UNIVERSITÀ
DI TRENTO

Department of Information Engineering
and Computer Science

Master Degree in
Computer Science

Final Master Thesis

Comparison of Multi-Agent Path Finding Algorithms in an Industrial Scenario

Supervisor

Luigi Palopoli

Co-supervisor

Marco Roveri

Student

Enrico Saccon

Academic Year 2021/2022

To whoever believed in me: thanks.

Contents

1 Abstract

Robotics is emerging has being one of the most critical aspects of the industrial revolution, allowing for improvements in various scenarios. In particular, in this thesis we will focus on the motion planning for multiple robots with a real warehouse located in the center of Italy as the main scenario.
First, we start with a description of the state-of-the-art algorithm used to solve the Multi-Agent Path Finding (MAPF) problem, showing their strengths and their weak spots.
Then, we move to the description of three algorithms we have implemented to tackle the aforementioned problem explaining the modification we had to make to the algorithms in order to be able to use them in our scenario.
The description of the tests created to examine their characteristics follows. Moreover, we explain how we divided the warehouse map in order to create tests of increasingly difficulty.
Finally a discussion on the results and the future work that may come with it is carried out.

2 Introduction

Robots are becoming a familiar presence in the daily life of people helping them in different scenarios: warehouses, healthcare, search and rescue and offices.
The industrial area is possibly one 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 [1], with a significant impact on productivity [2]. One of the most known examples of robotics applied to the industry is Spot from Boston Dynamics, which not only can freely move in the environment and record it with its cameras, but can also find possible problems and predict which components will need maintenance when integrated with sensors [3]. This is just an example of how robotics can be applied to the industrial domain. Indeed, robotics proves to enhance and solve more and more easily logistics manufacturing problems allowing for a better use of the industrial space [4].
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 [6].
Another important scenario in which robots have be successfully utilized is search and rescue missions in challenging environments [7], where the environmental conditions may otherwise endanger also the rescuers.
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 [8].
In the majority of these situations, there are multiple robots that, in order to complete one or multiple tasks in the most efficient way possible, need to cooperate with each other. Moreover, having the robots follow human-aware trajectories, dramatically increases the reliability and safety of the environment, while yet keeping the benefits of using robotics in the scenario.
The focus of this thesis will be the industrial scenario focusing in particular on motion planning. The main contribution we have worked on are:

- A comprehensive overview of the state of the art regarding techniques and algorithms used to solve motion planning problems, Chapter 3.
- A comparison of some of said techniques addressing a real warehouse scenario with the due modifications, Chapter 4.
- A discussion of the results obtained and the possible problems that should be addressed, Chapter 5.

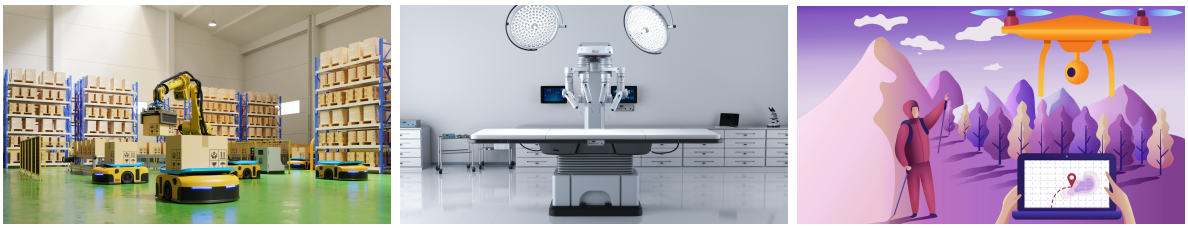


Figure 2.1: Robots can be employed in a different number of scenarios to help humans, from the movement of heavy packages in warehouses, to the precision of surgery, and also to help search and rescue in remote environments.

3 Motion Planning

In this thesis, we focuses 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 [9], the latter focuses on the best way of organizing the goals for each robots in the environment [10]. 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.

3.1 Safe Motion Planning

In environments shared by both humans and robots, one of the main challenges is safe motion planning. Firstly, a path planning algorithm for decentralized agents able to ensure collision avoidance was proposed in 2008, and it was proven by using reserved areas [11]. Prioritized path Planning (PP) is one of the main techniques used to plan robotics paths for multiple agents. It was first explored in 1987 [12] and it has then been refined providing a generally applicable approach without limitations to the form or degrees of freedom of the robots [13]. Later on in 2015, a revised algorithm of PP was presented, proving that if a solution exists, it guarantees to find a path for every robot to reach the final destinations [14]. At the same time, an asynchronous decentralized algorithm for PP that has been proven to find paths for all the robots in the system and to terminate twice as fast was also provided [14]. Another important factor for human-aware motion planning is the smoothing of the trajectory which is done by interpolating the points with well-behaved curves: we proposed a novel solution using dynamic programming for the multi-point Markov-Dubins problem [15] in 2020 and we later refined it using parallel programming on GPUs [16].
The integration of human motion in the planning has been studied for single-agent motion in various works that can then be adapted to a multi-agent scenario. A first reactive model based on velocity obstacles, i.e., a set of velocities which the robot should not use to avoid a dynamic obstacle was proposed in 1998 [17]. In 2020, a refinement was presented which was meant to be applied to multi-agent navigation by using the velocity obstacle paradigm only for local and reactive navigation, while using RTT* to generate the global path of each robot, showing a more human-like and collaborative behaviour of the robots [18]. Another effective solution was proposed in 2018 using a combination of velocity obstacles, dynamic windows, that is, a set of velocities easily reachable by the robot that would allow it to either avoid the obstacle or to stop, and Monte Carlo sampling to account for the dynamic obstacles [19]. Other solutions are based on learning-based methods and predictive planners. The former suffer from the generalization problem and hence produce solutions that are troublesome to deploy in sensitive environments due to the difficulty in providing safety guarantees. The latter instead produce human-aware paths if the models are well-fitted and can rely on precise estimations. Indeed, one of the main

difficulties of multi-agent navigation in environments with the presence of humans is to find a good model for the human motion. Recent works [20] [21] showed promising results using headed social force model, i.e., social force model also considering the direction of the person. In 2019, it has been proved that also the maximum-entropy model was able to offer safe navigation, for which the human dynamics are affected by an action drawn from a Boltzmann probability distribution [22].

3.2 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 waypoints accuracy for remote controlled agents [23]. 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 [24]. Moreover, they have also been adopted to run in more than 2 dimensions such as when running manipulators [25].
This thesis focuses on the path planning problem that can be defined as follows:

Definition 3.1: Single-Agent Path Finding

Given a graph $G = (V, E)$, where V is the set of the vertexes and E the set of edges joining two vertexes, the Single-Agent Path Finding (SAPF) problem consists in finding the shortest feasible plan π between a starting vertex $S \in V$ and a final one $F \in V$.

A plan π is the sequence of s actions that take the agent from the starting position S to the final position F in s steps:

$$\pi = \{a_1 ... a_s\} : a_s(...a_2(a_1(S))...) = F$$

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*).

3.2.1 Dijkstra's Algorithm

Dijkstra's algorithm [26] 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 modelled 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 [26] 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 for which a brief description can be found looking at the pseudo code Algorithm 1.
The complexity of the algorithm depends on the number of vertexes and edges. Moreover,

Algorithm 1: Description of the Dijkstra algorithm

Input: $G = (V, E)$ strongly connected graph
Input: S initial node
Input: F final node

Data: `dist` vector of distances from S
Data: `prev` vector of the nodes from which the path comes from
Data: `OPEN` set of nodes to be considered

for $v \in V$ **do**
    `dist[v]` $\leftarrow inf$
    `prev[v]` $\leftarrow$ `NULL`
    `OPEN`.$add(v)$
`dist[`S`]` $= 0$

while *`OPEN`* **do**
    $u \leftarrow$ node from `OPEN` with smaller distance
    `OPEN`.$del(u)$

    for $v \in$ *`neigh(u)` still in `OPEN`* **do**
        $tmp \leftarrow$ `dist[u]` $+ E(u, v)$
        if $tmp <$ *`dist[v] and dist[u]`* $!= inf$ **then**
            $dist[v] \leftarrow tmp$
            $prev[v] \leftarrow u$

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 [27, 28, 29].

3.2.2 A* Algorithm

A* is an heuristic best-first search algorithm for finding the shortest path on a graph [30]. It is also an admissible algorithm, that is, it is guaranteed to find an optimal from the starting node to the arrival one [31].
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, usually called `OPEN`, which is initially set to only the starting node, but then it is added with the neighbors that the algorithm deems worthy to be expanded. A node is said to be expanded when it is added to the `OPEN` 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 [31].

3.3 Multi-Agent Path Finding

The multi-agent path finding (MAPF) problem is the problem of planning a feasible path for multiple agents [32]:

Definition 3.2: Multi-Agent Path Finding

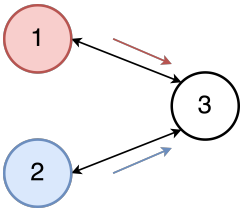
Given a graph $G = (V, E)$, a function S that maps the agents to their initial positions and a function E that maps the agents to the their final positions, the problem amounts to finding a feasible joint plan Π such that each agent reaches its destination, while globally minimizing an objective function.

A joint plan Π is the set of the individual plans π_i that each agent a_i has to follow to reach its destination:

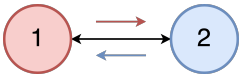
$$\Pi = \{\pi_1, \ldots, \pi_k\}$$

There are different definitions of the problem in terms of how the graph topology is assumed to be, but the classical one considers [32]:

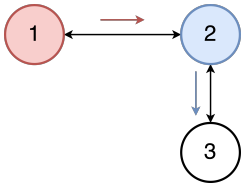
- G is an *undirected* graph;
- A node (or vertex) can host only one agent at the time;
- The edges between nodes have a binary value, either there is a connection and it costs 1, or there is no connection;
- It follows that time is assumed to be discretized into timestamps: at each time the agent can either move to a neighbor or stay on the node it is on.



(a) A vertex conflict occurs when two agents want to move to the same node at the same time.



(b) A swap conflict happens when two agents want to move on the same edge at the same time in opposite directions.



(c) A follow conflict happens when an agent wants to move to a node that is currently occupied by another node but will be then be freed.

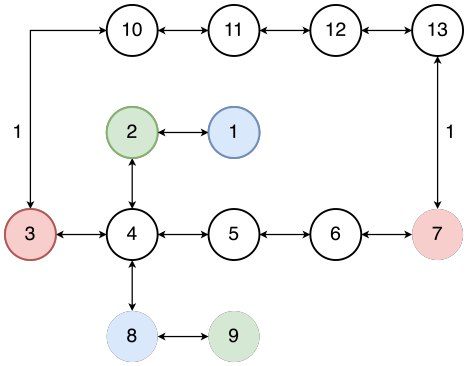
Figure 3.1: The most common type of conflicts considered for the MAPF problem.

A joint plan Π is simply the set of the paths that each agent needs to follow to reach its goal, and it is said to be feasible if no conflict happens between different agents. In literature, there are different definitions of conflict, but the most important ones are the following [30]:

- *Vertex conflict*: when two agents a_i and a_j are on the same node at the same time t (Figure 3.1a):
$$\pi_i(t) = \pi_j(t) \iff \text{vertex conflict}$$
- *Edge conflict*: when two agents a_i and a_j move on the same edge in the same direction at the same time t;
$$\pi_i(t) = \pi_j(t) \wedge \pi_i(t+1) = \pi_j(t) \iff \text{edge conflict}$$
- *Swap conflict*: when two agents a_i and a_j are moving on the same edge, but in different directions at the same time t (Figure 3.1b).
$$\pi_i(t) = \pi_j(t+1) \wedge \pi_i(t+1) = \pi_j(t) \iff \text{swap conflict}$$

Given the definition of vertex conflict, it follows that in the classical definition of MAPF, the edge conflict is avoided a priori by imposing the absence of vertex conflicts since since two agents at time t must have different positions. For this reason, the terms edge conflict and swap conflict may be used to indicate the same conflict in this work.
Another type of conflict is the *follow conflict*, shown in Figure 3.1c which happens when an agent wants to occupy a position at time t that was occupied by another agent at time $t-1$. This is particularly interesting for an industrial scenario since a follow conflict may lead to delays or even collisions. Indeed, when an agent stops at a goal point to be loaded, it may undergo some delays leading to possible problems for the other agent following up. In this work, the following conflict will not be taken into consideration as it is not strictly part of the classical MAPF problem.
Another important aspect of the MAPF problem is the objective function to minimize. The reason to have a cost to minimize is due to the fact that the problem may otherwise have different solutions as it is possible to see in Figure 3.2. The classical problem considers mainly two objective functions: the *makespan* and the *sum of costs*.
The makespan $\texttt{MKS}(\Pi)$ is a function that returns the length of the longest path in the plan:

$$\texttt{MKS}(\Pi) = \max_{\pi_i \in \Pi} |\pi_i| \tag{3.1}$$



Π_i	$\texttt{SIC}(\Pi_i)$	$\texttt{MKS}(\Pi_i)$
$\Pi_1 = \begin{cases} \pi_1 = \{3, 10, 11, 12, 13, 7\} \\ \pi_2 = \{2, 4, 8, 9\} \\ \pi_3 = \{1, 2, 4, 8\} \end{cases}$	14	6
$\Pi_2 = \begin{cases} \pi_1 = \{3, 4, 5, 6, 7\} \\ \pi_2 = \{2, 2, 4, 8, 9\} \\ \pi_3 = \{1, 1, 2, 4, 8\} \end{cases}$	15	5

Figure 3.2: On the left a MAPF problem with three agents: the first agent in red wants to go from 3 to 7, the second agent in green wants to go from 2 to 9 and the last agent in blue wants to go from 1 to 8. On the right a table summarizing the two possible solutions and showing how the solution changes by using one objective function instead of the other. Notice that the set of possible solutions is not exhaustive, although these two are the best ones and hence the ones that would be chosen.

Minimizing the makespan means finding the plan that contains the shortest path among the possible longest paths.
The sum of costs, or sum of individual costs (SIC), is instead the sum of the costs of all the paths inside the plan:

$$\texttt{SIC}(\Pi) = \sum_{\pi \in \Pi} |\pi_i| \tag{3.2}$$

3.3.1 Solutions

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 [33, 34, 35]. 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. The Kornhauser's algorithm [36] solves the MAPF problem in $O(|V|^3)$, but the solution it returns it may not be the optimal one.
So an algorithm can be optimal and complete, or a combination of the two. We say that the algorithm is *optimal* if the solution it returns is the solution that minimizes the objective function, while we say that a solution is suboptimal, or *bounded suboptimal*, when the solution it finds minimized the cost within a certain degree of freedom. An algorithm is said to be *complete* when it guarantees to return a solution.

3.3.1.1 Kornhauser's

The Kornhauser's algorithm is complete but not optimal. It extends the famous 15-puzzle problem which consists in having15 agents moving no a 16 4-connected nodes graph as shown in Figure 3.3, where there is only one free node and the agents can move one step at a time to reach their final destination.
The considered generalization is the following [36]: let G be a graph with n vertices with $k < n$ agents on different vertices and let a move be an agent shifting to a free adjacent vertex. The question the algorithm aims to answer is whether it is possible or not to find a way to move all the agents from one arrangement to another. The solution is obtained by decomposing the problem in subproblems each one composed by the agents that can reach the same set of nodes and the subgraph made of these nodes [37].

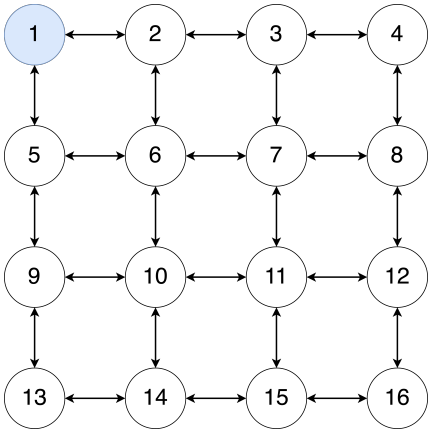


Figure 3.3: The 15 puzzle problem is a 16 node undirected graph problem in which there are 15 agents that need to move and only one free location (represented in blue.)

The algorithm is generally considered difficult to be implemented [30] and in this work the algorithms that are discussed are the optimal ones instead of suboptimal.

3.3.1.2 Extended A*

One could be tempted to use A* to solve the MAPF problem by simply running the algorithm for each of the k agents and finding the shortest path from each node to each destination. This though proves to be really inefficient. Indeed, by considering the search space, i.e., the number of vertexes $|V|$, and the branching factor, i.e., the average number of edges exiting from each node, it is easy to see that the previous solution would lead to 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 [30].
Two extensions were proposed to solve the MAPF problem [38]: 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.
When using A* in MAPF the problem is that the moves of all agents are considered at the same time leading to an exponential branching factor. The idea of OD is to move one agent at the time, so only after k moves, the agents will all have changed their position. The algorithm proposes a different representation of the state space in which vertexes that represent the positions of all agents at the same time are called **full vertexes** and are the ones that happen every k moves, while all the other vertexes are called **intermediate vertexes** [**?**, **?**]. This allows to have a branching factor of $O\left(k \times \frac{|E|}{|V|}\right)$, which is not exponential. Moreover, it is important to notice that the heuristic used in A* may help in deciding whether an intermediate node is worth to be expanded speeding up the search.
ID is meant to reduce the main problem of k agents to easier subproblems. The idea is to start by computing the best plan for each agent as if they were alone on the graph by using A* . If a conflict is found, then the two agents leading to the conflict are considered together as a meta-agent and the best path for the two agents avoiding conflicts is computed with A* with OD. This process is iterated every time a conflict arises and is solved by merging the new agent into the meta agent. The process stops when there are no more conflicts [38].
Notice that actually ID is a general framework and one could replace the A* with OD with any other MAPF solver.

3.3.2 Increasing Cost Tree Search (ICTS)

This algorithm is actually a two-component system 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 [30, 39]. In particular, the 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 sum of the internal costs $C = C_1 + \ldots + C_k$ 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, with the only change that the cost solution for the i-th agent is one more 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 [39] uses an Multi-value Decision Diagram (MDD) [40] 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. Notice two things: 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.

3.3.3 Constraint Based Search (CBS)

Similarly to ICTS, it uses two distinct search processes, a high-level and a low-level one, and a tree to solve the MAPF problem.
The CBS tree is called Constraint Tree (CT) and is composed of nodes tracking three elements:

- The joint plan;
- The cost of the joint plan;
- 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 [41] 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.

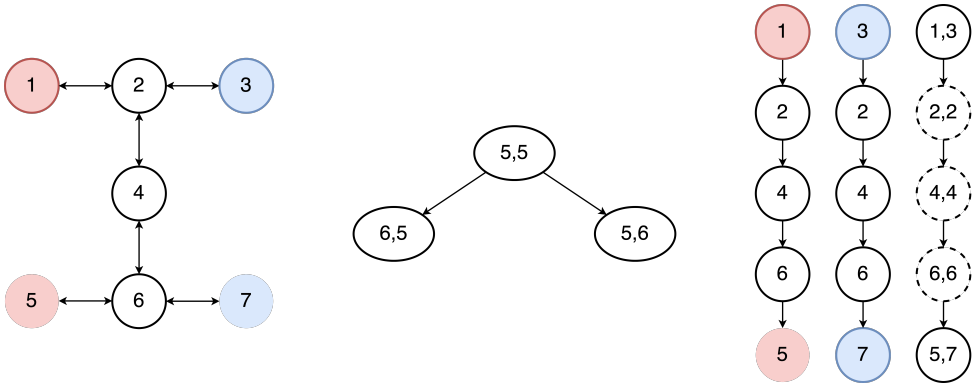


Figure 3.4: On the left a MAPF problem with two agents: one moving from 1 to 5 and the other moving from 3 to 7. In the middle, the ICT that comes from solving the problem by using ICTS: the algorithm starts with the lengths of the paths as if solving a SAPF problem. By finding a conflict as depicted in the MDDs on the right, it creates two nodes and tries to find a combination of paths solving the problem with those costs. Finally, on the right, the two MDDs for the two agents with cost 5 and the combination of the MDDs showing the conflicts with dotted lines.

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 [42] 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 [30]. 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.

3.3.4 Constraint Programming (CP) and Mixed Integer Linear Programming (MILP)

Constraint programming is an alternative approach to the imperative paradigm, which is mainly used when coding [43]. MILP is a mathematical modeling paradigm in which the decision variables are subject to some inequalities. These types of programming are usually divided into two parts: a first one called modelling that addresses the shaping of the aspects of the problem introducing variables over specific domains and constraints over such variables. A second part instead aims at choosing how to solve the representation of the problem that was modelled. There are various types of solver, for example, one could use a CP solver or model the problem

so that it can be solved with Boolean Satisfiability (SAT) or Mixed Integer Programming (MIP) solvers.
If the constraints are well-formed, i.e., they correctly cover the variables and their domains, than constraint programming is both optimal and correct. A brief description of the constraint used in [44] to solve the MAPF problem is now carried out, a deeper and more precise one will be performed later on. The main constraints that are needed are the following [30]:

- Agents occupy only one vertex in each time step: this is to ensure that we are solving the classical MAPF, whereas a MAPF problem for large agents takes into consideration the fact that each agent may occupy more than 1 vertex at a time;
- A node can be occupied by only one agent in each time step: it guarantees that there are 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 a solver, which tries to look at all the possible combinations without infringing any constraint.

3.3.5 MAPF Variations

Below, a brief description of some MAPF variations is carried out. The list is not exhaustive as it introduces only those variations from which some key aspects have been taken into consideration to model the problem this work focuses on.

3.3.5.1 Multi-Agent Pickup and Delivery (MAPD)

The multi-agent pickup and delivery (MAPD) is a problem that strictly relates to the multi-agent path finding one. While MAPF aims at finding a feasible path for all the agents in the system from a starting position to a desired destination, the MAPD problem introduces the option for all the agent to have to meet an intermediate position. Indeed, in an industrial environment, robots usually need to complete a task by starting from an initial position then to move to an intermediate one called "pickup" location and finally to reach their delivery location.
This introduces a remarkable difference from MAPF, which can be seen as a one-shot version of the problem [45].
Similarly to the MAPF , also the MAPD problem consists of m agents $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ and an undirected connected graph $G = (V, E)$, where V is the set of the vertices, i.e., the locations, and E is the set of the edges, i.e., the connections between locations that the agents can travel through. Let an agent a_i start from a location $l_i(0)$, then at each timestamp t, the agent either stays in the same position $l_i(t+1) = l_i(t)$ or it moves to a neighboring node $(l_i(t), l_i(t+1)) \in E$. Moreover, as for the MAPF problem, *vertex* and *swap* conflicts should be avoided, that is two agents a_i, a_j cannot be on the same node at the same time $l_i(t) \neq l_j(t)$ and they cannot move on the same edge at the same time in opposite directions $l_i(t+1) \neq l_j(t) \wedge l_j(t+1) \neq l_i(t)$, respectively.
The part that mainly differs from the MAPF problem is the fact that the agents need to complete *tasks*. A task τ_j can be defined as the tuple (s_j, g_j), where $s_j \in V$ is the pickup location and $g_j \in V$ is the destination to be reached. An agent is called *free* at a certain moment t if it is not executing any task, otherwise it is called *occupied* [45]. Notice that the agent is bound to a

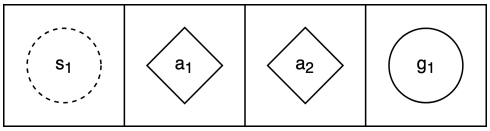


Figure 3.5: The figure shows a MAPD instance which is not solvable. The example shows an environment with two agents, a_1, a_2 and a task set $\mathcal{T} = \{\tau_1 = (s_1, g_1)\}$. The instance is not solvable because a_2 cannot move to s_1 without a vertex conflict and neither can a_1 reach g_1 without a vertex conflict.

given task from the moment it reaches the pickup location, i.e., if an agent a_i is associated with a task $\tau = (s, g)$, but has not reached s yet, then another agent a_j may take over, e.g., when a_j is closer to s than a_i is. Consider a task set $\mathcal{T}$ in which the system inserts all the unexecuted tasks, then the goal of the problem is to execute all the tasks $\tau \in \mathcal{T}$.
As presented up until now, the problem is as wide as possible. In this case, there are instances that are not solvable, for example the one shown in Figure 3.5. One possibility is to consider only well-formed, or *valid*, MAPD instances [46]. The idea is that agents can rest only on locations called *non-task endpoints*, that is locations from where no other agent is blocked.
One final distinction is done between *offline* and *online* MAPD problems. The former assumes that all the tasks are known *a priori* and no other task will be added to the task set during the execution of the agents. Instead, the latter means that new tasks may continue to arrive and that no information regarding the number of the tasks, nor their structure is known *a priori*.

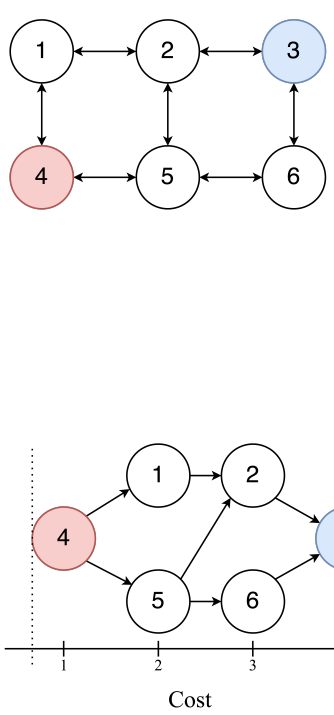
3.3.5.2 Non-unitary edges

As said in Section 3.3, classical MAPF considers an undirected graph with edges that have unitary cost, that is, the cost of an action is always one when computing the objective function, independently by the fact that the agent moved over an edge or stayed on a node. This introduces a constraint regarding the possible movements of the agents. Indeed, the main problem with having edges with different costs is the fact that agents may conclude their actions in different times causing only a portion of the action to overlap. Walker et al. proposed a solution showing great results when applied to the ICTS algorithm proposing two new variations called $\varepsilon-$ICTS and $\omega-$ICTS [47].
Both algorithms consider an agent either staying on the vertex or moving from one vertex to another traversing an edge at constant speed. A collision is defined as the condition for which one or more agents overlap at the same instant of time. The collision check is done by considering only agents with partial time overlaps and then by running a continuous-time conflict detection algorithm for moving agents [48].
Then, the multi-value decision diagram of the low-level search of ICTS needs to be changed to account for different costs and especially for the fact that non-unitary costs may lead to have multiple sink. For example, looking at Figure 3.6, it is possible to notice that a 4-connected grid[1] generates only one sink for the goal node at a given time, whereas if the graph was fully connected, then the possible sink in a given interval could be more than one. This problem depends on the value set to discretize the increment of time. Indeed, if the value is small, the possibility of finding multiple sinks in the graph in an interval of time is smaller, but at the same time the depth of the graph will increase. On the contrary, a larger value of the increment may lead to a higher probability of finding multiple sinks, but to a smaller graph.
Some changes had to be made also to the high-level search. In the original protocol, the nodes would keep track of only the cost of the paths for each agent. In the extended versions, each node

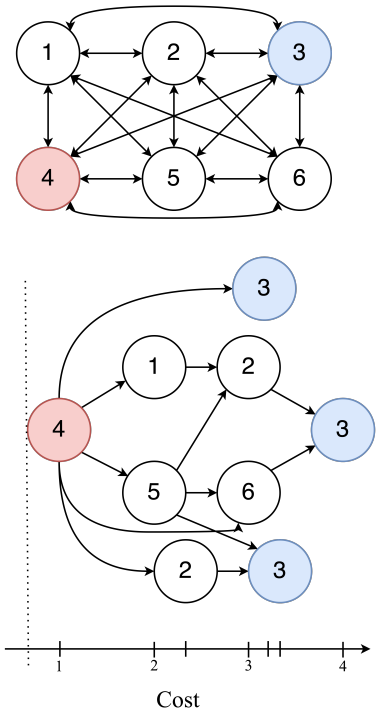
[1] A 4-connected grid is a grid where the nodes are connected only upwards, leftward, rightward and downwards

keeps track of a lower and an upper bound for each agent, other than the path. The low-level will not return the existence of a solution, but it will return the cost in the interval between the lower bound and the upper bound. The final solution is then obtained in a best-first manner. The proposed algorithms ε-ICTS and ω-ICTS provide the following improvements:

- ε-ICTS: instead of finding the optimal solution, the low-level search exits on the first feasible solution. This allows the algorithm to save time and space in the high-level, but at the cost of optimality.
- ω-ICTS: the value of the increment is obtained dynamically by computing a ratio between the weighted cost of the previous iteration and the number of agents.



(a) On the left, a standard MAPF problem where each edge has weight one.



(b) On the right, instead, we consider the variation in which the graph is fully connected, so there are some edges which weigh more than one. In particular, from the image on the right, it is possible to see that the cost increment is one, then in the interval $[3-4]$ there are 3 different sinks for the goal position. In case the cost increment is smaller, for example 0.5, the interval $[3, 3.5]$ contains two sinks, whereas the $[3.5, 4]$ interval contains only one sink.

Figure 3.6: A comparison between ICTS for the classical MAPF and how the MDD is built for non unitary edges.

4 Algorithms Descriptions

In this chapter, the main algorithms used to solve the Multi-Agent Path Finding (MAPF) are described. We first give a formal definition of the problem and how it differs from the classical multi-agent path finding problem. Then a description of the implementation in constraint programming is presented, showing promising results obtained by modelling the problem with constraints.
Then a state-of-the-art algorithm using imperative programming language is presented showing the revisions that had to be made in order to adapt it to the industrial scenario we were faced with.

4.1 Problem definition

The problem is the following:

Definition 4.1

Let $G = (V, E)$ an undirected graph with weighted edges. Consider k agents having to move from a set of k initial position S, one for each agent, to a set of k final positions, passing through an agent-variable number j_i of goal positions. The problem consists in finding the best joint plan Π that minimizes an objective function so that no conflict arises from the execution of such plan.
Moreover, we shall assume that the goal positions are known a priori and no new goal position is later added to the list.

The main difference with the classical MAPF problem is the fact that we introduces a Multi-Agent Pickup and Delivery flavor (albeit offline) since each agent is not associated only with a starting and arrival position, but also with a list of tasks it has to complete before reaching the final destination.
Another key difference is the fact that in our case-study the edges of the graph are weighted: indeed, we need to consider the cases in which some links from a point to another are longer than others.
The following symbols are used in the rest of the thesis:

- $\mathcal{A}$ or *agents*, the set of all the agents;
- $S = initPos$ the list of initial positions of the agents;
- $E = endPos$ the list of final positions of the agents;
- π_i a valid plan for an agent i;
- $|\pi_i|$ the length of the plan for agent i, i.e., the number of nodes traversed by agent i during the execution of its plan;

- $t_f = \texttt{M} = \max_{\pi_i \in \Pi} |\pi_i|$ the final timestamp of the execution which corresponds to the longest path between all agents;
- t_i the initial step that can either be 0 or 1 depending on the programming language starting index;
- $\mathcal{S} = \texttt{steps} = \{t_i ... t_f\}$ the set of the steps;
- $\mathcal{N} = \texttt{nodes}$ a set containing the nodes of the graph;
- $\texttt{neigh}(n)$ the set of neighbors connected to a node n;

4.2 Constraint Based Search (CBS)

As said CBS is a two levels algorithm for solving the MAPF problem in a correct and optimal way [30]. The work from Sharon et al. in which CBS was first proposed actually contains also a variant of the algorithm to speed up cases in which two or more agents are frequently colliding [41]. This variant is called Meta-Agent Conflict Based Search (MA-CBS) and it merges the agents that frequently conflict and considers them as a unique entity. This though implies the fact that the low-level search must already be a MAPF solver and not simply a SAPF solver. For this reason, this thesis focuses on the classical version of CBS.
We are going to first introduce two algorithms to solve the low-level search of CBS and then move to the description of the implementation of the high-level search.

4.2.1 Low-Level Search

In CBS, the low-level search is the algorithm responsible for finding a feasible path from an initial point to a final point on the graph for a single agent. We shall distinguish two situations: the first call to the low-level is for the creation of the root of the constraint tree, while all the other calls are for subsequent nodes. Each of the nodes, excluding the root, will have constraints to abide to, and hence the low-level algorithm must be modified to account for the presence of possible limitations.
Moreover, the majority of SAPF algorithms are one-shot in the sense that they do not provide support for additional intermediate goals to reach. As mentioned in Chapter 2, this thesis does not consider the problem of mission planning, but only the motion planning one, this means that we assume that the order in which the tasks are received is actually the best one hence the problem focuses on to the best way of combining the various point-to-point paths.
We shall now present two algorithms that have been adapted to solve the low-level search: the first is based on spanning trees, while the second is a time dependent shortest path algorithm based on Dijkstra's.

4.2.1.1 Spanning Tree

The main problem that this algorithm aims at solving raises from an observation of the difficulty in finding alternative paths when faced with constraints. For example, consider Figure 4.1: it is possible to follow two paths to go from vertex 1 to vertex 6:

$$\pi_1 = \{1, 2, 4, 7, 6\}$$
$$\pi_2 = \{1, 2, 3, 5, 8, 7, 6\}$$

Obviously, the second path is longer than the first one and it would never be selected if the first could be used. Let's suppose though that the agent cannot be on node 4 at times 3 and 4, then

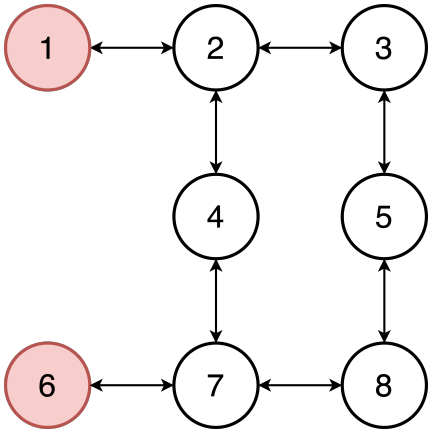


Figure 4.1: A possible example a SAPF problem that the low-level search must solve. The agent starts from vertex 1 and has to reach vertex 6.

the two paths start to have the same length.
`spanningTree` provides alternative paths in an easy way, so that the low-level search may select the best to pass to the high-level search. The algorithm works as described in Algorithm 2.
The point-to-point spanning tree function `spanningTreeP2P` works in the following way:

1. Initialize a set of nodes (`OPEN`) to be explored with their distance from the source and a list of paths (`paths`).
2. Initialize the distance of the first node.
3. Start from the source, add to `OPEN` all the neighbors of the source increasing the distance by one.
4. Then until `OPEN` has elements, pop the last element from the list and iterate with the same procedure as for the root adding the neighbors of the considered node to `OPEN` and adding each considered node to the current path.
5. At each iteration, the following checks are carried out:
   - If the element that was popped is the arrival node, then the path we have found until now is valid and can be stored.
   - If the considered node is already inside the path, then the path is a cycle and should not be considered, discard the path.
   - If the distance of the current node is less than the distance of the last node added to the path, then the path is not valid, should be dropped and the new node should be added to the path.

   Each time one of the conditions is true, either the path is dropped or it is stored. In the latter, a new path is added to `paths` starting from the last node that had more than one neighbor added to `OPEN`. When the former happens instead, the path is not added to `paths`, but we erase all the nodes up to (and excluded) the node that generated more neighbors in `OPEN`. To check which node is the one responsible for multiple neighbors, we check the last distance in `OPEN` and the position of the node to be erased inside its path.
6. When `OPEN` is empty, then we can return. Before returning we check if the last path that was being built has reached the final destination and if it has not, then it is removed from the list.

Algorithm 2: The pseudo code for the spanning tree algorithm

```
Input: nodes: the list of nodes
Input: initPos: the initial position
Input: endPos: the final position
Input: goalPos: the list of goal positions
Input: connect: the connectivity matrix

Data: finalPaths: a vector containing the paths that are valid

begin
    if goalPos is empty then
        return spanningTreeP2P between initPos and endPos
    else
        start ← initPos
        end ← goalPos[0]
        Insert the solution of spanningTreeP2P from start to end in finalPath
        for Each remaining goal in goalPos do
            start ← end
            end ← next goal
            Insert the solution of spanningTreeP2P from start to end in finalPath
        start ← last goal
        end ← endPos
        Insert the solution of spanningTreeP2P from start to end in finalPath
```

Another key aspect is how to merge the point-to-point paths to obtain a single path. Indeed this is a weak spot of the proposed algorithm since it is actually a combinatorial problem which may cause the memory usage to raise exponentially. Indeed, for every `spanningTreeP2P` invocation after the first, the previous vector of solutions is copied as many times as the number of the new solutions and each one of the solutions is added to the old ones.
The algorithm so described works for solving a SAPF problem with the addition of intermediate goals and it returns a vector of possible paths to reach the destination while passing through the goals. We now need to also consider the possible constraints from high-level. To do this, we first order the constraints following a temporal order, then we iterate through the paths and check whether a given path violates one or more constraints. If it does, then we insert a copy of a safe node to indicate that the agent must stay on the previous node to avoid the conflict. Finally, once we have gone through all the paths and fixed the possible conflicts, we are going to return the shortest path to the high-level search.
The important point of this problem is how to choose a "safe" location. Initially, the safe location is the starting node, i.e., index 0, then each time a constraint is fixed, the safe location moves to the time of the constraint. This allows to produce a path that is free from collisions. Notice that it is important that the constraints are temporally ordered, otherwise adding nodes in which the agent should stay may produce other violations.

4.2.1.2 Time-Dependant Shortest Path (TDSP)

This low-level search is mainly based on the algorithm from Dijkstra [26], but we introduce a new structure to handle the edges between nodes.

This structure is trivially named `Connection` and stores two important pieces of information:

- The type of connection, which can either be:
  - `ONE`: when the edge is always valid;
  - `ZERO`: when there is never an edge between two nodes;
  - `LIMIT_ONCE`: when the connection is always valid except for the specified time step(s);
  - `LIMIT_ALWAYS`: when the connection is never valid except for the specified time step(s).
- A vector of time steps, which can either be times in which the edge should not be considered, or times in which the edge is valid.

Before calling the shortest path algorithm, the low-level search transforms the connectivity matrix from integer values to objects of time `Connection` taking into consideration the constraints. This allows the algorithm to avoid a path if the edge is not valid in that time step as it would lead to a vertex or a swap conflict.
This alone though may not solve the problem, i.e., find a feasible path given the constraints, as all it does is stating that the agent should or not use a certain edge. In order for the agent to stay on a node, we need to add a virtual node, which we will refer to as placeholder. What the algorithm does is to see that at a given time t the agent cannot be on a node n, then it adds a placeholder for the node, it limits the connections of all the nodes that previously had a valid connection to the node n at time t and replaces this connections with a valid connection to the placeholder only for time t. This basically increases the path length of one unit and during the phase of post-processing it is possible to remove the placeholder and add a node where the agent can stay.
Instead, we consider swap conflicts as a double vertex conflict: if a node cannot go from a node n_i to a node n_j at time t, this means that at time $t+1$ it can neither be on n_i since it will be occupied from the other node, nor on n_j since it would cause a swap conflict.
The rest of the shortest path algorithm is dealt with in the following way:

- Compute the shortest path between the initial point and the first goal point;
- Iterate between the various goal points in list;
- Compute the shortest path between the last goal point to the final vertex.

The so obtained path is the shortest path (assuming correctness and optimality of the chosen shortest path algorithm) that meets all the goals and respects the constraints. In our case, the shortest path algorithm is the Dijkstra's algorithm which is optimal unlike A* that must prove that the used heuristic is admissible to be optimal.

4.2.2 High-Level Search

The high-level search is basically the same as the one described in Section 3.3.3. We start from a root node which does not contain any constraint and we compute the best path for each agent considering the agent as if it was in SAPF problem. Then we check for conflicts:

- if no conflict was found, then this is the solution to return;
- if a one or more conflicts were found, then only the first conflict is considered.

To check for conflicts, we compare each path with all the other paths. Each element in the vector representing the path corresponds to the node occupied by an agent at a certain time t, so the indexes of the vector correspond to the time steps. So when a vertex conflict is found, it happens because at the same index, both vectors have the same node, while when a swap conflict is found then the node of a vector at index t corresponds to the node at index $t + 1$ of the other vector and vice versa.
When a conflict is found, two constraints are produced:

- Vertex conflict: in one constraint, agent a_i cannot be on node n at time t, on the other constraint, agent a_j cannot be on node n at time t.
- Swap conflict: in one constraint, agent a_i cannot go from node n_1 to node n_2 at time t, on the other constraint, agent a_j cannot go from node n_2 to node n_1 at time t.

For each constraint, the low-level search is called: since the constraint affects only one agent, then only its path is going to be recomputed. If the low-level manages to find a solution, then the high-level adds a new node with the new solution and also adds the constraint to the constraints list of the new node. The search is repeated until a valid solution is found.

4.3 Constraint Programming

As said, Multi-Agent Path Finding (MAPF) problem is NP-hard [33] and so its possible to model it using constraint programming. Such technique has been deeply explored using different solvers for the constraints:

- Constraint Satisfaction Programming (CSP): a set of logical and mathematical statements that must satisfy a number of constraints [49];
- Boolean Satisfiability (SAT): a set of boolean formulas for which the solver tries to find a set of boolean variables such that the formulas are true, i.e., satisfiable [50].
- Mixed Integer Programming (MIP): a set of linear equations that must be solved while satisfying a list of constraints [30];

The first step was to take the model proposed by Barták et al. in 2017 [44] and implement it using CPLEX Optimization Programming Language (OPL) [51]. The goal of moving the model to another programming language was to exploit the straightforward similarity between the problem and MIP and also to exploit the optimal performance of IBM's solver.
Barták et al. used Picat, a logic programming language that allows the programmer to code a series of constraints as generally as possible and then to solve these by means of different solvers offered by the framework. Indeed, Picat can solve the coded constraints either with a SAT solver, or with MIP or also with a CSP solver. This actually allows for an easy and great comparison between the possible solvers, but in our study we focus on having a reliable, easy-to-implement and especially easy-to-maintain solver. Picat presents an initial steep learning curve, which does not comply with the easy-to-implement and easy-to-maintain requirements.
Instead, CPLEX has a much nice learning curve (largely but badly documented, Ed.) with an easy to pick up idea of how the solver works and a clear distinctions of the phases in which the programmer should divide their code. This is particularly noticeable if one starts writing the code using OPL. A CPLEX program structure is usually divided in:

- Pre-processing: we used this step to ensure that the input problem was well-defined;
- The input data: in OPL a `.dat` file is used to store the input variables;

- The objective function: CPLEX is able to minimize (or maximize) only one objective function, but for our use-case this is enough;
- A series of constraints to which the decision variables should be bound to;
- Post-processing: a series of instructions which conclude the program, usually used to print and check the solution.

We are now going to describe this steps.

4.3.1 Variables

A variable in CPLEX can either be an input variable or a *decision variable*, which is a variable whose value should be computed by the solver.
The input variables in our cases are the following:

- `n_agents`: the number of agents in the problem;
- `n_nodes`: the number of nodes in the graph;
- `n_steps`: the number of steps in which the problem is supposed to be resolved;
- `connect[n_nodes][n_nodes]`: a non negative connectivity matrix of the nodes;
- `n_goals[n_agents]`: a vector containing the number of goals each agent has to complete;
- `max_goals`: the maximum number of goals an agent must reach;
- `init_pos[n_agents]`: a vector containing the initial positions of all the agents;
- `end_pos[n_agents]`: a vector containing the final positions that each agent should reach;
- `goal_pos[n_agents][max_goals]`: a list of lists of goal positions, one list for each agent.

Notice that it is not possible to know `n_steps` a priori, hence its value is usually a simple estimation of what the actual solution would be. The value is refined during the execution: whenever a model is not solvable, it is incremented by one to check whether it is possible to solve the scenario with more time. The value is incremented until a solution is found.
To solve the MAPF problem we use multiple decision variables. The primary one is the `X[n_steps][n_nodes][n_agents]`, which is a boolean three dimensional matrix that keeps track of whether an agent a_i is on node n at time t.
To keep track of whether the agents have reached or not their goals, another three-dimensional matrix is created: `goal_points[n_agents][max_goals][n_steps]` that states whether an agent a_i has reached a certain goal g at a certain moment t.
Finally, we need to keep track of the movements of the agents so that we can then minimize the objective function of choice. This is done by using another matrix, `movement[n_agents][n_steps]` that memorizes whether an agent a_i at time t has moved or not.
One final structure is needed to avoid swap conflicts, but will be discussed later on.

4.3.2 Objective Function

The objective function that was initially sought was the minimization of the sum of the movements of all the agents in the graph, i.e., minimizing the *sum of costs* [44]. To do this in CPLEX one could simply write:

```
minimize sum(a in agents, s in steps) movement[a][s];
```

which corresponds to just minimizing the sum of all the cells in the movement matrix. We will discuss later about the implementation in C++ of the model and how it is possible to also minimize for the makespan.

4.3.3 Constraints

The constrained defined by Barták et al. [44] are being reused:

- Each agent a cannot be on more than one node at each time step:

$$\forall s \in \mathcal{S},\ \sum_{n \in \mathcal{N}} X[s][n][a] = 1 \tag{4.1}$$

- Each node n cannot be occupied by more than one agent at each time step:

$$\forall s \in \mathcal{S},\ \sum_{a \in \mathcal{A}} X[s][n][a] \leq 1 \tag{4.2}$$

- If an agent a occupies a node n at time s, it will occupy a neighboring node m at time $s+1$:

$$X[s][n][a] = 1 \quad \Rightarrow \sum_{m \in \texttt{neigh}(n)} X[s+1][m][a] = 1 \tag{4.3}$$

The above constraints do not bind the initial and final position of the agents. Indeed, in the code proposed in [44], the assignment of the initial and final position for each agent is done via preprocessing, but this was not possible with OPL since the decision variables cannot be directly modified. Two constraints were then introduced:

$$\forall a \in \mathcal{A},\ X[t_0][S_a][a] = 1 \tag{4.4}$$

$$\forall a \in \mathcal{A},\ X[t_f][E_a][a] = 1 \tag{4.5}$$

4.3.3.1 Swap Conflict

Although the swap conflict was not taken into consideration due to the increased computational cost as stated in [44], in this work, it is important to avoid this type of conflict[1]. A first constraint to avoid a swap conflict is the following:

$$\begin{aligned} &\forall s \in \mathcal{S}/\{t_f\},\ \forall a_i \in \mathcal{A},\ \forall a_j \in \mathcal{A}/\{a_i\},\ \forall n_i \in \mathcal{N},\ \forall n_j \in \mathcal{N}/\{n_i\}, \\ &X[s][n_i][a_i] \wedge X[s][n_j][a_j] \Rightarrow X[s+1][n_j][a_i] \wedge X[s+1][n_i][a_j] \end{aligned} \tag{4.6}$$

This though increases a lot the time required to find a solution.
The best alternative seems to follow the work from Barták and Šancara in 2019 [52], who propose to use another structure to keep track of the edges occupation. For this reason the following three dimensional matrix is used: `edges[n_nodes][n_nodes][n_steps]` which keeps track of whether at time step s the edge from node n_i to node n_j is being used or not.
The constrain is then so defined:

$$\forall s \in \mathcal{S}/\{t_f\},\ \forall n_i \in \mathcal{N},\ \sum_{n_j \in \mathcal{N}/\{n_i\}} \texttt{edges}[n1][n2][s] + \texttt{edges}[n2][n1][s] \leq 1 \tag{4.7}$$

[1] We would like to make known that also a constructive email exchange has been carried out with the original authors of the Picat algorithm, highlighting this aspect and a possible solution.

Barták and Šancara noticed that the model is easier to be solved while using easier constraints and more variables, than while using less variables and more difficult constraints. This is possible imputable to the fact that the solver needs to internally transform the constraints into decision variables and a constraint that is more complex provides more decision variables than a simpler constraint with a priori defined decision variables.

4.3.3.2 Tracking Movements

Since we need to minimize the objective function, we need to memorize the movements that are done in the simulation. This is done by using the following constraint:

$$\begin{gathered}\forall s \in \mathcal{S},\ \forall a \in \mathcal{A},\ \forall n_i \in \mathcal{N},\ \forall n_j \in \mathcal{N}, \\ X[s-1][n_i][a] \wedge X[s][n_j][a] \Rightarrow \texttt{movement}[a][s] = \texttt{connect}[n_i][n_j] \wedge \texttt{edges}[n_i][n_j][s]\end{gathered} \tag{4.8}$$

This constraints basically checks if the agent has moved, and if it has, then it assigns the cost of the edge to the movement matrix.

4.3.3.3 Goal Reaching

Another aspect that was not taken into consideration in previous works [44, 52] is the absence of intermediate goals that the agents have to meet before reaching the final destination. For this reason, the following constraints were added:

$$\forall a \in \mathcal{A},\ \forall s \in \mathcal{S},\ \sum_{g \in \texttt{n\_goals}[a]} \texttt{goal\_points}[a][g][s] \Rightarrow x[s][\texttt{goal\_pos}[a][g]][a] \tag{4.9}$$

Which means that if the goal has been taken in a given movement s from a agent a, then the agent must be in said position n that time step s.
We also need to make sure that an agent takes at most one goal in each timestamp and that also each goal is taken at most once:

$$\forall s \in \mathcal{S},\ \forall a \in \mathcal{A},\ \sum_{g \in \{1,\ldots,\texttt{n\_goals[a]}\}} \texttt{goal\_points}[a][g][s] \leq 1 \tag{4.10}$$

$$\forall a \in \mathcal{A},\ \forall g \in \{1,\ldots,\texttt{n\_goals[a]}\},\ \sum_{s \in \mathcal{S}} \texttt{goal\_points}[a][g][s] \leq 1 \tag{4.11}$$

Finally, constraint from Equation 4.5 must be modified in order to account for the fact that the agents cannot reach the final destination without having taken the goals:

$$\forall a \in \mathcal{A} \sum_{\substack{s \in \mathcal{S} \\ g \in \{1,\ldots,\texttt{n\_goals}[a]\}}} \texttt{goal\_points}[a][g][s] = \texttt{n\_goals}[a] \wedge X[t_f][E_a][a] \tag{4.12}$$

4.3.4 Pre-processing, main and Post-processing

OPL allows also to have a main function that can be used to create the model, solve it and analyze whether it was possible to find a feasible solution or not. This is particularly useful to the MAPF problem since it is not possible to know a priori which are the correct lengths of the paths to follow. The idea is to start with a lower bound on the number of steps of the solution and increase the value each time the solver could not find a solution until one could be found. Will discuss later with the C++ implementation of a better way to estimate the lower bound.
The post-processing instead is basically just a way to output information about the found solution.

4.3.5 Python and C++ Implementations

Since CPLEX offers the possibility to integrate the solver in Python or C++, we tried to develop a more portable version of the software so that it could run on multiple platforms without requiring the IDE of OPL . Moreover, having the possibility to run the constraint programming code in C++ or in Python would allow us to test the performance far more easily.
The code was first moved to Python resulting in a quite seamless transition from CPLEX. Once the *docplex* library was imported, the data structures used to input data are basically the same and the creation of the model is straightforward. Also the decision variables have a class that clearly states that they are used as decision variables reducing the confusion. In same way, declaring constraints for the model using the given functions is easy. Indeed, the syntax of the majority of the code stayed almost the same, leading to the hope that maybe the solver called from Python was the same called from the IDE of OPL . Unfortunately we were wrong and the performance dropped dramatically.
We then decided to move to C++ to check whether the performance could improve. The CPLEX framework for C++ is called *Concert* and it suffices to say that moving the code-base to comply with it was not as immediate as it was with Python. With Concert we first need to describe a `IloEnv` environment in which the variables will be stored. We can then create the model and the CPLEX solver. The types offered by Concert do not have a clear distinction between decision variables and input (or data) variables. Moreover, in some cases the name of the types is even misleading, e.g., constraints are declared with type `IloRange` that is quite similar to `IloIntRange` which is used to define an interval of integers[2], or, for example there is not a function similar to how `sum` works in OPL IDE or of how `model.sum()` works in Python, but instead the sum over a given set of variables must be done by using an auxiliary variable declared as `IloExpr` and then incremented in a loop. The last weak point of Concert is the fact that accessing the variables to change their values before repeating the solver is not directly possible, at least to our knowledge Instead one needs to free both the solver, the model and the environment each time a variable needs to be changed to solve the model again. That said, moving the code to C++ proved to be a good decision in terms of performance.
Moreover, doing so allows us to use the shortest path algorithm shown in Section **??** to provide a lower bound in the pre-processing phase of the program reducing by a lot the number of needed iterations to reach a solution.
Finally, both the implementations in Python and C++ allow to minimize not only the sum of individual costs function, but also to decide to minimize the makespan. Yet again, in Python this is done in a straightforward way by using the function `model.max()` to find the maximum length between the paths, while in C++ one needs to create the function manually. To note though, that as in CPLEX, it is not possible to have two objective functions at once.

[2]After some time and a little bit of research, one understands that constraints seem to be extracted as a series of variables and hence the type `IloRange`

5 Test Discussion

To test the quality and the performance of our algorithms, a series of tests was created. We start by first talking about a list of random tests that were performed to assess the algorithms and then we give a description of the industrial warehouse and how it has been divided for the tests.

5.1 Random Tests

The tests were randomly created and saved to a JSON file, later taken in input by the tested program. They were created to be of increasing difficulty. The aspects that play an important role in setting the complexity of a Multi-Agent Path Finding (MAPF) problem are:

- The number of nodes: the bigger the graph we are considering, the higher the number of possible paths that solve the problem. Considering the complexity of Dijkstra's algorithm being $\Theta\left((|E| + |V|) \log |V|\right)$, this means that a larger number of nodes leads to a larger computational time.

- The number of goals: a bigger number of goals to take before reaching the destination implies a larger simulation time and hence a larger number of possible solutions. Indeed, if we consider the Multi-value Decision Diagram (MDD) of the mission, which corresponds to its time expanded graph, it is easy to observe the fact that a longer simulation time leads to a more complex solution.

- The number of agents: the higher this value is, the more complicated the solution is. Actually the number of agents does not matter as much as the occupancy w.r.t. the number of nodes. Indeed, if we had 100 agents in a graph of 1 million nodes, the ration 1:10000 would not be as much a problem as having 100 agents in a graph with 1 thousand nodes. This is due to the fact that the higher the occupancy is, the higher the probability of collisions, which then leads to having a much deeper Constraint Tree (CT) at the high-level search. In this cases, we would be better using sub-optimal algorithms, or more complex algorithms such as MA-CBS [41].

One thing to notice is that in the process of generating possible random tests, we used directed graphs instead of undirected ones in order to create a more challenging environment. In this case, another interesting factor that can make the MAPF problem more complex is the number of neighbors each node has. Indeed, a very sparse graph may lead to problems such as sinks, which are basically dead ends, from which an agent cannot return, while a higher number of neighbors implies a higher number of possible paths to consider. In this second case, if the scenario was a fully connected graph, one would be better off using an heuristic algorithm such as A* to avoid having to look at all the possible paths. To avoid the problem of having sinks in the test-case, we required the graph to be strongly connected so that there could be a path between all pairs of nodes.

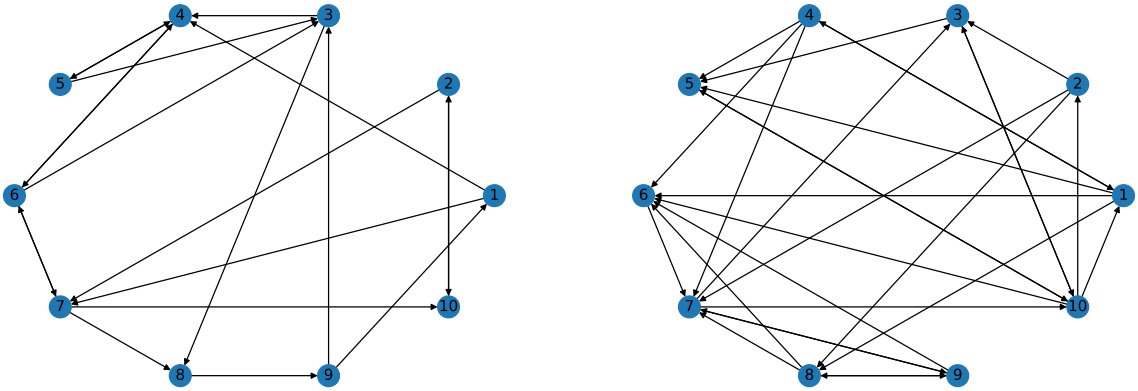


Figure 5.1: In this figure, some random tests are shown that were generated with 10 nodes and an increasing probability of finding a connection between two nodes (0.1 on the left and 0.25 on the right).

The tests were created by increasing the previous values (agent occupation, number of nodes, number of goals and connectivity of the graph) and randomly sampling the initial, final and goal positions of the nodes, and also whether an edge between two nodes was present or not.
One more constraint we placed on the generation of the test graphs is the fact that final positions cannot be a goal position, which is congruent with reality: usually the final positions are the charging stations of the robots. Some random generated tests are shown in Figure 5.1.

5.1.1 Warehouse tests

In Figure 5.2 we show the whole warehouse and the network that was used to connect the various parts it is composed of.
Then, we have broken down the schema in two easier problems to handle, the first called `MAG1`, which is the part on top right of the warehouse and that is shown in Figure 5.3. The second part is called `MAG2` and is the bottom left part of the warehouse, shown in Figure 5.4. Differently from the random generated tests, the warehouse tests use an already built graph, so there is no randomness in how the graph is constructed. The tests were then created by changing the number of agents and the number of goals in the test. Obviously, as for the random tests, also in these cases, the initial, final and goal positions were randomly extracted from the set of nodes with the limitation that a goal position cannot be a final position.
In addition to `MAG1`, `MAG2` and the whole warehouse (namely `MAG12`), we obtained 6 more graphs from the map to check how the dimension of the graph impacts on the performances:

- `MAG2_1`: the red rectangle in Figure 5.4;
- `MAG2_2`: the green rectangle in Figure 5.4;
- `MAG2_1_1`: the blue part of `MAG2_1`, i.e., the first 4 rows of nodes of the red rectangle, in Figure 5.4;
- `MAG2_1_2`: the grey part of `MAG2_1`, i.e., the first 4 rows of nodes of the red rectangle, in Figure 5.4;
- `MAG2_2_1`: the blue part of `MAG2_2`, i.e., the first 4 rows of nodes of the green rectangle, in Figure 5.4;

- `MAG2_2_2`: the grey part of `MAG2_2`, i.e., the first 4 rows of nodes of the green rectangle, in Figure 5.4;

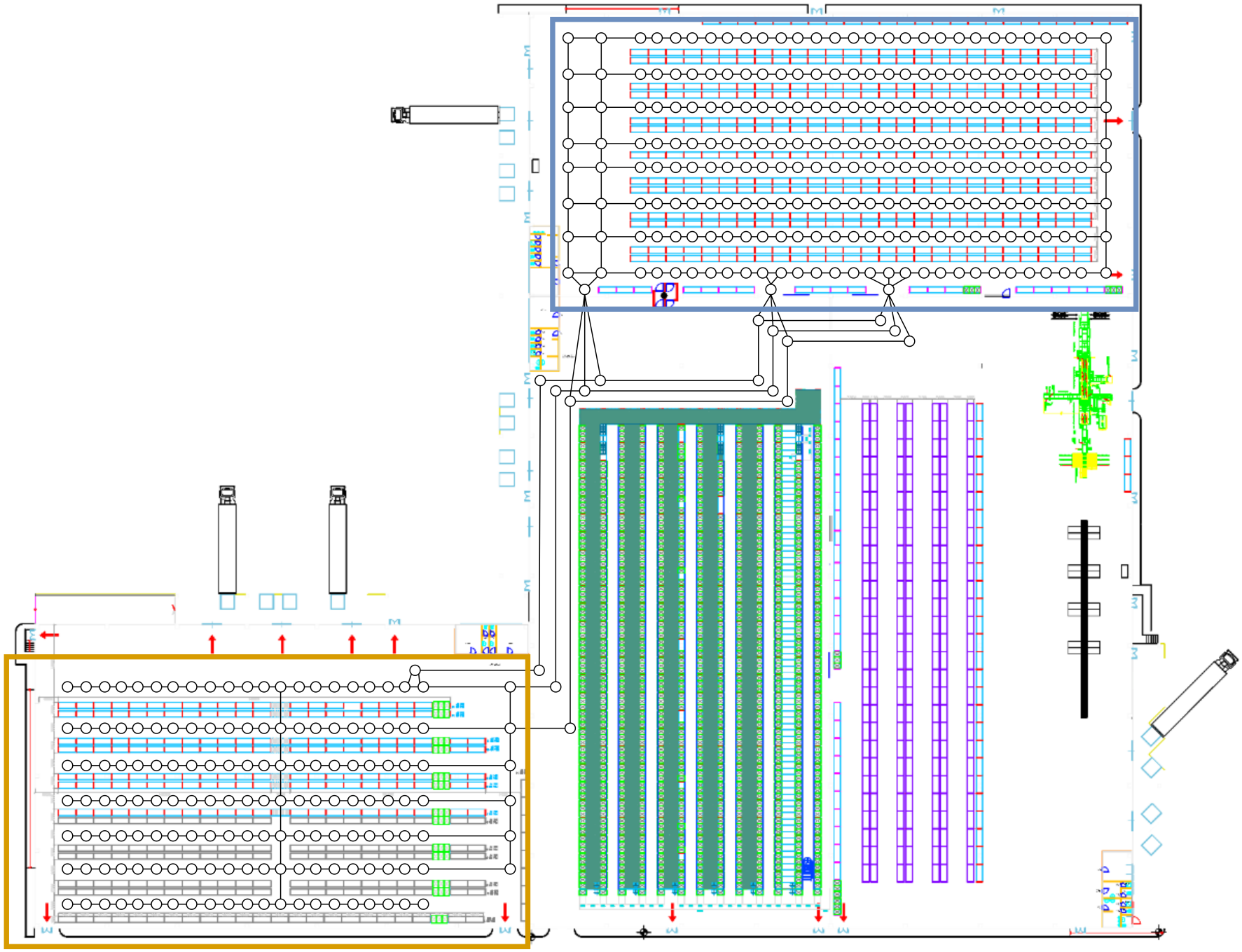


Figure 5.2: The schema for the whole warehouse with the superimposed road network used in the tests. `MAG1` is the portion squared in blue, while `MAG2` is the one squared in yellow.

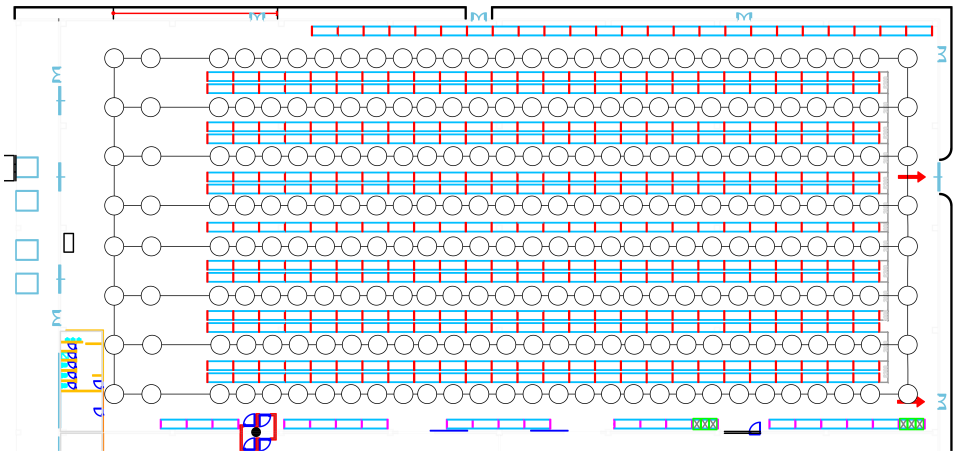


Figure 5.3: The first portion of the warehouse to be analyzed.

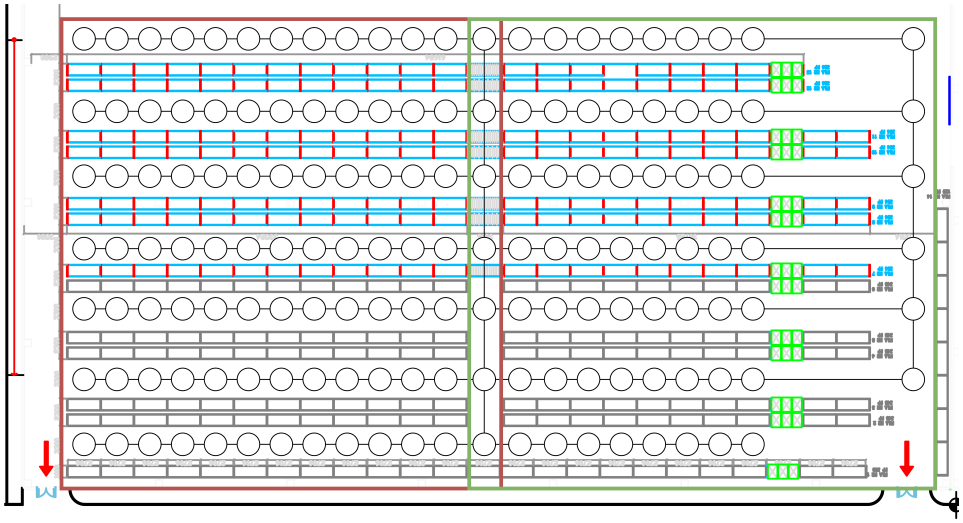


Figure 5.4: The second portion of the warehouse to be analyzed.

6 Results Comparison

In this chapter, we are going to compare the results obtained after running the tests. Then we give a description of some hard cases that the presented algorithms fail to solve and we conclude with a brief description of how they may be addressed.

6.1 Results

We created 300 random tests with different values of number of nodes, number of agents, number of goals and connection density, taken from:

```
agents_occ = [5, 10, 20, 50, 75]  # Percentages of agents w.r.t. #nodes
nodes = [10, 20, 50, 100]         # Number of nodes in the scenario
n_goals = [1, 2, 5, 10, 20, 50]   # Number of goals in the scenario
p_conn = [0.1, 0.25, 0.5]         # Probability that an edge between two nodes exists
```

The specifics of the computer used to run the tests are the following:

- Processor: AMD Ryzen 3700X;
- RAM: 32GB 3600MT/s;
- Operating system: Arch Linux with Linux Kernel 5.18.8;
- CPLEX version 22.1.0;
- C++ compiler: `gcc` version 12.1.0, code compiled with standard C++17.

The tests were then run multiple times to ensure the reliability of the data collected. Each test was run with the different versions of the algorithms:

- `CBS_TDSP_MKS`: CBS with TDSP for the low-level search and makespan for the objective function;
- `CBS_TDSP_SIC`: CBS with TDSP for the low-level search and SIC for the objective function;
- `CBS_ST_MKS`: CBS with spanning trees for the low-level search and makespan for the objective function;
- `CBS_ST_SIC`: CBS with spanning trees for the low-level search and SIC for the objective function;
- `CP_MKS`: constraint programming minimizing the makespan;
- `CP_SIC`: constraint programming minimizing the SIC;

Also, the executions were sequentialized through Python using the `subproces` module and, to avoid infinite loops when it is not possible to find a solution, we used the Unix `timeout` system call to kill the program after a certain amount of time. We used three different periods of time for the timeout in order to test whether some tests would require only more time or if they could not be solved. We have actually noticed an increment in the solved tests as it is possible to see from Table 6.1. In particular, we noticed that the constraint programming approach requires a lot of time to solve its inputs and this actually agrees with the findings of other authors [30].
It is worth noticing that the majority of the cases in which CT fails is due to the long computational times. Indeed, CT basically tries to find all the possible solutions that are valid with the given constraints and that minimize the objective function. The problem though is combinatorial in the number of decision variables and hence at each incremented step, it becomes more and more demanding to complete.

Table 6.1: The results of the program when faced with 300 random tests of increasing difficulty.

	Test completed			Average run time [ms]		
Timeout / Program	1s	10s	60s	1s	10s	60s
CBS_TDSP_MKS	187	189	189	15.86	39.93	40.51
CBS_TDSP_SIC	184	185	185	13.32	59.78	56.98
CBS_ST_MKS	30	34	35	24.26	398.02	1459.09
CBS_ST_SIC	27	28	29	37.45	123.25	381.41
CP_MKS	21	36	36	362.09	2086.08	2234.65
CP_SIC	17	28	28	273.51	1594.02	1501.23

The warehouse results are show from Table 6.2 to Table 6.10. What appears evident is that the algorithm TDSP presented in Section 4.2.1.2 is the one that is able to solve the most number of tests out of all the others and especially in the lowest time possible.
Moreover, we see that CP takes a lot of time to compute a feasible solution, and it also needs a lot of memory, albeit not reported in the tables due to difficulties in accurately stating the memory consumption.

Table 6.2: The results for the `MAG12` map over which 25 scenarios were tested with different numbers of goals and agents.

	Test completed			Average run time [ms]		
Timeout / Program	1s	10s	60s	1s	10s	60s
CBS_TDSP_MKS	12	15	16	207.87	950.47	1857.94
CBS_TDSP_SIC	12	14	15	208.28	524.43	1529.46
CBS_ST_MKS	0	0	0	–	–	–
CBS_ST_SIC	0	0	0	–	–	–
CP_MKS	0	0	0	–	–	–
CP_SIC	0	0	0	–	–	–

Table 6.3: The results for the `MAG1` map over which 20 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
`CBS_TDSP_MKS`	13	13	13	117.59	118.13	117.43
`CBS_TDSP_SIC`	12	12	12	124.19	124.25	125.51
`CBS_ST_MKS`	0	0	0	–	–	–
`CBS_ST_SIC`	0	0	0	–	–	–
`CP_MKS`	0	0	0	–	–	–
`CP_SIC`	0	0	0	–	–	–

Table 6.4: The results for the `MAG2` map over which 20 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
`CBS_TDSP_MKS`	10	11	11	72.78	263.56	262.04
`CBS_TDSP_SIC`	10	11	11	73.07	185.24	186.90
`CBS_ST_MKS`	0	0	0	–	–	–
`CBS_ST_SIC`	0	0	0	–	–	–
`CP_MKS`	0	0	0	–	–	–
`CP_SIC`	0	0	0	–	–	–

Table 6.5: The results for the `MAG2_1` map over which 20 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
`CBS_TDSP_MKS`	9	9	9	27.53	27.9	28.13
`CBS_TDSP_SIC`	9	9	9	27.43	27.6	27.63
`CBS_ST_MKS`	1	1	1	0.55	0.56	0.56
`CBS_ST_SIC`	1	1	1	0.96	0.55	0.96
`CP_MKS`	0	0	0	–	–	–
`CP_SIC`	0	0	0	–	–	–

Table 6.6: The results for the MAG2_2 map over which 20 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
CBS_TDSP_MKS	13	13	13	7.50	7.54	7.49
CBS_TDSP_SIC	14	14	14	32.93	32.73	32.67
CBS_ST_MKS	1	1	1	4.57	4.58	4.57
CBS_ST_SIC	2	2	2	43.00	42.74	42.74
CP_MKS	0	0	1	–	–	47782.30
CP_SIC	0	0	0	–	–	–

Table 6.7: The results for the MAG2_1_1 map over which 15 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
CBS_TDSP_MKS	7	7	7	8.13	8.08	8.26
CBS_TDSP_SIC	7	7	7	7.32	7.59	7.28
CBS_ST_MKS	2	2	2	9.21	10.28	9.13
CBS_ST_SIC	2	2	2	9.04	9.11	8.94
CP_MKS	0	0	0	–	–	–
CP_SIC	0	0	0	–	–	–

Table 6.8: The results for the MAG2_1_2 map over which 15 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
CBS_TDSP_MKS	5	5	5	28.86	29.43	29.10
CBS_TDSP_SIC	5	5	5	29.16	28.79	29.06
CBS_ST_MKS	0	0	0	–	–	–
CBS_ST_SIC	0	0	0	–	–	–
CP_MKS	0	0	0	–	–	–
CP_SIC	0	0	0	–	–	–

Table 6.9: The results for the MAG2_2_1 map over which 12 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
CBS_TDSP_MKS	4	4	4	1.73	1.69	1.79
CBS_TDSP_SIC	4	4	4	1.88	1.91	1.82
CBS_ST_MKS	1	2	2	0.80	4554.97	2.74
CBS_ST_SIC	1	3	3	303.65	3039.31	6.48
CP_MKS	0	0	0	–	–	–
CP_SIC	0	0	0	–	–	–

Table 6.10: The results for the MAG2_2_2 map over which 12 scenarios were tested with different numbers of goals and agents.

Program \ Timeout	Test completed			Average run time [ms]		
	1s	10s	60s	1s	10s	60s
CBS_TDSP_MKS	5	5	5	181.13	184.19	184.30
CBS_TDSP_SIC	4	4	4	0.37	0.50	0.40
CBS_ST_MKS	2	2	2	2.41	2.55	2.61
CBS_ST_SIC	2	2	2	3.26	3.47	2.66
CP_MKS	0	2	2	–	4754.48	4737.85
CP_SIC	0	2	2	–	3227.03	3244.67

6.2 Hard Scenarios

By analyzing some of the failing cases we notice a pattern: each time two agents follow the same shortest path in opposite directions and need to swap places at an edge, the algorithms fail to efficiently solve the problem. An example of this problem is shown in Figure 6.1 where agent 1 (in blue) starts from node 1 and has to reach node 6, while agent 2 (in red) starts from node 7 and has to reach node 1. In this case, the shortest paths, and also only paths, to go from node 1 to 6 and from 7 to 1 are the following:

```
1 : 1 2 4 6
2 : 7 6 4 2 1
```

The problem is that none of the agents can wait on a node of the shortest path without creating a conflict. The solution would be to consider additional nodes that are not part of the shortest path, so that an agent could move aside while the other freely passes. This is not something that is naturally done by neither spanning trees nor shortest path algorithms, since it would require to move twice on the same node.
The idea to solve this problem in TDSP is similar to the idea of how vertex conflicts have been handled so far, and that is by using a placeholder. Indeed, at the moment the virtual nodes are used as a way to say to the algorithm that the agents has to wait on a node to avoid a conflict, but they may well be used as a way to tell the agent to move to a neighboring node and then move back to the previous node. In this case, to avoid problem with the shortest path algorithm, we could use two placeholder, one for the neighboring node and one for the original node, which otherwise would not be considered. This though raises the problem of how to choose the neighboring node: in the test we have carried out, no good solution on this choice

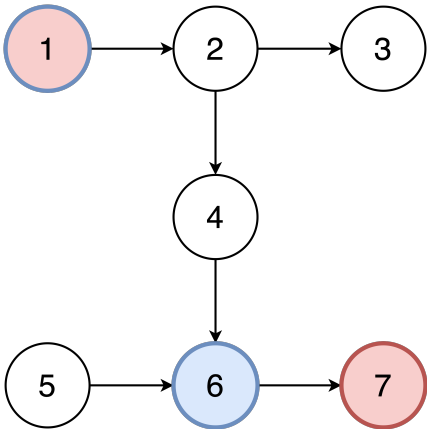


Figure 6.1: A scenario which is not easily solvable with CBS. In blue are the initial (1) and final (6) nodes of hte first agent

has been found. The best solution would be to return a list of possible paths instead of a single one, but this would also imply that the high-level search should carry multiple paths weighing the process down. Indeed, it is not possible to say a priori that choosing a neighbor instead of another is the best idea in terms of future conflicts.
Instead, the spanning tree algorithm may be improved by considering a counter keeping track of how many times the algorithm has moved over some nodes. For example, if we know that at time t there would be a swap conflict between node n_1 and node n_2, then we could let the algorithm move multiple times over n_1, which would allow the neighboring nodes to be explored. This though is part of the future work on the MAPF aiming at creating a framework for solving the problem with a wide range of possible algorithms.

7 Conclusions

We have carried out a formal comparison of state-of-the-art algorithms. From these, we have chosen three algorithms with the optimality criterion in mind, and we also wanted to keep the implementation light so that they could be easily maintained. We have shown their implementation providing an accurate description of how they work.
Finally, two sets of tests have been devised, one based on random scenarios regarding the MAPF problem, and one based on scenarios taken from a real warehouse. The results of the tests show that the algorithms as they are need to be improved. Indeed, they fail to provide a solution in many cases, with CP taking too much time and CBS failing to find a possible solution when using spanning trees. As stated, sub-optimal and more efficient versions of the proposed algorithms exist, especially for CBS, but they imply major revisions of the code in many cases introducing heuristics and sub-optimality, which was outside the scope of the project
The future work will focus on improving this existing algorithms in order to handle corner cases that are difficult to solve. Moreover, we would like to explore the combination of CBS and CP. Indeed, CP fails to find solutions especially for time and memory limitations, but these problems could be greatly reduced when the constraints are thought for SAPF and not for a MAPF problem. Solving only a small area of the graph with CP for a certain agent given some constraint could speed up the CBS process a lot and provide an optimal solution.

Bibliography

[1] S. Bragança, E. Costa, I. Castellucci, and P. M. Arezes, "A brief overview of the use of collaborative robots in industry 4.0: Human role and safety," 2019.

[2] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Magazine*, vol. 29, p. 9, 3 2008.

[3] B. Ochiai, "A new approach to predictive maintenance challanges," 2 2022.

[4] M. Javaid, A. Haleem, R. P. Singh, and R. Suman, "Substantial capabilities of robotics in enhancing industry 4.0 implementation," *Cognitive Robotics*, vol. 1, pp. 58–75, 2021.

[5] P. Brett, R. Taylor, D. Proops, C. Coulson, A. Reid, and M. Griffiths, "A surgical robot for cochleostomy," pp. 1229–1232, IEEE, 8 2007.

[6] F. Ferrari, S. Divan, C. Guerrero, F. Zenatti, R. Guidolin, L. Palopoli, and D. Fontanelli, "Human–robot interaction analysis for a smart walker for elderly: The acanto interactive guidance system," *International Journal of Social Robotics*, vol. 12, pp. 479–492, 5 2020.

[7] R. D. Arnold, H. Yamaguchi, and T. Tanaka, "Search and rescue with autonomous flying robots through behavior-based cooperative intelligence," *Journal of International Humanitarian Action*, vol. 3, p. 18, 12 2018.

[8] M. M. Veloso, J. Biswas, B. Coltin, and S. Rosenthal, "Cobots: Robust symbiotic autonomous mobile service robots," pp. 4423 – 4429, 7 2015.

[9] J.-C. Latombe, *Robot motion planning*, vol. 124. Springer Science & Business Media, 2012.

[10] B. Brumitt and A. Stentz, "Dynamic mission planning for multiple mobile robots," pp. 2396–2401, IEEE.

[11] O. Purwin, R. D'Andrea, and J.-W. Lee, "Theory and implementation of path planning by negotiation for decentralized agents," *Robotics and Autonomous Systems*, vol. 56, pp. 422–436, 2008.

[12] M. Erdmann and T. Lozano-Pérez, "On multiple moving objects," *Algorithmica*, vol. 2, pp. 477–521, 11 1987.

[13] J. P. van den Berg and M. H. Overmars, "Prioritized motion planning for multiple robots," pp. 430–435, 2005.

[14] A. Komenda, J. Vokřínek, M. Čáp, and M. Pěchouček, "Developing multiagent algorithms for tactical missions using simulation," *IEEE Intelligent Systems*, vol. 28, pp. 42–49, 2013.

[15] M. Frego, P. Bevilacqua, E. Saccon, L. Palopoli, and D. Fontanelli, "An iterative dynamic programming approach to the multipoint markov-dubins problem," *IEEE Robotics and Automation Letters*, vol. 5, pp. 2483–2490, 4 2020.

[16] E. Saccon, P. Bevilacqua, D. Fontanelli, M. Frego, L. Palopoli, and R. Passerone, "Robot motion planning: can gpus be a game changer?," pp. 21–30, IEEE, 7 2021.

[17] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," *The International Journal of Robotics Research*, vol. 17, pp. 760–772, 7 1998. doi: 10.1177/027836499801700706.

[18] M. Boldrer, A. Antonucci, P. Bevilacqua, L. Palopoli, and D. Fontanelli, "Multi-agent navigation in human-shared environments: A safe and socially-aware approach," *Robotics and Autonomous Systems*, vol. 149, p. 103979, 2022.

[19] D. Claes and K. Tuyls, "Multi robot collision avoidance in a shared workspace," *Autonomous Robots*, vol. 42, pp. 1749–1770, 2018.

[20] P. Bevilacqua, M. Frego, D. Fontanelli, and L. Palopoli, "Reactive planning for assistive robots," *IEEE Robotics and Automation Letters*, vol. 3, pp. 1276–1283, 4 2018.

[21] A. Antonucci, G. P. R. Papini, P. Bevilacqua, L. Palopoli, and D. Fontanelli, "Efficient prediction of human motion for real-time robotics applications with physics-inspired neural networks," *IEEE Access*, vol. 10, pp. 144–157, 2022.

[22] A. Bajcsy, S. L. Herbert, D. Fridovich-Keil, J. F. Fisac, S. Deglurkar, A. D. Dragan, and C. J. Tomlin, "A scalable framework for real-time multi-robot, multi-human collision avoidance," pp. 936–943, IEEE, 5 2019.

[23] M. Pouke, "Using gps data to control an agent in a realistic 3d environment," pp. 87–92, IEEE, 9 2013.

[24] S. Bhattacharya, M. Likhachev, and V. Kumar, "Topological constraints in search-based robot path planning," *Autonomous Robots*, vol. 33, pp. 273–290, 10 2012.

[25] B. Cohen, S. Chitta, and M. Likhachev, "Single- and dual-arm motion planning with heuristic search," *The International Journal of Robotics Research*, vol. 33, pp. 305–320, 11 2013. doi: 10.1177/0278364913507983.

[26] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 12 1959.

[27] H. Wang, Y. Yu, and Q. Yuan, "Application of dijkstra algorithm in robot path-planning," pp. 1067–1069, IEEE, 7 2011.

[28] Y. zhou Chen, S. fei Shen, T. Chen, and R. Yang, "Path optimization study for vehicles evacuation based on dijkstra algorithm," *Procedia Engineering*, vol. 71, pp. 159–165, 2014.

[29] G. Qing, Z. Zheng, and X. Yue, "Path-planning of automated guided vehicle based on improved dijkstra algorithm," pp. 7138–7143, IEEE, 5 2017.

[30] R. Stern, "Multi-agent path finding – an overview," 2019.

[31] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, 1968.

[32] S. Roni, S. Nathan, F. Ariel, K. Sven, M. Hang, W. Thayne, L. Jiaoyang, A. Dor, C. Liron, K. T. K. Satish, B. Eli, and B. Roman, "Multi-agent pathfinding: Definitions, variants, and benchmarks," *CoRR*, vol. abs/1906.08291, 2019.

[33] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," pp. 1443–1449, AAAI Press, 2013.

[34] D. Ratner and M. K. Warmuth, "Finding a shortest solution for the n × n extension of the 15-puzzle is intractable," 1986.

[35] P. Surynek, "An optimization variant of multi-robot path planning is intractable.," vol. 2, 7 2010.

[36] D. Kornhauser, G. Miller, and P. Spirakis, "Coordinating pebble motion on graphs, the diameter of permutation groups, and applications," pp. 241–250, IEEE, 1984.

[37] G. Röger and M. Helmert, "Non-optimal multi-agent pathfinding is solved (since 1984)," 2012.

[38] T. Standley, "Finding optimal solutions to cooperative pathfinding problems," vol. 24, pp. 173–178, 2010.

[39] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, "The increasing cost tree search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 195, pp. 470–495, 2 2013.

[40] A. Srinivasan, T. Ham, S. Malik, and R. Brayton, "Algorithms for discrete function manipulation," pp. 92–95, IEEE Comput. Soc. Press.

[41] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2 2015.

[42] E. Boyarski, A. Felner, R. Stern, G. Sharon, O. Betzalel, D. Tolpin, and S. E. Shimony, "Icbs: The improved conflict-based search algorithm for multi-agent pathfinding," 2015.

[43] K. Apt, *Principles of Constraint Programming*. Cambridge University Press, 8 2003.

[44] R. Bartak, N.-F. Zhou, R. Stern, E. Boyarski, and P. Surynek, "Modeling and solving the multi-agent pathfinding problem in picat," pp. 959–966, IEEE, 11 2017.

[45] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," 5 2017.

[46] M. Čáp, J. Vokřínek, and A. Kleiner, "Complete decentralized method for on-line multi-robot trajectory planning in valid infrastructures," 1 2015.

[47] T. T. Walker, N. R. Sturtevant, and A. Felner, "Extended increasing cost tree search for non-unit cost domains," pp. 534–540, International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[48] C. Ericson, *Real-Time Collision Detection*. CRC Press, Inc., 2004.

[49] M. Ryan, "Constraint-based multi-robot path planning," pp. 922–928, 2010.

[50] P. Surynek, A. Felner, R. Stern, and E. Boyarski, "Efficient sat approach to multi-agent path finding under the sum of costs objective," 7 2016.

[51] IBM Corportation, “Optimization programming language.”

[52] R. Barták and J. vancara, “On sat-based approaches for multi-agent path finding with the sum-of-costs objective,” 2019.

Acronyms

CBS Constraint Based Search. 1, 14, 15, 20, 35, 40, 41

CP Constraint Programming. 1, 15, 36, 41

CP constraint programming. 16, 35

CSP Constraint Satisfaction Programming. 24

CT Constraint Tree. 14, 15, 29, 36

CT constraint tree. 20

ICT Increasing Cost Tree. 14, 15

ICTS Increasing Cost Tree Search. 1, 14, 15, 17, 18

ID Independence Detection. 13

MA-CBS Meta-Agent Conflict Based Search. 20, 29

MAPD Multi-Agent Pickup and Delivery. 16, 17, 19

MAPD multi-agent pickup and delivery. 16

MAPF Multi-Agent Path Finding. 1, 3, 10–16, 18–20, 24, 25, 27, 29, 40, 41

MAPF multi-agent path finding. 10–13, 16, 17, 19, 41

MDD Multi-value Decision Diagram. 14, 15, 18, 29

MILP Mixed Integer Linear Programming. 1, 15

MIP Mixed Integer Programming. 16, 24

OD Operator Decomposition. 13

OPL Optimization Programming Language. 24, 26–28

PP Prioritized path Planning. 7

SAPF Single-Agent Path Finding. 8, 15, 20, 21

SAPF single-agent path finding. 8, 14, 15, 22, 23, 41

SAT Boolean Satisfiability. 16, 24

SIC sum of individual costs. 12, 28, 35

TDSP Time-Dependant Shortest Path. 22, 35, 36, 39