despite this, industry is the domain in which automated machines have had the most successful applications. Indeed, the 4.0 industrial revolution meant for many workers an increased level of interaction with the machines present in the factory [4], with a significant impact on productivity [29]. Indeed, robotics proves to enhance and solve more easily logistics and manufacturing problems allowing for a better use of the industrial space [12]. Since the last decade, robots have been used with great profit in the healthcare sector. For example, they have been successfully used in precise surgical procedures to help surgeons reach difficult anatomical compartments and doing operations that would otherwise be impossible [5]. Also, robotics has been applied to help elderly and impaired people move more freely, besides being used to assist during rehabilitation [10]. Robots have been also successfully utilized is

⋆ The work of M. Roveri was partially funded by the Italian MUR programme PRIN 2020, Prot.20203FFYLK (RIPER – Resilient AI-Based Self-Programming and Strategic Reasoning).

search and rescue missions in challenging environments [1]. Finally, robots can be used to help in the day-to-day life of an office allowing affairs to be sped up and simplifying the general workday [27]. The majority of the above applications, involve multiple robots that need to cooperate while moving in a shared environment (e.g. a warehouse) without interfering with each other in order to complete one or multiple tasks in the most efficient way possible, and requiring prompt response to contingencies (e.g., arrival of a new task). This can be achieved by an automatic synthesis of a plan (i.e., a sequence of movements for each agent) to fulfill the full set of tasks. The automatic synthesis to be applied in real industrial scenarios requires that i) the solution plan will be generated for real-size industrial scenarios; ii) the solution plan will be generated quickly (e.g., in at most 1 minute) to quickly adapt to contingencies (E.g., new order, change of priority, order cancellation).

The problem of finding a plan for coordinating a fleet of autonomous robotic cooperating agents aiming to complete assigned tasks has been tackled in the literature as Multi-Agent Path Finding (MAPF). Several algorithms have been proposed to solve the MAPF problem like e.g., the Kornhauser's algorithm [13], the Extended A* [11], the Increasing Cost Tree Search (ICTS) [22], several variants of the Constraint Based Search (CBS) [21], and the Constraint Programming (CP) and Mixed Integer Linear Programming (MILP) approaches. However, as far as our knowledge is concerned, these algorithms have been only applied to synthetic randomly generated graphs, and their application in real scenarios has not been studied.

In this work we make the following contributions. First, we perform a detailed analysis of the MAPF literature, from which we selected the most effective algorithms, and we implemented them as efficiently as possible. Second, we carry out an experimental analysis on a real warehouse of a large distribution company. To evaluate the performance and applicability of the considered algorithms we decomposed the whole warehouse in sub-areas of increasing size (from a smaller area to the whole warehouse). For each scenario we considered different number of agents and several randomly generated tasks for each agent. The results show that the CP approach is able to solve very small cases and does not scale to large real-size scenarios, although it generates optimal solutions and is able to solve also small critical hard problems. The algorithms that performs better are the two variants of the CBS, although none of them is able to solve many cases. This work also contributed to identify some situations for which none of the considered algorithms is able to find a solution in a set amount of time. These results contribute to pave the way for investigating new heuristics to solve these hard problems that appear in real scenarios.

This paper is organized as follows. In Section 2, we revise the literature about MAPF. In Section 3, we formally define the problem we aim to, and we provide an high-level description of the most relevant approaches studied in the literature. In Section 4, we describe the most relevant implementation details, the considered warehouse, and we critically discuss the results. Finally, in Section 5, we draw the conclusions and we outline future works.

2 Related Works

In this work, we focus on the aspect of motion planning considering the equally important problem of mission planning as completed before starting the motion planning task. While the former focuses on the best path to follow starting from a position, executing the intermediate objectives and reaching the final destination [14], the latter focuses on the best way of organizing the goals for each robots in the environment [6]. The reason why mission planning is not considered is due to the fact that usually warehouses use specialized software to handle their internal structures, and such software is usually responsible for the generation of an ordered set of goals. The aspect of motion planning is particularly important in a populated environment because it needs to guarantee people safety.

2.1 Single-Agent Path Finding

The Single-Agent Path Finding (SAPF) problem is the problem of finding the best path on a graph between two given nodes or vertexes. Such problem is of great importance in various scenarios. Indeed, one of the main algorithms used to solve the SAPF problem, A*, has been successfully applied to GPS localization in order to improve the way-points accuracy for remote controlled agents [15]. Nevertheless, the field in which single-agent path finding has found the most importance is the field of robot routing and planning, as the problem name also suggests. SAPF algorithms have been successfully implemented in robot routing, where they have been used to search a graph constructed by environmental data in order to avoid obstacles and to explore possible routes [2].
This thesis focuses on the path planning problem that can be defined as follows:

Definition 1 (Single-Agent Path Finding). *Given an undirected graph $G = (V, E)$, where V is the set of the vertexes (that correspond to possible locations for the agent) and E the set of edges joining two vertexes (representing the possible transitions between two locations), the Single-Agent Path Finding (SAPF) problem consists in finding the shortest feasible plan π between a starting vertex $v_S \in V$ and a final one $v_F \in V$.*

A plan π is the sequence of N actions $\alpha_i, i \in \{1, \dots N\}$ that take the agent from the starting position $v_S \in V$ to the final position v_F in N steps by following the graph edges:

$$\pi = [\alpha_1, ..., a_N] : \pi(v_S) = \alpha_N(...\alpha_2(\alpha_1(v_S))...) = v_F$$

Where with $\alpha_i(v_s)$ we denote the movement to the vertex $v_e \in V$ from $v_s \in V$, such that $\langle v_s, v_e \rangle \in E$. We denote with $\pi[h], h \leq N$ the h-th action of the plan $\pi = [\alpha_1, ..., \alpha_N]$, i.e. $\pi[h] = \alpha_h$. We also denote with $|\pi| = N$ the length of the plan $\pi = [\alpha_1, ..., \alpha_N]$.

Due to its definition, the SAPF problem can be reduced to the problem of finding the shortest path on a graph. What follows is a brief description of the main algorithms that can be applied to single-agent path finding which can be divided in deterministic algorithms (e.g. Dijkstra's) and heuristic ones (e.g. A*).

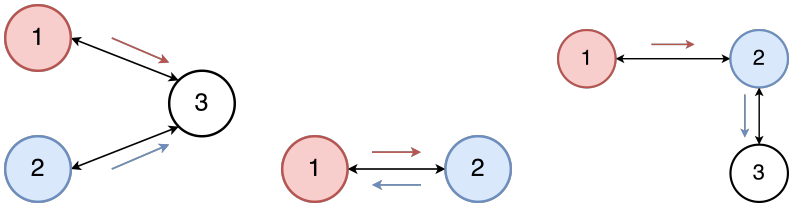
Dijkstra's Algorithm Dijkstra's algorithm [9] aims to find the shortest path between two nodes on a graph whose edges have only positive values. Note that the graph needs to be strongly connected, i.e., there must be at least one path between any two nodes. While this seems quite a strong limitation, industrial scenarios usually provide such graph: no node can be a sink since it must be possible for an agent to come back from each location, that is, usually graphs modeled on warehouses are either undirected, and hence strongly connected, or directed but no node can be a sink. The work of Dijkstra published in 1959 [9] presents two possible algorithms, one to find the shortest path from one node to another and one to find a tree of minimum length starting from a node and reaching all the other nodes. We focus on the second aspect. The complexity of the algorithm depends on the number of vertexes and edges. Moreover, different and improved versions of the algorithm have different worst-case performance, but the initial one proposed by Dijkstra runs in time $O((|V| + |E|) \log |V|)$. Finally, the algorithm has been successfully used in robot path planning [28, 7, 16].

A* Algorithm A* is an heuristic best-first search algorithm for finding the shortest path on a graph [25]. It is also an admissible algorithm, that is, it is guaranteed to find an optimal from the starting node to the arrival one [11]. The idea of A* is to direct the search over the nodes towards the arrival node without having to necessarily examine all the vertexes. To do so, A* keeps a set of nodes to be visited, which is initialized with only the starting node, but then it is enlarged with the neighbors that the algorithm deems worthy to be expanded. A node is said to be expanded when it is added to the set to be analyzed later on. The choice of which nodes should be expanded and which not, is given by the heuristic function. Indeed, when examining the neighbors $u \in$ `neigh(n)` of the considered node, A* uses a heuristic $h(u)$ to estimate the distance to the arrival vertex. Let $h*(u)$ be the perfect heuristic, that is, a function that returns the correct distance from the node u to the arrival vertex, then if $h*(u)$ is known for all the nodes, the best path is obtained just by choosing to go to the neighbor with the lower heuristic distance between neighbors. It has been proved that if $h(n) \leq h*(n)$, then the heuristic is admissible and A* is optimal [11].

3 Problem Statement

The Multi-Agent Path Finding (MAPF) problem is the problem of planning feasible movements for multiple agents [18] such that each one can reach its final location from a respective initial.

Definition 2 (Multi-Agent Path Finding). *Given a finite set $A = \{a_1, ..., a_k\}$ of k agents, given an undirected graph $G = (V, E)$, where V is the set of the vertexes (that correspond to possible locations for the agents) and E the set of edges joining two vertexes (representing the possible transitions between two locations), given an initial start location $v_S^{a_i} \in V$ and final location $v_F^{a_i}$ for each agent a_i, the multi-agent path finding (MAPF) problem consists in finding a joint* feasible *plan $\Pi = \{\pi_{a_1}, ..., \pi_{a_k}\}$ such that for each π_{a_i}, $\pi_{a_i}(v_S^{a_i}) = v_F^{a_i}$, and it minimizes a given cost function $C(\Pi)$.*



(a) vertex conflict (b) swap conflict (c) follow conflict

Fig. 1. The different kinds of conflicts.

In this work, we focus on edges with unitary cost (i.e. all edges have cost 1, whereas extensions for which edges have non-unitary costs will be left for future work). We say that a joint plan Π is *feasible* if no conflict happens between any two different agents. In the literature, the most widely used notions of conflicts are the following [18]:

- *Vertex conflict*: when two agents $a_i, a_j \in A$ with $i \neq j$ are not occupying the same vertex at the same time. We say that the two agents have a vertex conflict iff $\exists 1 \leq h \leq N$ such that $\pi_{a_i}[h](v_S^{a_i}) = \pi_{a_j}[h](v_S^{a_j})$.
- *Edge conflict*: when two agents $a_i, a_j \in A$ with $i \neq j$ are aiming to use the same edge on the same direction at the same time. We say that the two agents have an edge conflict iff $\exists 1 \leq h < N$ such that $\pi_{a_i}[h](v_S^{a_i}) = \pi_{a_j}[h](v_S^{a_j}) \wedge \pi_{a_i}[h+1](v_S^{a_i}) = \pi_{a_j}[h+1](v_S^{a_j})$.
- *Swap conflict*: when two agents $a_i, a_j \in A$ with $i \neq j$ are aiming to use the same edge but on opposite direction at the same time. We say that the two agents have a swap conflict iff $\exists 1 \leq h < N$ such that $\pi_{a_i}[h](v_S^{a_i}) = \pi_{a_j}[h+1](v_S^{a_j}) \wedge \pi_{a_i}[h+1](v_S^{a_i}) = \pi_{a_j}[h](v_S^{a_j})$.
- *Follow conflict*: when two agents $a_i, a_j \in A$ with $i \neq j$ are such that agent a_i want to occupy a position at a given time h that was occupied by agent a_j at time $h-1$. We say that the two agents have a follow conflict iff $\exists 1 < h \leq N$ such that $\pi_{a_i}[h](v_S^{a_i}) = \pi_{a_j}[h-1](v_S^{a_j})$.

In Fig. 1, we provide a pictorial representation for the vertex, swap, and follow conflicts. The edge conflict is pictorially similar to the swap conflict where the two agents are in the same location and want to take the same edge. It should be noted that avoiding vertex conflicts will avoid edge conflicts by definition.
In the literature, two different kinds of cost function $C(\Pi)$ have been considered: the *makespan* and the *sum of costs* (we refer to [18] for a more thorough discussion).

- The *makespan* is a function that returns the length of the longest plan $\pi_{a_j} \in \Pi$: I.e. $C(\Pi) = \text{MKS}(\Pi) = \max_{\pi_{a_i} \in \Pi} |\pi_{a_i}|$. Thus, minimizing the makespan means finding the plan that contains the shortest path among the possible longest paths.
- The *sum of costs* is a function that returns the sum of the individual cost of the different plan $\pi_{a_j} \in \Pi$: I.e. $C(\Pi) = \text{SIC}(\Pi) = \sum_{\pi_{a_i} \in \Pi} |\pi_{a_i}|$. Here we

assume that each action costs 1. If a cost c_e for $e_i \in E$ is associated to each edge, then instead of the length of the plan, one has to consider the sum of the cost of each action in the plan.

The classical multi-agent path finding problem has been proved to be NP-hard, i.e., it is not possible to find an optimal solution in polynomial time [30, 17, 26]. Notice that the problem is NP-hard when finding an optimal solution, i.e., a solution that minimize the objective function, may it be the makespan or the sum of individual costs.

3.1 Solutions

In the literature, several algorithms to solve the MAPF problem have been proposed. These algorithms can be *correct* and *complete* (i.e. if they terminate with a solution, then the computed solution is a solution to the given MAPF problem, and if the problem admits no solution the algorithm says that no solution exists); and can compute an *optimal* solution if it minimizes the given cost function, or a *bounded optimal* one if the computed solution minimizes the cost function within a given bound (i.e. there is some degree of freedom), or a *non optimal* one if there is no guarantee of optimality for the computed solution.

In the following description, we consider the these approaches with the corresponding algorithms: the Kornhauser's algorithm [13], the Extended A* [24], the Increasing Cost Tree Search (ICTS) [22], the Constraint Based Search (CBS) [21], and the Constraint Programming (CP) and Mixed Integer Linear Programming (MILP) approaches.

The Kornhauser's algorithm [13] is a complete but not optimal algorithm that solves the MAPF problem in $O(|V|^3)$. This algorithm considers all the agents in their position, and it tries to move one single agent to a neighbor free location one at a time with the aim to find a way to move all the agents from one arrangement to another. The solution is obtained by decomposing the problem in sub-problems each one composed by the agents that can reach the same set of nodes and the sub-graph made of these nodes [19]. This algorithm has been considered very hard to be implemented efficiently [25].

The extended A algorithm* considers moving all possible agents from one location to a free neighbor one at the same time. This results in a search space of $|V|^k$ and a branching factor of $\left(\frac{|E|}{|V|}\right)^k$, which are both exponential in the number of agents and hence intractable [25]. Two extensions were proposed to solve the MAPF problem [24]: Operator Decomposition (OD) and Independence Detection (ID). The first aims at reducing the exponential branching factor while the other tries to decouple the problem of k agents to smaller problems with less agents. The two extensions can also be combined. This algorithm is correct, complete and optimal.

The Increasing Cost Tree Search (ICTS) algorithm is a two-stage search in which a high-level search aims at finding the lengths of the paths for the different agents, while the low-level search carries out the creation of the path for the

various agents with the *cost constraints* given by the high-level search [25, 22]. This algorithm creates a tree called Increasing Cost Tree (ICT) in which each node contains a vector of the costs C_i of the individual path of each agent a_i. The total cost of the node C is given by the result of the objective function applied to the joint plan and all the nodes at the same level in the tree have the same total cost. The root of the tree is initialized with the costs of the individual paths of the agents as if they were considered in a SAPF problem. If there are no conflicts, then the solution is fine as it is and the algorithm stops. If instead a conflict was found, then k new nodes are going to be created, one for each agent: the i-th node is composed of the solution of the parent and by only increasing the cost solution for the i-th agent by one unit than before. The idea is the following: if with a given solution it was not possible to find a solution without conflicts, then it may be possible to find a solution by increasing the path of an agent by one. The algorithm continues until a solution is found. The ICT nodes not containing conflicts are called *goal nodes*. The low-level search is instead the part of the algorithm that has to find a path for the i-th agent of cost C_i and such that it reaches its final destination. There may be different implementations for this part of the algorithm: the most trivial would be to start from the initial node and enumerate all the possible path of length C_i and check which are reaching the final node. This though may become very expensive as the number of possible paths of cost C_i may be exponential. The solution proposed [22] uses an Multi-value Decision Diagram (MDD) [23] which are a generalization of the binary decision diagrams in the sense that they allow for more than two choices for every node. Basically, the MDD has a single source node which corresponds to the starting node of the agent. Then, it keeps track of all the neighbors of the source node adding them only if the path going through them can lead to the final node with cost C_i. This implies also that the MDD has a single sink and that it is the final goal of the agent. The problem is then how to choose which path is best to return to the high-level search since a path may produce more conflicts than another leading to a bigger and sub-optimal ICT. This is done by doing the cross-product, i.e., merging, the different MDDs and removing those branches that contains conflicts. We remark that, given the structures of the ICT and of the cross-product of the MDDs, the optimization problem can be reduced to a satisfaction problem: the first ICT node that satisfy the constraint of not having any conflict is also going to be optimal, and the same is true for the paths found in the combination of the MDDs.

The Constraint Based Search (CBS) algorithm uses two distinct search processes similarly to ICTS, a high-level and a low-level one, and a tree to solve the MAPF problem. Differently from ICTS, the CBS algorithm builds a Constraint Tree (CT) composed of nodes tracking three elements: i) the joint plan; ii) the cost of the joint plan; iii) a set of *constraints* associated with the joint plan. The idea is that whenever a joint plan contains a conflict, it is resolved by creating two new nodes with different constraints, which are limitations of an agent movement. In particular, the original CBS [21] defines constraint as a negative restriction tuple (a_i, n, t), meaning that the agent a_i is not allowed to be on node n at time

t. The protocol works in the following way: the root is built by considering the paths of the agents as in a single-agent path finding (SAPF) problem. Then, the high-level search checks for possible conflicts. Let π_i and π_j be the plans for agents a_i and a_j respectively, and suppose that they have a vertex conflict at time t on node n. Then, the high-level search creates two new CT nodes from the parent, one in which agent a_i *cannot* be on node n at time t, and the other CT node in which agent a_j *cannot* be on node n at time t. An improvement to CBS [3] suggests that using two positive constraints and a negative one may produce better results since the set of paths that complies with the constraints is disjoint [25]. This means that, instead of having two children from a node, the high-level search creates three children, one in which agent a_i must be on node n at time t, one in which agent a_j must be on node n at time t and one in which neither of them is allowed to be on node n at time t. The process of expanding nodes, i.e., creating new children, stops when there are no more conflicts to be solved. Whenever a new node is added, the low-level search is called to find a solution to the problem with the new added constraints. If a feasible solution can be found, then the node is added to the set of nodes to be further explored. To pick the next node to examine, CBS uses the cost function of the joint plan. Finally, as it regards the low-level search, it can be any SAPF algorithm, although it needs to be properly modified to support the presence of constraints.

The Constraint Programming (CP) approach leverages a mathematical modeling paradigm in which the problem is encoded as a set of constraints among two kind of variables: state variable and decision variables. This approach is usually divided in two parts: a modeling part that addresses the shaping of the aspects of the problem introducing variables over specific domains and constraints over such variables; a solving part that aims at choosing the value of the decision variables that minimize a given cost function and that make the constraints satisfiable. If the constraints are well-formed, i.e., they correctly cover the variables and their domains, than constraint programming is both optimal and correct. Typical modeling considers Boolean variables for each agent for each vertex for each time point, and a constraint that enforces that each agent can occupy only one vertex in each time step (thus ensuring no vertex conflict). Agents are positioned on their initial position at the first time step, and must be on their arrival position at the last time step. Agents move along edges towards neighbors of the node on which they are: this is to ensure the validity of the solution since an agent cannot jump from one node to another. Once the constraints are fixed, the model can be solved with any off-the-shelf constraint solver, which tries to look at all the possible combinations without infringing any constraint.

4 Experimental Evaluation

In this section, we first provide the high level details of the implementation of the considered algorithms, and the information about the software and hardware infrastructure used for the experiments. Then we describe the considered indus-

trial scenarios, we report and critically discuss the results of the experiments.

4.1 Implementation

For the implementation we have considered only three of the approaches discussed in Section 3. We implemented the CP approach and two variants of the CBS family of algorithms, in particular the Spanning Tree (ST) and the Time-Dependant Shortest Path (TDSP). The CBS ST and the CBS TDSP differs in the low-level search used to build the constraint tree. The CBS ST in the local search builds a spanning tree as to allow the high-level search to choose among the possible different paths that have the same length. The CBS TDSP in the local search uses a variant of the Dijkstra [9] algorithm to compute shortest paths where the costs of the edges depends on the time the edge is considered. We do not report here the pseudo-code of the considered algorithms for lack of space, and we refer to [20] for further details. We decided not to implement the Kornhauser's algorithm since this algorithm has been considered very hard to be implemented efficiently from the research community [25], and it produces non-optimal solutions. We did not implement the extended A* algorithm because of its large branching factor that will make it not applicable in large industrial scenarios. Finally, we also did not implement the ICTS approach since it requires to know possible bounds for the costs of the searched solutions a priori (an impractical information to get for realistic scenarios).

All the algorithms have been implemented in C++ using the standard template libraries. For the CP algorithm we have leveraged the latest release of the C++ API of the CPLEX commercial constraint solver [8]. The source code with the implementation of all the algorithms is available at our open repository[1].

We run all the experiments on an AMD Ryzen 3700X equipped with an 8 core CPU at 3.6GHz base clock, and 32GB of RAM running Linux. We considered as run-time timeouts 1s, 10s, and 60s to mimic the time response expectations requested in industrial realistic scenarios.

4.2 Industrial scenarios

For the experiments we considered a real warehouse taken from a collaboration with a company operating in the field of robotic assisted warehouses. The entire warehouse and its graph representation is depicted in Fig. 2. The topological graph obtained from the map consists of 414 nodes with undirected edges. For the experiments we decomposed the warehouse into sub-problems as follows: i) `WH1` that corresponds to the gold rectangle in the top right corner of Fig. 2; ii) `WH2` that corresponds to the blue rectangle in the bottom left corner of Fig. 2; iii) `WH2_1` that corresponds to the red rectangle in the bottom left corner of Fig. 2; iv) `WH2_2` that corresponds to the green rectangle in the bottom left corner of Fig. 2; v) `WH2_1_1` that corresponds to the top 4 rows of red rectangle in the bottom left corner of Fig. 2; vi) `WH2_1_2` that corresponds to the bottom 4 rows of red rectangle in the bottom left corner of Fig. 2; vii) `WH2_2_1` that

[1] https://www.bitbucket.org/chaff800/maof

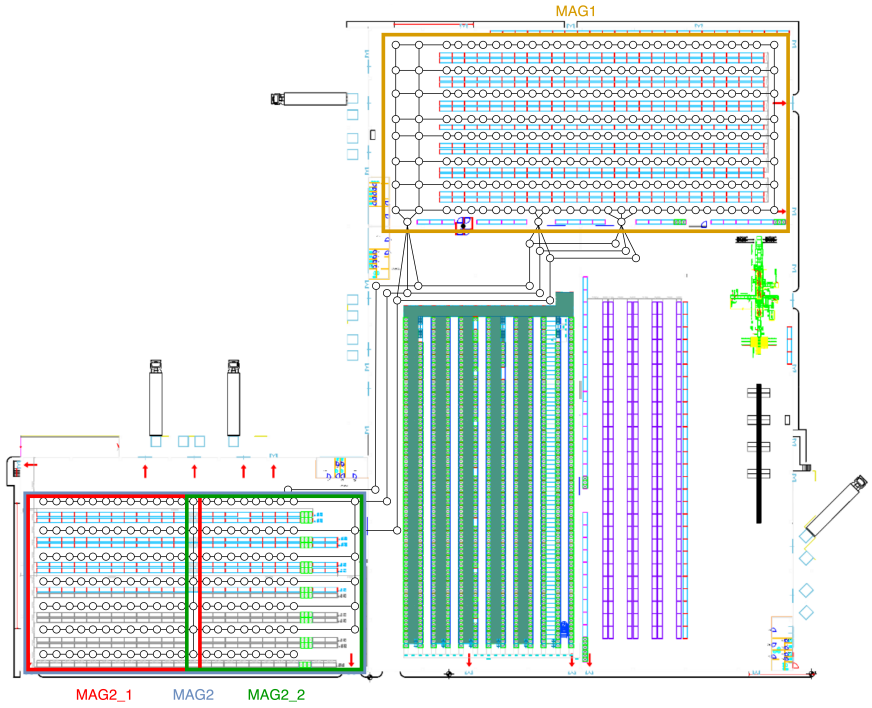


Fig. 2. The schema of the real warehouse considered for the experiments.

corresponds to the top 4 rows of green rectangle in the bottom left corner of Fig. 2; viii) `WH2_2_2` that corresponds to the bottom 4 rows of green rectangle in the bottom left corner of Fig. 2. For each scenario, we considered problems with increasing number of robotic agents taken from $\{2, 5, 10, 20\}$ and increasing number of goals taken from $\{1, 2, 5, 10, 20\}$. These numbers are the results of the discussion with the company owner of the reference warehouse we considered. The goals have been generated to resemble typical goals taken from the logistic activities carried out in the considered warehouse. In the results, we only report the name of the scenario followed by the number of problems considered in that scenario in parenthesis (E.g., `WH2_2_2` (10) means the scenario `WH2_2_2` with ten problems). For each experiment, we report the number of problems solved among the one considered, and the average search time in milliseconds (ms) required for the solved problems. We use TO to indicate that the algorithm was not able to find a solution within the given time budget for any of the problem in the scenario.

4.3 Results

The results are reported in the Table 1: the upper left table reports the results for CBS with TDSP; the upper right table reports the results for CBS with ST; the lower down table reports the results for CP. For CP we also report the

Test \ Timeout	Test completed 1s	10s	60s	Average run time [ms] 1s	10s	60s
WH2_2_2 (12)	2	2	2	5.58	9.96	5.53
WH2_2_1 (12)	1	1	1	5.29	4.88	4.84
WH2_1_1 (20)	0	0	0	*TO*	*TO*	*TO*
WH2_1_2 (15)	0	0	0	*TO*	*TO*	*TO*
WH2_1 (20)	0	0	0	*TO*	*TO*	*TO*
WH2_2 (20)	1	1	1	1.93	1.94	3.64
WH2 (20)	0	0	0	*TO*	*TO*	*TO*
WH1 (20)	1	1	1	159.91	159.05	158.77
WH12 (25)	0	0	0	*TO*	*TO*	*TO*

Results for CBS with TDSP.

Test \ Timeout	Test completed 1s	10s	60s	Average run time [ms] 1s	10s	60s
WH2_2_2 (12)	2	2	2	0.70	0.61	0.64
WH2_2_1 (12)	2	3	4	6.14	3051.67	6130.67
WH2_1_1 (20)	0	0	0	*TO*	*TO*	*TO*
WH2_1_2 (15)	0	0	0	*TO*	*TO*	*TO*
WH2_1 (20)	0	0	0	*TO*	*TO*	*TO*
WH2_2 (20)	2	2	2	31.86	31.96	34.04
WH2 (20)	0	0	0	*TO*	*TO*	*TO*
WH1 (20)	0	0	0	*TO*	*TO*	*TO*
WH12 (25)	0	0	0	*TO*	*TO*	*TO*

Results for CBS with ST

Test \ Timeout	Test completed 1s	10s	60s	Average run time [ms] 1s	10s	60s	Average memory [MB] 1s	10s	60s
WH2_2_2 (12)	0	2	2	*TO*	4459.78	4688.98	714.33	3195.38	9310.51
WH2_2_1 (12)	0	0	0	*TO*	*TO*	*TO*	863.83	3653.43	12630.42
WH2_1_1 (20)	0	0	0	*TO*	*TO*	*TO*	973.95	7311.85	19306.53
WH2_1_2 (15)	0	0	0	*TO*	*TO*	*TO*	951.51	4719.59	14422.11
WH2_1 (20)	0	0	0	*TO*	*TO*	*TO*	776.29	9682.30	24465.94
WH2_2 (20)	0	0	1	*TO*	*TO*	45942.20	896.01	7629.73	20344.22
WH2 (20)	0	0	0	*TO*	*TO*	*TO*	733.01	8299.46	25802.48
WH1 (20)	0	0	0	*TO*	*TO*	*TO*	1071.18	6575.35	29965.53
WH12 (25)	0	0	0	*TO*	*TO*	*TO*	1438.79	7072.68	29261.20

Results for CP

Table 1. Results for CBS with TDSP (up left), CBS with ST (up right), CP (down).

average memory in megabytes (MB) required to either find a solution or used before ending in timeout.

The results clearly show that none of the considered algorithm was able to solve all the problems in the considered budget constraints but only very few cases (e.g. in the `WH2_2`, `WH2_2_2`, `WH1`). In particular, the results show that the CBS algorithms are able to solve slightly more scenarios than CP (which solves only 3 cases in the 60s time boundaries with the best run-time completed in 1.1s). More specifically, the results show that the CBS algorithms are complementary. Indeed, for `WH2_2_2` CBS TDSP is slower than the CBS ST, whereas for `WH1` CBS TDSP is able to solve one instance while CBS ST none ending always in TO. CP is always worse in performance than the CBS algorithms. As the table with the results for CP reports, it is clear that this approach consumes a larger amount of memory w.r.t. the other approaches. Indeed, each time it does not finds a solution, it tries to increase the time steps by 1 unit thus resulting in a much larger complexity due to the used variables matrix structure.

These results clearly show that although these algorithms have been thoroughly studied in the literature, and experimented on random graphs with random goals, when applied to realistic scenarios, they fail to find solutions in typical industrial budget of resources. A more thorough analysis of the cases where no solution was found (even with larger resource budgets) are cases where two robotic agents need to follow the same shortest path but in

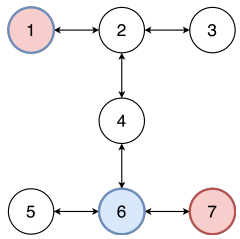


Fig. 3. A simple scenario not solvable by CBS.

opposite direction thus requiring to swap places in one edge (see Fig. 3). In this cases, a simple strategy would move one of the two agents into a lateral position (if available) to allow the other to pass, and then go back to the previous location (thus taking a longer path that visit the same node more than once). The problem in solving such a situation stands in the difficulty to differentiate between a waiting action, which can be done on the node on which the agent currently is, or the action of exploring the neighbors of the node. Algorithms such as TDSP and ST are not meant to visit multiple times the same node. To solve this problem, both the high-level and low-level searches of CBS should be modified, the former to consider multiple possible nodes for a given time step h on the plan of an agent, and the latter to allow moving over the same node multiple times. Both changes are already planned for future works.

5 Conclusions

In this paper, we studied the performance of the state-of-the-art MAPF algorithms on a set of scalable industrial scenarios all derived from a real warehouse of a large distribution company. The results show that the CP approach find optimal solutions, but it is applicable to only very small scenarios. The CBS approaches scale better and allows to solve in the given resource budgets more problems. However, these approaches fail to find a solution in cases where it was requested some agent to move to other locations and then go back to the same location to continue the motion to allow other agents to exit from conflicting cases. This particular case is really likely to happen by construction of the graph: the aisles are long and they can basically be occupied by just one agent at a time without having to solve many swap conflicts.

The results show that there is not a clear winner, but all the approaches have pros and cons. This work paves the way for several future works that go from investigating new heuristics to solve hard problems that appear in real scenarios, to new algorithms that combine the pros of each approach, or that consider the use of divide-et-impera approaches to leverage different low-level search strategies. Moreover, we aim also to extend the work so that each agent does not only consider a set of tasks, but also other information such as batteries level and the possibility to recharge. Also, while in this work we have given the mission planning for granted, integrating mission planning in the MAPF problem may lead to more effective ways of allocating tasks to the different agents to minimize the overall cost of the computed solution.

The final goal is an open source framework containing different MAPF solvers that can be used to tackle the problem and that may be integrated in platforms such as ROS. For this same reason, the algorithms have been re-implemented instead of employing pre-existing code. Moreover, any existing code would have had to be adapted to our use-case, leading to a loss in performance.

References

1. Arnold, R.D., Yamaguchi, H., Tanaka, T.: Search and rescue with autonomous flying robots through behavior-based cooperative intelligence. Journal of International Humanitarian Action **3**, 18 (12 2018). https://doi.org/10.1186/s41018-018-0045-4
2. Bhattacharya, S., Likhachev, M., Kumar, V.: Topological constraints in search-based robot path planning. Autonomous Robots **33**, 273–290 (10 2012). https://doi.org/10.1007/s10514-012-9304-1
3. Boyarski, E., Felner, A., Stern, R., Sharon, G., Betzalel, O., Tolpin, D., Shimony, S.E.: Icbs: The improved conflict-based search algorithm for multi-agent pathfinding (2015)
4. BraganÃ§a, S., Costa, E., Castellucci, I., Arezes, P.M.: A brief overview of the use of collaborative robots in industry 4.0: Human role and safety (2019). https://doi.org/10.1007/978-3-030-14730-3_68
5. Brett, P., Taylor, R., Proops, D., Coulson, C., Reid, A., Griffiths, M.: A surgical robot for cochleostomy. pp. 1229–1232. IEEE (8 2007). https://doi.org/10.1109/IEMBS.2007.4352519
6. Brumitt, B., Stentz, A.: Dynamic mission planning for multiple mobile robots. pp. 2396–2401. IEEE. https://doi.org/10.1109/ROBOT.1996.506522
7. zhou Chen, Y., fei Shen, S., Chen, T., Yang, R.: Path optimization study for vehicles evacuation based on dijkstra algorithm. Procedia Engineering **71**, 159–165 (2014). https://doi.org/10.1016/j.proeng.2014.04.023
8. Corportation, I.: Ibm ilog cplex optimization studio
9. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**, 269–271 (12 1959). https://doi.org/10.1007/BF01386390
10. Ferrari, F., Divan, S., Guerrero, C., Zenatti, F., Guidolin, R., Palopoli, L., Fontanelli, D.: Human–robot interaction analysis for a smart walker for elderly: The acanto interactive guidance system. International Journal of Social Robotics **12**, 479–492 (5 2020). https://doi.org/10.1007/s12369-019-00572-5
11. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4**, 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136
12. Javaid, M., Haleem, A., Singh, R.P., Suman, R.: Substantial capabilities of robotics in enhancing industry 4.0 implementation. Cognitive Robotics **1**, 58–75 (2021). https://doi.org/10.1016/j.cogr.2021.06.001
13. Kornhauser, D., Miller, G., Spirakis, P.: Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. pp. 241–250. IEEE (1984). https://doi.org/10.1109/SFCS.1984.715921
14. Latombe, J.C.: Robot motion planning, vol. 124. Springer Science & Business Media (2012)
15. Pouke, M.: Using gps data to control an agent in a realistic 3d environment. pp. 87–92. IEEE (9 2013). https://doi.org/10.1109/NGMAST.2013.24
16. Qing, G., Zheng, Z., Yue, X.: Path-planning of automated guided vehicle based on improved dijkstra algorithm. pp. 7138–7143. IEEE (5 2017). https://doi.org/10.1109/CCDC.2017.7978471
17. Ratner, D., Warmuth, M.K.: Finding a shortest solution for the n × n extension of the 15-puzzle is intractable (1986)
18. Roni, S., Nathan, S., Ariel, F., Sven, K., Hang, M., Thayne, W., Jiaoyang, L., Dor, A., Liron, C., Satish, K.T.K., Eli, B., Roman, B.: Multi-agent pathfinding: Definitions, variants, and benchmarks. CoRR **abs/1906.08291** (2019)

19. Röger, G., Helmert, M.: Non-optimal multi-agent pathfinding is solved (since 1984) (2012)
20. Saccon, E.: Comparison of Multi-Agent Path Finding Algorithms in an Industrial Scenario. Master's thesis, Department of Information Engineering and Computer Science - University of Trento (July 2022), https://www5.unitn.it/Biblioteca/en/Web/RichiestaConsultazioneTesi
21. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence **219**, 40–66 (2 2015). https://doi.org/10.1016/j.artint.2014.11.006
22. Sharon, G., Stern, R., Goldenberg, M., Felner, A.: The increasing cost tree search for optimal multi-agent pathfinding. Artificial Intelligence **195**, 470–495 (2 2013). https://doi.org/10.1016/j.artint.2012.11.006
23. Srinivasan, A., Ham, T., Malik, S., Brayton, R.: Algorithms for discrete function manipulation. pp. 92–95. IEEE Comput. Soc. Press. https://doi.org/10.1109/ICCAD.1990.129849
24. Standley, T.: Finding optimal solutions to cooperative pathfinding problems. vol. 24, pp. 173–178 (2010)
25. Stern, R.: Multi-agent path finding – an overview (2019). https://doi.org/10.1007/978-3-030-33274-7_6
26. Surynek, P.: An optimization variant of multi-robot path planning is intractable. vol. 2 (7 2010)
27. Veloso, M.M., Biswas, J., Coltin, B., Rosenthal, S.: Cobots: Robust symbiotic autonomous mobile service robots. pp. 4423 – 4429 (7 2015)
28. Wang, H., Yu, Y., Yuan, Q.: Application of dijkstra algorithm in robot path-planning. pp. 1067–1069. IEEE (7 2011). https://doi.org/10.1109/MACE.2011.5987118
29. Wurman, P.R., D'Andrea, R., Mountz, M.: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. AI Magazine **29**, 9 (3 2008). https://doi.org/10.1609/aimag.v29i1.2082, https://ojs.aaai.org/index.php/aimagazine/article/view/2082
30. Yu, J., LaValle, S.M.: Structure and intractability of optimal multi-robot path planning on graphs. pp. 1443–1449. AAAI Press (2013)