Department of Information Engineering and Computer Science

Master's Degree in
Artificial Intelligence Systems

Final Dissertation

Integrating Multi-Agent Path Finding Algorithms in ROS 2: The A* Case

Supervisor

Marco Roveri

Student

Giovanni Lorenzini

Academic year 2021/2022

Contents

Abstract

Industry 4.0 is revolutionizing many aspects of industrial and civil applications. In the field of intralogistics, *Autonomous Ground Vehicles* (AGV) are becoming increasingly present, alleviating work conditions and optimizing the flow of goods. AGVs need to move in the environment without collision, managing battery power, and accomplishing tasks. *Multi-Agent Path Finding* (MAPF) aims at computing non-conflicting paths for multiple robots. This problem is also called *Multi-Agent Pickup and Delivery* (MAPD) whenever agents need to move goods.

In this thesis, we have implemented an intralogistics system that runs in the Department of Information Engineering and Computer Science (DISI) based in Povo. Its task is to pick up and deliver packages between the first floor offices. First, a review of the different state-of-the-art methods to solve this task is presented. Then, a description of the architecture of the system is illustrated. We have developed the entire stack of the system which is composed of five main parts: Web GUI, Backend, Database, Planner, and Robot Operating System (ROS). The Web GUI is used by the end user to create new tasks and visualize the system status. It communicates with the Backend that manages all the components storing information in the Database, calling Planner instances to compute the paths, and instructing ROS on where to move the robots. In this group effort carried out by Lorenzini Giovanni and Planchenstainer Diego, all these components were developed. The core of the planner is based on the work of Enrico Saccon, in which we added one algorithm each, multi-agent A* and ICTS. Finally, the system has been tested with different problem instances to highlight strengths and weaknesses of the different algorithms. In this scenario, results proved that ICR (developed by Saccon), with ICTS as the sub-planner, is the best option among the tested ones, although a distributed approach could be a better way to solve the problem. The main obstacle that stops the system from being deployed permanently is the lack of charging points for the robots and battery management, which could be future work.

1 Introduction

The term intralogistics defines the logistic flows of goods and materials within a warehouse or an area inside a factory. It is used to distinguish processes that take place inside buildings, from the transport of goods in the outside world, defined as logistics. Intralogistics is a relevant problem for both civil and industrial applications since the movement of goods inside buildings takes place frequently.

Until a few years ago, this task was accomplished only by workers either diving forklifts or using pallet trucks. These are heavy tasks and have the risk to become alienating. Nowadays, *Autonomous Ground Vehicles* (AGVs) begin to be employed to alleviate laborer's work conditions. AGVs move inside the dynamic environment of warehouses, avoiding new obstacles, other robots, or even persons, to achieve their tasks. Multiple objectives can be accomplished in one task, such as moving between different locations, picking up and/or delivering goods, avoiding crashing, handling battery power, and returning to their charging base at the end of the mission. In this setting, AGVs should complete their task in minimum time, avoiding crashes, to maximize the efficiency of the intralogistics system.

This problem is known in the literature as the *Multi-Agent Path Finding* (MAPF) problem. In this thesis, it is inspected the problem of orchestrating a fleet of identical AGVs that deliver packages door to door between offices inside of the Department of Information Engineering and Computer Science (DISI) based in Povo. We proceeded to implement the entire stack of the system, from the specification of objectives to the generation of the agent's plan, motion planning, and real-time monitoring of mission progress.

Path Finding Algorithms

A floor map of the location in which the system has to be deployed should be provided. The map has to be discretized to be represented by a graph, upon which some robots should move, accomplishing some tasks, without colliding between each other or with other objects. To achieve this, a multi-agent path finding algorithm is necessary.

Firstly we implemented the single agent A* algorithm [9], to have alongside Dijkstra [78], already implemented by Saccon. A single-agent algorithm is necessary as many multi-agent algorithms rely on it.

At the heart of the thesis are the multi-agent pathfinding algorithms. We implemented the multi-agent A* algorithm with the improvements proposed by Standley [74] and the ICTS algorithm [23].

The implementation of the algorithms has been done using C++, by expanding the *Multi-Agent Open Framework* (MAOF) source code of Saccon [12].

Pickup and Delivery System

The other part of the thesis consists in integrating the constructed planner into a system usable by the end user to perform pickup and delivery tasks [29]. The user is presented with a graphical interface that shows a map on which it is possible to select the rooms to which a robot should go in order to pick up/deliver some goods. The GUI shows also the plans, the current position of the robots, the settings, and some statistics.

When a task is created it will be sent to the backend using the REST API [52] that we have developed. Then, as there are multiple robots, the one that will perform the task is selected with a simple algorithm, that manages the queue, in order to perform the greatest number of tasks in the shortest possible time.

It is then necessary to tell the robots how to move in space. This can actually be divided into two parts: a high-level one, where our multi-agent path planner is used, and a low-level one, provided by Nav2 [67], that manages the hardware. The high-level search is performed over a discretized map represented by a graph: the planner needs to find a plan for each robot, in a way that they do not collide and that they reach each goal of the tasks. It returns a list of waypoints that the low-level path planning of Nav2 will use, by connecting them into a physically feasible path.

The system can work either in a Gazebo simulated environment [56] or in the real world.

Experiments

The developed system has been tested in the simulated department environment. After this test, experiments were carried out to inspect planner performances. In this case, only the planners were tested, excluding all the other components. A total of 10 algorithms have been inspected: the four possible versions of A*, and ICTS with or without ID. These planners are our own contribution. The remaining are CP and ICR with A*, ICTS, or CP, which are developed by Saccon. Two typologies of experiments are evaluated: random tests, where goals in the map are chosen randomly, and handcrafted tests, where specific instances are run to inspect particular behaviors.

Teamwork

The system has been co-developed by Lorenzini Giovanni and Planchenstainer Diego. While it remains a team effort, here and along the thesis we will point out the parts that have been mainly developed by a single, which will also be differently written in the two theses, as each of us will focus on their part. In particular, Lorenzini is the author of the implementation of the A* algorithm (Sec. 6.1) while Planchenstainer is the author of the implementation of the ICTS algorithm (Sec. 6.2). Planchenstainer developed the Web GUI (Sec. 4.4) and the ROS (Sec. 4.5) parts and Lorenzini the Database (Sec. 4.2) and Backend (Sec. 4.3) parts. The Multi-Agent Open Framework (Sec. 4.1) has initially been developed by Enrico Saccon and then expanded by Lorenzini and Planchenstainer by implementing new planning algorithms.

2 Problem Definition

The aim of this thesis is to develop a system capable to pick up and deliver objects between the offices of Povo's first floor. Multiple robots can wander inside the building while performing their tasks, hence it is crucial that they do not collide with each other. Since it is not the scope of the work, no human interaction while moving is expected as it will need an ad hoc part of the system for dealing with that. To ensure that no collision happens, the paths of the robots need to be planned accordingly. This problem is called *Multi-Agent Path Finding* (MAPF) and is solved by ad hoc planners as the ones implemented during this thesis. The *solution* of the MAPF problem, defined as Π, will be a set of *k paths*, which are *Single-Agent Path Finding* (SAPF) problem's solutions, defined as π_i, where k is the number of agents and i is the agent's number. The result of a SAPF problem is a set of actions that if executed are able to move the agent from point A to point B. This set of actions can also be seen as a set of points through which the agents need to travel.

The problem of navigating between two near points is called Motion Planning and its burden is left to a specialized ROS node called Nav2. Motion planning is a low-level problem where a sequence of valid configurations that result in the motion of the agent from its start location to its destination has to be found. Moreover, the resulting path should be collision-free. A configuration is intended to define the pose of the agent. If the agent is represented as a point in 2D that can only translate, a configuration consists of two parameters (x, y). More complex robot definitions could consider also the shape of the robot which can now rotate and translate. The configuration will result then in 3 parameters: (x, y, θ). Motion planning subdivides the set of configurations in free space C_{free}, where agents avoid collisions, and its complementary obstacle space. It is then the task of the algorithm to find a set of configurations that can be feasible. As previously stated this problem will not be solved by our contribution, the aim of this paragraph is to inform the reader about this topic. More content can be found at [47, 41, 40].

Of course, robots need to know where to pick up a package and where to deliver it. This is done by the means of a web interface that can be employed by the user to request a service to the system. Clearly, the system needs some information in order to work. The fundamental one is the 2D map in which the robots will move. It is required that the map is a black-and-white image of the floor map, where the black parts represent walls or not accessible areas and the white parts represent free space. Note that doors should not be drawn for the correct functioning of the system. Another crucial information that should be provided is the conversion rate from image pixels to meters. So if a map long $l = 20\text{m}$ is represented by an image wide $w = 500\text{px}$, the conversion rate will be:

$$\text{conversion rate} = \frac{l}{w} = 0.04 \left[\frac{\text{m}}{\text{px}}\right]$$

After other minor pieces of information are provided, the system is set up and the user can now select the office from which one robot needs to pick up something and one or more offices where it needs to travel by and deliver the package(s). The system will then assign a robot to the task and solve the new MAPF problem. Then all the occupied robots will move to their destination.

2.1 MAPF - Multi-Agent Path Finding problem

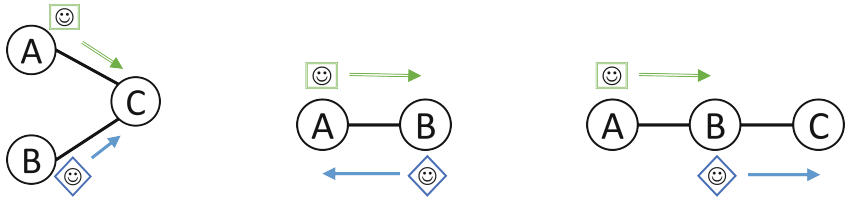
Multiple definitions of the MAPF problem are present in the literature, hence an introduction of the classical MAPF problem definition is presented. Then, a more precise variant of our problem will be depicted.

As a reference for the classical MAPF problem we report the one described by Stern in [61], where the problem with k agents is defined by a tuple (G, s, t) where:

- $G = (V, E)$ represents an undirected graph where its vertexes $n \in V$ are the possible locations that an agent can occupy and every edge $(n, n') \in E$ represents the connection between two nodes.
- s is a function that maps an agent to its initial location.
- t is a function that maps an agent to its desired destination location.

In this formulation, time is discretized into time steps. Each agent can perform only a single *action* per time step and during a time step all the agents must perform an action. The possible actions are either *move* to the neighbor vertexes or *wait* in place for one time step.

Now a formal definition of *single-agent plan, joint plan* and *valid solution* of the MAPF problem is given. A *single-agent plan* π_i for an agent a_i is defined as a sequence of actions $(\alpha_1, \ldots, \alpha_n)$ that lead the agent at location $t(a_i)$ if it starts at location $s(a_i)$. A *joint plan*, called also *solution*, Π is a set of k single-agent paths π_i, one for each agent a_i. A solution for the MAPF problem is considered *valid* if it does not present any of the following conflicts.



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

Figure 2.1: Possible types of conflict.
Source: Stern [61].

In Fig. 2.1 are presented the three main conflicts that can arise in the MAPF settings. The first is the *vertex conflict*, where two agents plan to occupy the same vertex at the same time. The second conflict is called *swap conflict* and happens when two agents plan to swap their locations over the same edge at the same time step. The last conflict is the *following conflict*, where an agent plans to occupy at time $t + 1$ the position of another agent at time t. This conflict is relevant when the length of the edges or the time used for their traversal is not unitary, which can result in two agents colliding. An extended *following conflict* where the agents enter into a circular pattern is called trivially *cycle conflict*. Since in our task we are considering unitary edges and the robots move at the same speed, we omit the last two kinds of conflicts.

The solution to MAPF problems is usually not unique. Often, a solution that optimizes some objective function is highly appreciated. The most common ones are *MaKeSpan* (*MKS*) or *Sum of Individual Costs* (*SIC*). The first is equivalent to the maximum number of time steps needed for one agent to reach the goal among all the agents.

$$MKS(\Pi) = \max_{1 \leq i \leq k} |\pi_i|$$

Instead, the SIC is the sum of actions performed until all agents reach their goal.

$$SIC(\Pi) = \sum_{1 \leq i \leq k} |\pi_i|$$

Usually, agents reach their target at different time steps. It turns out to be relevant how the agents are handled when they reach the goal while other agents are still in movement. Two common assumptions [62] are listed below:

- **Stay at target:** in this case, once one agent reaches its target position it remains there, causing a vertex conflict for agents that want to travel through that node.
- **Disappear at target:** here after an agent reaches its target position, it immediately disappear. This implies that other agents can pass through that node in the future. Obviously, this condition cannot work in real world scenarios.

The majority of the works in the literature assume a stay-at-target behavior, even though some works consider the other behavior, as in [28].

A possible way to solve MAPF is by using A*. However, as it will be clear later, it is unfeasible to solve for an increasing number of agents [61]. The difficulty of solving the problem can be estimated by considering the *size* of the state space, which is equal in this case to the number of vertexes of the graph G, and its *branching factor* which is the average outgoing degree. For this problem, in the worst case scenario, both the parameters

are exponential in the number of agents, $|V|^k$ for the size and $\left(\frac{|E|}{|V|}\right)^k$ for the branching factor. A MAPF problem instance on a 4-connected 500×500 grid with 20 agents results in a search space size of $500000^{20} \approx 9.1 \times 10^{107}$ and a branching factor of $4^{20} \approx 1.1 \times 10^{12}$. "The exponential branching factor is especially problematic for A*, since A* must at least expand all vertices along an optimal path" [61]. It is easy to see that no solution could be found in human times. A* is infeasible, but more advanced techniques can solve quite efficiently the problem. A deep insight will be presented in Chapter 3.

In [39] authors modeled a very similar problem called multi-robot path planning on graphs with parallel moves and they prove that it is NP-complete when trying to minimize either makespan or minimum total distance.

2.2 Pickup and delivery task

Our problem is very similar to the one detailed by Ma *et al.* [30]. There, the authors define a *lifelong* variant of the MAPF problem called *Multi-Agent Pickup and Delivery* (MAPD). MAPD models a realistic scenario where agents constantly receive new tasks that have to be accomplished, this is why is called *lifelong*. Since users can select a new task at any time, our problem fits very well in this definition. However, a difference is present. In this case, the agents could be required to travel to multiple destinations within a single task, whether in MAPD there is only one pickup and one delivery location. Also, we do not assume strict behavior for agents when they reach the goal. They can both stay at the goal if no other agent must travel there or move to another vertex if other agents must travel there.

Formally the problem is very similar to the MAPF instance. An instance of the MAPD problem with k agents is defined by:

- $G = (V, E)$ represents an undirected graph where its vertexes $n \in V$ are the possible location that an agent can occupy and every edge $(n, n') \in E$ represents the connection between two nodes.
- s is a function that maps an agent to its initial location.
- d is a function that maps an agent to its desired destination locations.

In this formulation, time is discretized into time steps. Each agent can perform only a single *action* per time step and during a time step all the agents must perform an action. Here, it is assumed that the length of the edges is the same for every edge. Furthermore, we assume that the actions performed by the robots are synchronous, i.e. they start performing an action (either move or wait) at the beginning of a time step and they end the action at the end of that timestep.

A task set $\mathcal{T}$ contains the set of unexecuted tasks. At every moment a task τ_i can be added to $\mathcal{T}$. Each task $\tau_i \in \mathcal{T}$ consists in a pickup location $p_i \in V$ and one or more delivery locations $\mathbf{d_i} = [d_{i1}, \ldots, d_{iz}]$, where $d_{ij} \in V$ and z is the number of delivery locations to be reached. An agent is called free if it is not currently performing any task, otherwise is called occupied. A free agent can be assigned to any task $\tau_i \in \mathcal{T}$. When a task is assigned is removed from $\mathcal{T}$. A task can also be assigned to an occupied agent that will queue the task and execute it after finishing the pending one. The motivation behind the multiple delivery locations can be seen in the following example. Suppose that a form needs to be signed by different professors and then delivered to the administrative office, this task applies perfectly to this problem modeling.

2.3 Povo offices floor

The location in which the robots will move is the first floor of the university building at Povo. It consists of two separate blocks connected by a bridge. A floor map is reported in Fig. 2.2. The building is 210 meters long and 47 meters wide. The majority of the halls are wider than 1.5 meters and doors are no wider than 1.2 meters.

To make the map "readable" for the robots, a decomposition is performed to transform the image into a graph of points. One could decide to manually decompose the map creating a graph with only relevant locations, but as it is a tedious job, an automatic Python script is used to create the graph as explained in Sec. 4.3.3. Two parameters are defined to control the script: the width of the doors and the discretization distance. The first is very straightforward and was set to 1.2 meters. The second is a parameter which regulates the distance between the points of the graph grid that is overlapped over the map. We found 1.3 m to be an effective discretization distance. This results in a graph with $\sim$ 1200 nodes. A higher value would consequence in halls that will not

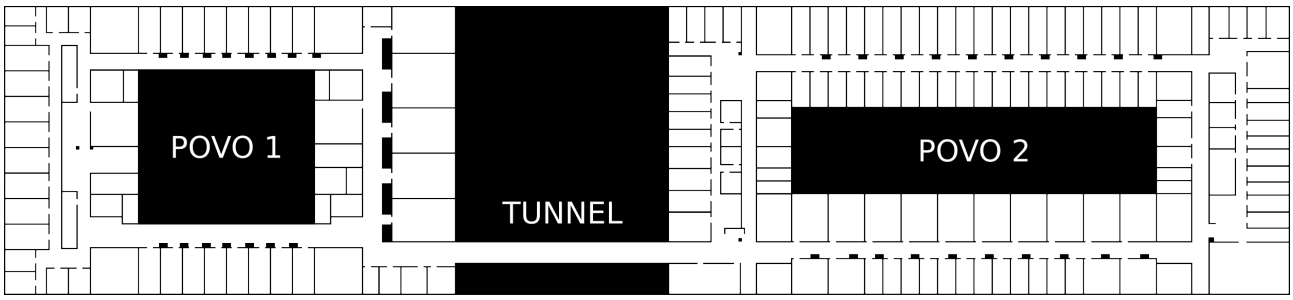


Figure 2.2: Floor map of Povo's DISI department building.

be discretized, a lower value will increase quadratically the number of nodes. In this setting the majority of the hallways are at least two nodes wide, allowing agents to travel easily in two opposite directions. Only three corridors will be composed of a single line. In those locations, agents will struggle to find a path, but since they leverage the planner no conflict will arise. These corridors are the horizontal upper corridors of Povo 1 and Povo 2 and the leftmost vertical corridor of Povo 2. There are also some dead ends located in the corners of the two buildings.

3 Background

The core of the thesis is to plan and actuate multi-robot movements inside a building. To do so, two fundamental components have to be present: the multi-agent planner which plans the path and a system that orchestrate robots movement to execute the plan. With these premises, an analysis of the state of the art for these two arguments has been carried out.

3.1 MAPF Algorithms

Many solutions for solving the MAPF problem have been proposed over the years. The majority of the algorithms are either optimal, but computationally demanding, or fast, but suboptimal. Examples of the first category are [35, 73], where the solution is optimal but it does not scale up to many agents. For the second category, prioritized planning [45] decomposes the problem into several sub-problems giving back results faster, sacrificing optimality.

One of the sub-optimal methods that apply A* to this problem is Local Repair A* [5], a family of algorithms used in the game industry, which search a path for every agent ignoring the others except for its current neighbors. The agents start following their routes until a collision is imminent. Here, a full A* search is performed, resulting in extensive CPU usage, bottlenecks, and cycles.

Cooperative A* [6] and its variants proposed by Silver in 2006 aim at solving LRA*'s performance issues. The task is divided into a series of single-agent searches. Here the search is performed along space and time and takes into account other agents' moves by inserting them into a reservation table. Since the order in which the moves are inserted into the reservation table can affect the solution, it is important to note that certain problems can not be solved if the inserting order is wrong. This implies that the greedy nature of this algorithm results in sub-optimal solutions.

These and the following combinatorial approaches can be subdivided into two categories: centralized and distributed. Centralized algorithms leverage a central processing unit where all the computations are performed and then send the planned paths to each associated agent. This category of algorithms was very popular in the old days, but distributed algorithms are starting to emerge in these last years. Distributed algorithms deliver the burden of planning to the single intelligent agent and limit or delete the existence of a centralized entity that manages the agents.

Other than combinatorial algorithms, another category of MAPF solvers is moving its first steps: Reinforced Learning based algorithm, often enhanced by Deep Learning.

Now, more interesting algorithms will be introduced in order of publication.

3.1.1 A*

Based on the single-agent A* search algorithm adapted to deal with multiple agents, in 2010 Standley [74] proposed two improvements. His algorithm is the first practical, admissible, and complete algorithm to solve the MAPF problem. He introduced the *Operator Decomposition* (OD) to reduce the branching factor (applicable to many search algorithms) and exploited the intrinsic independence usually present in MAPF problems through the *Independence Detection* (ID) technique. These two improvements are described below.

OD - Operator Decomposition

Motivated by the fact that in the standard A* every operator moves at the same time, thus expanding nodes with a branching factor of 5^k, Standley proposes to assign a move to one agent at a time, in a fixed order. This technique is called *Operator Decomposition* (OD) and reduces the branching factor from 5^k to 5 since only one agent is moved per step, but multiplies the depth of the goal node by k. Once a solution is found only the states where every agent has performed an action are returned.

ID - Independence Detection

Here the intuition of the previously developed algorithm is expanded by independently planning optimal paths for disjoint and exhaustive subsets of agents. If these paths were all found to not interfere with one another, then the total solution obtained would be optimal. This algorithm starts considering all paths independently and recomputes the conflicting ones together with an optimal multi-agent algorithm. This reduces the algorithm's complexity from exponential in the number of agents to exponential in the size of the largest group.

3.1.2 ICTS - Increasing Cost Tree Search

Increasing Cost Tree Search [23] (ICTS) is a combinatorial search strategy applied to the MAPF problem. It divides the problem into two sub-problems. The first is finding the minimum cost solution for each individual agent between all the possible combinations in the search space, this phase is called high-level search. The second is finding if a valid solution exists under the cost constraints imposed by the high-level search. This phase is called low-level search and can be seen as the goal testing of the solution given by the first phase.

Next, the two parts are inspected in detail.

High-level search

The high-level search is performed on the *Increasing Cost Tree* (ICT). An example of ICT can be seen in Fig. 3.1, here dashed lines denote duplicate children that can be pruned. Each node of the ICT is composed of a k-vector of individual path costs $[C_1, C_2, \ldots, C_k]$, where k is the number of agents. The node represents all the possible complete solutions in which agent a_i's path length is equal to C_i. The sum of all the costs in the node is equal for every node at the same level in the tree. The tree starts from a root where the cost of the optimal individual path for each agent assumes that other agents do not exist. Each node will generate k successors. Successor $succ_i$ increments the costs of agent a_i resulting in a cost vector of $succ_i = [C_1, C_2, \ldots, C_{i-1}, C_i + 1, C_{i+1}, \ldots, C_k]$, thus increasing the total cost by one. A goal node is found when it exists a non-conflicting complete solution such that the cost of each individual path of agent a_i is equal at C_i.

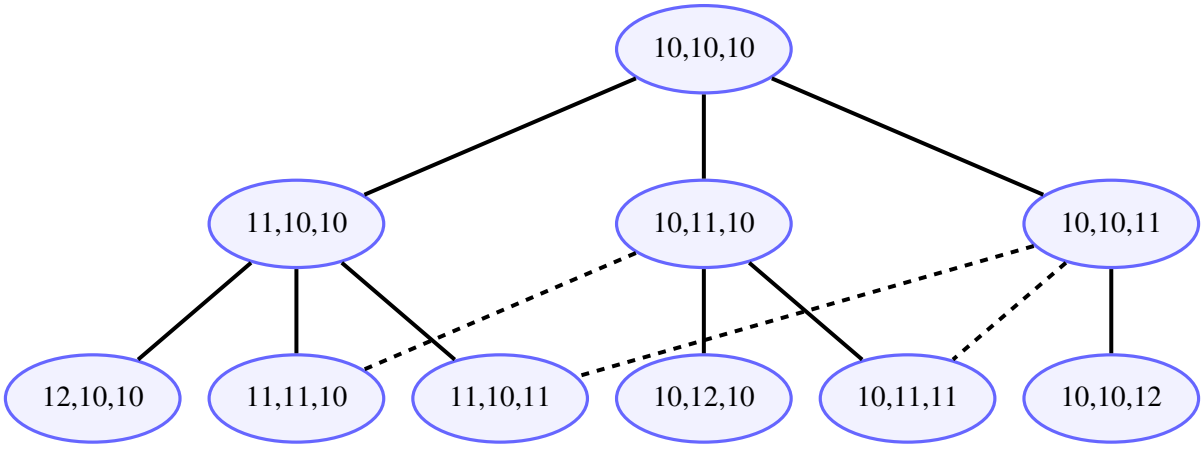


Figure 3.1: ICT node for three agents, dashed lines are duplicate nodes that can be pruned. Single agent's paths are long 10 steps.

It's easy to see that a *Breadth-First Search* (BFS) on the ICT will find the optimal solution as the cost increases by one at each successive level. For each node explored in the ICT, the low-level search determines if it is a goal node.

Low-level search

To find a goal node a non-conflicting complete solution should exist. All the possible combinations between the possible paths of every agent should be checked to ensure its existence. This operation is computationally demanding so, for every agent, a *Multi-value Decision Diagram* (MDD) is used to store all the possible paths. MDDs are *Directed Acyclic Graphs* (DAGs) that generalize *Binary Decision Diagrams* (BDDs) by allowing more than two choices for every decision node. The MDD of the cross product between all the agents' paths is then explored to reveal if a set of non-conflicting paths exists. If so, the algorithm returns the path and ends.

Other variants

Extended Increasing Cost Tree Search for Non-Unit Cost Domains [75] is a proposal for extending ICTS to non-unit cost domains, so composed of graphs with edges of different lengths. This is achieved by two variants, ϵ-ICTS and w-ICTS, that are sub-optimal.

3.1.3 CBS - Conflict Based Search

Since the state space of the MAPF problem is exponential in k number of agents, but for the SAPF problem is linear in the graph size, CBS [22] aims at decomposing the MAPF problem into a large number of SAPF problems. The SAPF ones are easy to solve but there can be an exponential number of them. Some definitions are needed for better clarity in the following. The tuple (a_i, v, t) describes a *constraint* for a given agent a_i that can not occupy a vertex v at time t. A *consistent path* for agent a_i is a path that satisfies all its constraints. Likewise, a *consistent solution* is a solution that is made up of consistent paths. A *conflict* is defined by a tuple (a_i, a_j, v, t) where agents a_i, a_j occupy the same vertex v at time t. A solution is *valid* if all of its paths do not conflict with each other. Note that a consistent solution may be *invalid* if its paths have conflicts.

CBS incrementally adds a set of constraints for each agent and finds paths consistent with these constraints. If the solution is invalid, add a new constraint and search for new paths. This algorithm is based on two search levels that will be discussed below.

High-level search

In the high-level search, CBS searches a binary tree called *Constraint Tree* (CT). Each node contains a set of constraints, a solution, and the total cost of the current solution, which is the sum of all the single agent path costs. The root of the CT does not have constraints. The child of a node inherits its constraint, but does not have to store them all, it can traverse the tree until the root to obtain them if necessary. Also, it adds a new constraint related to only one agent, the other constraint will be associated with the other child. The paths that compose the solution are found by the low-level search. The search on the CT is performed with a best-first search with nodes ordered by their costs.

When the path is returned by the low-level search, it is validated with the others and if no conflict is present this CT node is declared as the goal node. If a conflict (a_i, a_j, v, t) arises while confronting the paths, the validation process stops and the node is declared non-goal. In this case, two children are generated for the given CT node, one has (a_i, v, t) as a new constraint, and the other has (a_j, v, t). Since the only change in the new node is caused by the newly added constraint, one can call the low-level search only on the related single agent.

Low-level search

The low-level search is given an agent a_i and its set of constraints. It performs a search ignoring the other agents finding an optimal path that satisfies all the constraints. Any SAPF algorithm can be used while verifying that the constraints are satisfied.

Other variants

Other variants of CBS exist like ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding [8] that provides the improvements Merge and Restart for enhancing MA-CBS and Prioritize Conflicts for enhancing the bypass improvement.

3.1.4 CP - Constraint Programming

Constraint Programming (CP) is a methodology for representing and solving combinatorial search problems through constraint propagation and intelligent search. Usually in CP two steps are required: problem modeling and solving. Problems should be encoded as a finite set of *variables* and *constraints* that model relations between variables that have to be *satisfied*. If constraints are well-formed, CP proves to be optimal and complete. The objective of this methodology is to leverage the new powerful SoA optimal solvers by converting the MAPF problem either in CSP [51], SAT [57], Inductive Logic Programming [53], and others, as done in these works.

Saccon's approach [12] for CP is based on [58], but for some implementation constraints, he opted for IBM's CPLEX solver [34] as it was easier to implement and easier to maintain.

3.1.5 TP - Token Passing

Ma *et al.* [30] argue that the majority of the MAPF planners ignore important characteristics of real-world scenarios, such as automated warehouses, where agents receive constantly new tasks. In their paper, they study a *lifelong* version of MAPF problem which they call *Multi-Agent Pickup and Delivery* (MAPD). Two decoupled algorithms are presented: *Token Passing* (TP) and *Token Passing with Task Swaps* (TPTS), which proved to be effective in solving well-formed MAPD instances. A decoupled algorithm is defined where each agent assigns itself a new task and computes its own collision-free path independently given some global information, as for example other agents' planned paths. TP takes its inspiration from Cooperative A* [6] in which agents plan their path one after the other. Token passing has been used before to develop COBRA [80], which is a MAPF algorithm that does not consider certain real-life problems that cause its execution to fail sometimes.

In TP it is assumed that when an agent reaches its last location, it rests there until a new task is issued. Agent a_i uses A* search to find the path to its pickup and delivery destination. The search moves into a 3-dimensional state space formed by (x, y, t). When a state causes a collision, it is removed from the state space. The algorithm of TP is summarized below.

The system initializes the token with paths where all agents are resting at their position. At each time step all new tasks are inserted in $\mathcal{T}$. So $\mathcal{T}$ is a task set that contains the set of unexecuted tasks. Any free agent can request the token once per time step and the system provide it to them. Token possessors choose a task from the set, where the task's last location is not another agent's pickup or delivery location. If such a task or more exist, the agent assigns itself to the one with the lowest h-value between its current location and the pickup location. Then it removes the task from $\mathcal{T}$ and updates its path with one that causes no collisions with other paths in the token. If no such task exists, it either rests in position or moves away to avoid deadlocks. Finally, return the token to the system.

TPTS is a refinement of TP where a free agent can "steal" the task to another agent that is moving to the pickup location, if they are closer.

The two algorithms are proved to be able to solve all well-formed MAPD instances. TP and TPTS are compared with a centralized algorithm. It turns out that TP is the perfect choice when real-time computation with a very high number of agents is required and it can be easily extended to a fully distributed algorithm. The only cost that has to be paid is a decrease in the performances which results in paths that are $\sim 45\%$ longer with respect to TPTS and the centralized algorithm.

3.1.6 PRIMAL - Pathfinding via Reinforcement and Imitation Multi-Agent Learning

An interesting approach to find a solution for the MAPF problem is represented by the use of *Reinforcement Leaning* (RL) to learn a policy able to solve MAPF instances. A series of works [21, 20, 19] proposed by the MAPF group at the University of Southern California culminates in PRIMAL 2 [44] the only framework (to our knowledge) able to solve the MAPF problem with this method.

As the name suggests, PRIMAL 2 [44] is the enhancement of PRIMAL [19]. Since the scope of this chapter is to introduce the reader to the possible ways to solve the problem in a general way, only PRIMAL will be presented as it is more relevant. PRIMAL is a framework for MAPF that combines reinforcement and *Imitation Learning* (IL) to teach fully-decentralized policies. Here agents will reactively plan paths online while expressing coordinated behavior in a partially observable world. The framework combines RL and IL from an expert MAPF planner. Agents will simultaneously learn to follow efficient single-agent paths through RL and to imitate the expert to favor movements that will benefit the whole team, thus avoiding selfish behaviors. Since the policy learned is equal for all the single agents, it can be copied onto an arbitrary number of them. A partially observable world is considered, where robots can only observe a 10×10 region centered on themselves. A fixed *Field of View* (FoV) helps to reduce the input dimension and allows the policy to generalize to arbitrary world sizes. Since the agent needs to have information about its goal position, the geometric vector between itself and its goal is provided. For the correct functioning of RL, a *reward* function should be defined: agents are penalized for every time step in which they are not on the goal (-0.3 in case of movement, -0.5 if they remain still). Stopping at the goal has no penalty. Collisions between agents can be discouraged by adding a penalty equal to -2. Once the episode ends (all agents on their goal) the reward function returns $+20$. The network structure is presented in Fig. 3.2. It takes in input the vector between agent and goal (2D tensor, distance/direction to the goal) and four channel matrices of the FoV, each one representing one of the following: obstacles, other agents' position, goal position (if present in FoV) and other agents' goals (if present in FoV). The two inputs are processed along two different paths. The FoV instances pass through a 6-layer convolutional network

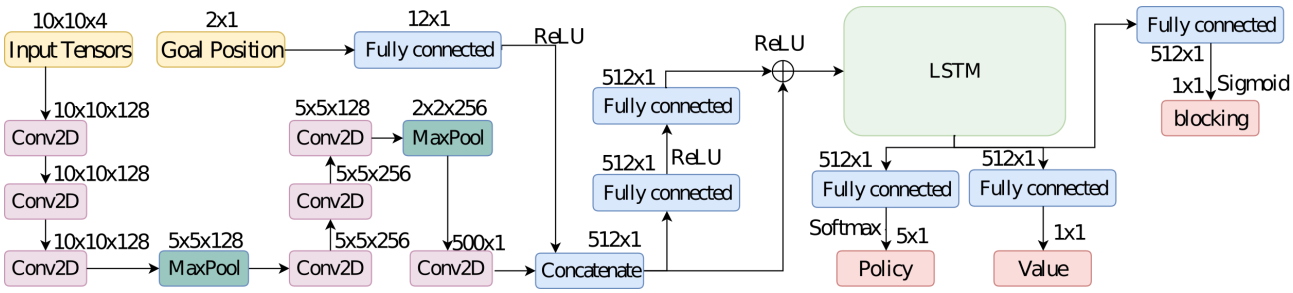


Figure 3.2: PRIMAL network structure consists of 7 convolutional layers interleaved with max pooling layers, followed by an LSTM.
Source: PRIMAL [19].

inspired by VGGNet [42] with 3×3 kernels between each max-pooling layer. Instead, the goal information is processed by a fully connected layer, which output is concatenated to the output of the "simil-VGGNet". The concatenation of the preprocessed input is fed to 2 fully connected layers with a residual connection that runs in parallel and connects to their outputs. The features are then processed by an LSTM cell with an output size of 512. Three output branches with fully connected layers fork from the LSTM's output. One activated by a softmax generates the policy, another one activated by a sigmoid outputs the *blocking* prediction, and the last one, without activation function returns the value in output. Four different losses are used to train the network.

Three different training tricks are used to enhance the network performances:

- **Blocking penalty:** a penalty of -2 is added to the reward function that applies when an agent that stays on its goal decides to stand still and block an agent that has to pass through to reach its goal. The *blocking* output of the network is trained to provide an explanation to the agent of what causes this penalty.
- **RL/IL combination:** during training the learning paradigm is selected randomly at the beginning of the episode. In the case of IL, the solution is computed online by a suboptimal, since $\epsilon = 2$, centralized planner (ODrM*) [24] and the behavior cloning loss is minimized.
- **Environment Sampling:** smaller and dense environments, where coordination is needed, are forcefully generated since with random distributions this is not usually the case and agents do not learn cooperative behaviors.

The results show that in low obstacle densities PRIMAL excels, being capable of managing up to 1024 agents in 160×160 map size, whether other combinatorial planners fail because of the fast growth of the state space. Also, the solutions found by PRIMAL are usually as much as twice times longer than optimal ones. But as soon as the density of obstacles increases and the map dimensions shrink the performances of the Deep Reinforcement Learning algorithm drop. Instead, in those settings, centralized planners work quite well overtaking the results of PRIMAL. One of the main drawbacks is the training time: it took 20 days of training at the Pittsburgh Supercomputing Center (PSC) on 7 cores of an Intel Xeon E5-2695 and one NVIDIA K80 GPU.

3.1.7 ICR - Increasing Conflict Resolution

ICR [10] is an algorithm developed by Enrico Saccon based on *Large Neighborhood Search* (LNS) [72]. It aims to improve the speed of other MAPF algorithms, such as A*, ICTS, or CP. The idea is to compute the single-agent paths and then check for conflicts. For each conflict found, a group between the conflicting agents is created along with a sub-graph, used as a map for the problem. The problem is thus solved using another MAPF algorithm.

The sub-graph is created by expanding two graphs from the conflicting agent's positions until an intersection is found. Every agent present in the sub-graph is involved in the conflict resolutions even if originally it was not part of the conflict. If a solution is not found inside the sub-graph, it will be enlarged by a hop until a solution is found or a limit is reached. It is possible that the sub-graph includes all the locations of the map, in that case, the problem is resolved by only the MAPF algorithm.

When the paths for solving the conflicts are found, they are merged back with the general solution. This is repeated until all the conflicts are solved.

3.2 ROS - Robot Operating System

Robot Operating System (ROS) is an open source software development kit for robotics applications, but it is not an operating system. Two versions are currently available ROS 1 [50] and ROS 2 [68]. The first version was mostly used in academics and the objective was to create a quality and performing system setting aside security, network topology, and system up-time, therefore not being ready for industrial applications. As more commercial and industrial applications started to use ROS 1, a major refactor was necessary to ensure that properties that were ignored in the beginning received the attention needed. So ROS 2 was completely redesigned to address these challenges. This was allowed by leveraging the *Data Distribution Service* (DDS), an open standard for communications, that is used in critical infrastructure such as military, spacecraft, and financial systems [18]. DDS proved to be game-changing as it enabled ROS 2 to obtain best-in-class security, embedded and real-time support, multi-robot communication, and operations in non-ideal networking environments.

ROS 2 was designed following four main principles:

1. **Distribution:** Usually robotics problems are solved with distributed approaches. Different components are needed to satisfy different requirements, like perception systems, control systems, hardware device drivers etcetera. Each component runs in different contexts and shares data via explicit communication. This structure has to be handled in a secure and decentralized way.

2. **Abstraction:** The semantics of the data exchanged must be defined through interface specification. This abstraction allows for a compromise between the need to expose the details of a component and that of overfitting the application to that component.

3. **Asynchrony:** An event-based system is created leveraging on asynchronous communication of messages.

4. **Modularity:** This principle recalls the UNIX design goal that aims at making each program do one thing well [70]. Modularity is enforced at multiple levels of the ecosystem.

The authors state that adhering to these principles "facilitates code reuse, software testing, fault isolation, collaboration within interdisciplinary project teams, and cooperation at a global scale" [68].

ROS 2 has also to fulfill some other requirements in order to be compliant with industries' needs. First, it should be **secure** to accidental or malicious misuse. Consequently, authentication, encryption, and access control are integrated. The widespread of micro-controllers either in sensors, actuators, or other peripherals, lead to the need for particular ROS instances (*Micro-ROS*) [31] that can run on **embedded systems**. Furthermore, since the **possible networking environments are countless**, ROS 2 should be capable of configuring the flow of the data adapting to the constraint of the network itself. This is done by the means of *quality of service*. Another important aspect is **real-time computing**, a fundamental requirement for many applications such as self-driving cars or humanoids. ROS 2 offers APIs for developers of real-time systems to enforce application-specific constraints.

3.2.1 Architecture

ROS 2 architecture is based on multiple abstraction layers distributed across several decoupled packages. This implementation allows users to swap components or take only the pieces of the system they need, which could be important for certifications. The abstraction layers are reported in Fig. 3.3 and are generally hidden, ROS developers usually do not have to work with them. In fact, the majority will work only with the client library. The latter provides access to the core communication APIs. ROS 2 provides three APIs: one based on C++ (*rclcpp*), one on Python (*rclpy*), and the last one based on Java (*rcljava*). Each one is written ad hoc to leverage the particular capabilities and features of the related programming language. Client libraries depend on *rcl*, an intermediate interface written in C that provides a baseline for each client library. Below *rcl*, the middleware abstraction layer called *rmw* (ROS MiddleWare) provides the essential communication interfaces. Each middleware vendor has to implement the *rmw* interface, which has to be interchangeable with the other without code manipulation. This results in the possibility for the user to select the preferred middleware based on their requirements. Network interfaces, such as topics, services, and actions (that will be presented in Sec. 3.2.3), are defined with *Message Types* using an *Interface Description Language* (IDL). Two possible types definition are provided: ROS IDL format (*.msg* files) or OMG IDL standard (*.idl* files). Interface definitions provided by the user are generated at compile time and create the code required for communication in any client library language.

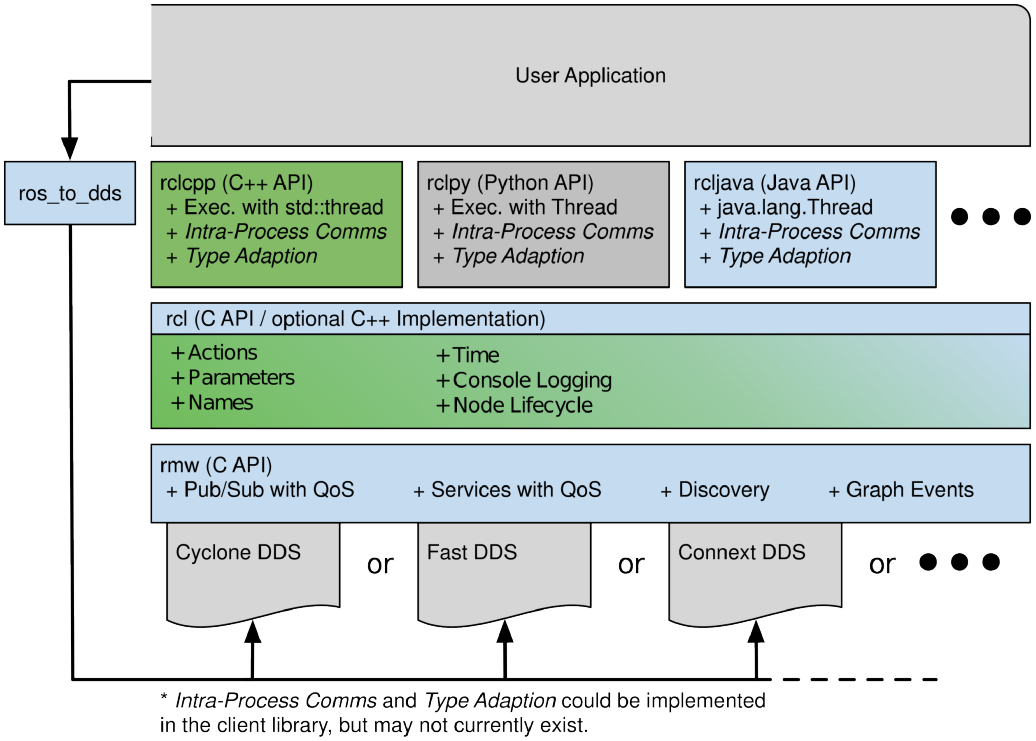


Figure 3.3: API stack of ROS 2 Client Library.
Source: ROS 2 [68].

3.2.2 Nodes

The system is divided into more easy-to-comprehend parts called *nodes*. A node is an organizational unit that can talk with other nodes exchanging information by the means of network interfaces. ROS 2 presents an architectural pattern for managing the *lifecycle* of the nodes. This pattern is represented as a state machine with 4 different states: Unconfigured, Inactive, Active, and Finalized. The possibility to access the state of a particular node is important for the coordination of the different components of a distributed asynchronous system. Nodes can also be written as components that can be associated with any process, leaving the possibility to change the location of the node based on a variety of circumstances.

3.2.3 Topics, Services, and Actions

Finally, network interfaces (or communication patterns) are presented herein. The main patterns are topics, services, and actions, but other minor instances are present as parameters, timers, launch, and other auxiliary tools. An example of the exchange of information can be seen in Fig. 3.4. Here 3 nodes dialogue by using the three main network interfaces.

Topics

Topics are an asynchronous message-passing framework. ROS provides an anonymous publish-subscribe functionality that permits many-to-many communications. To observe any message passing on that topic it is sufficient to create a node that subscribes to that topic, without any further change.

Services

When asynchronous communication is not the right option, services can be used. They are a request-response style pattern that provides data association between a request-response pair. This can be useful when we want to ensure whether a task is completed or received. Services clients' processes can not be blocked during a call.

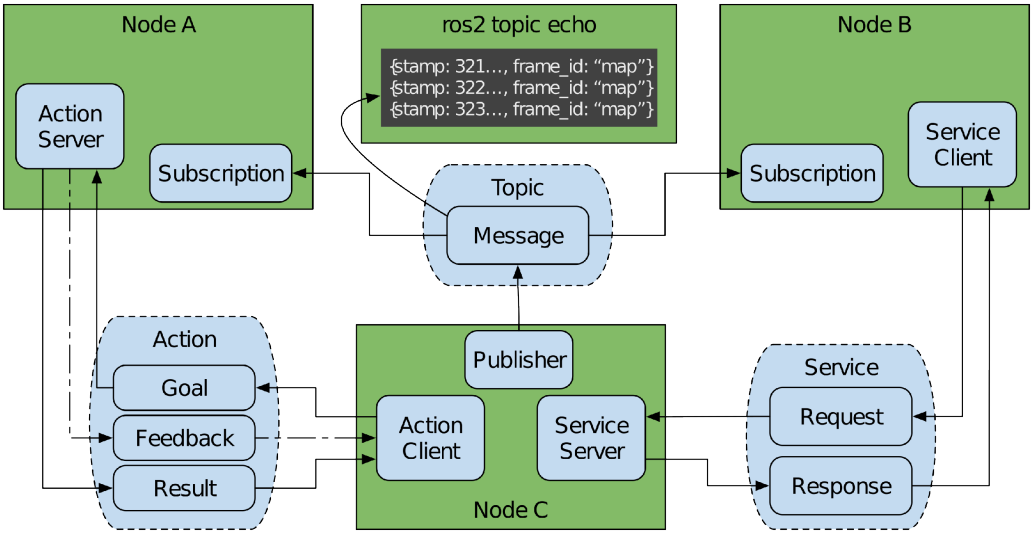


Figure 3.4: ROS 2 node interfaces: topics, services, and actions.
Source: ROS 2 [68].

Actions

The last communication interface is the action. Actions are goal-oriented and asynchronous communication interfaces with a request, response, periodic feedback, and the ability to be canceled. This type of pattern is often used in long-running tasks such as autonomous navigation. In fact, the majority of Nav2 communication interfaces are actions. This interface is built on top of topics and services. They function similarly to services, but as previously said, can be canceled during execution. Actions use a client-server model: the action client sends a goal to the action server that read the information and provides a feedback topic and a result. Actions are composed of three "hidden" services and two "hidden" topics listed below:

- Topics:
  - `/action/name/_action/status`
  - `/action/name/_action/feedback`
- Services:
  - `/action/name/_action/send_goal`
  - `/action/name/_action/cancel_goal`
  - `/action/name/_action/get_result`

Here, `/action/name` stands for the name of the action, in our case would be `FollowWaypoints`. A deeper introspection on the actions used in this project will be presented in Sec. 4.5.3.

3.2.4 Nav2

Navigation 2 [67], shorten to Nav2, is an open source ROS 2 package that is able to handle navigation for robots. This package plunge its root in ROS Navigation, which was one of the most popular solutions adopted in the field. "Navigation2 uses a behavior tree for navigator task orchestration and employs new methods designed for dynamic environments applicable to a wider variety of modern sensors" [67]. *Behaviour tree* (BT) proved to be the new SoA for task planning in robotics, surpassing *Finite State Machines* (FSM) [27]. These system components are designed to be highly interchangeable, in fact, every node of the behavior tree invokes a remote server to compute its task. In this way, many algorithms can be used to solve the problem. Furthermore, a plugin interface for the server is implemented to allow the creation and selection of new algorithms at run-time. Hence it's clear that the framework is built with modularity and configurability in mind. Not only these principles are taken into account, but also safety, security, and determinism are pursued to create an industrial-grade system.

Reliability

Reliability is achieved by leveraging the DDS middleware of ROS 2. Moreover, since ROS 2 introduced the concept of lifecycle nodes, Nav2 servers use these kinds of nodes to allow deterministic program lifecycle management and memory allocations.

Modularity and Reconfigurability

Since many types of robots can work in countless types of environments Nav2 is meant to be highly modular. Authors chose solutions that are not only modular but easily configurable and effortlessly selected at run-time.

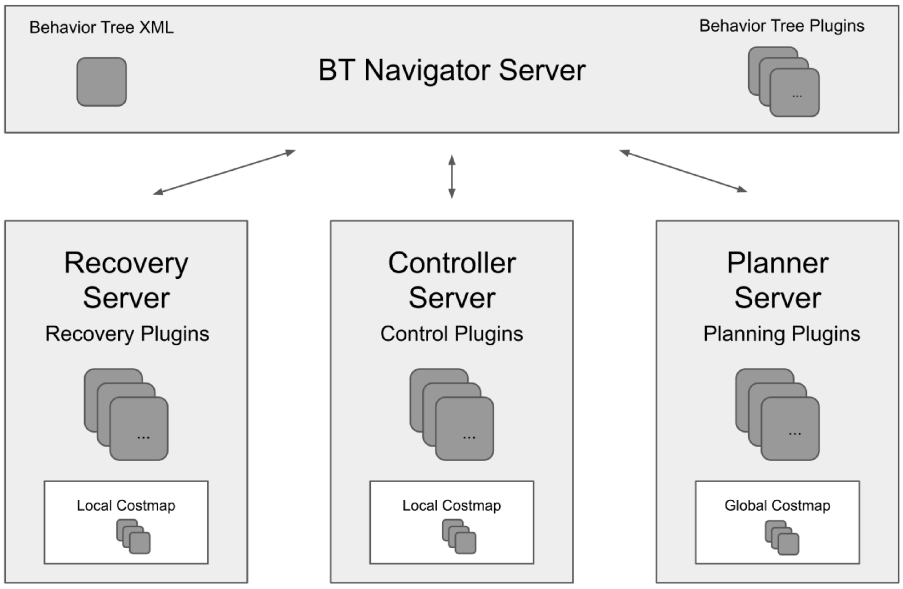


Figure 3.5: Nav2 navigation stack. On top it can be seen the BT navigation server that orchestrates the task-specific servers.
Source: Nav2 [67].

To do so, two design patterns have been created: *behavior tree navigator* and *task-specific asynchronous servers*. An overview of the system is presented in Fig. 3.5.

A BT is used to organize the navigation task by enabling and supervising the progress of the different task-specific servers, namely: planner, controller, and recovery. The calls to the different servers are performed through action interfaces, as navigation is a long-running task. Ad hoc navigation behaviors can be modeled by modifying the XML of the BT. BT is built upon *BehaviorTree.CPP* library because of its popularity. This implies that Nav2 can be used as a subtree in more complex BTs.

Each task-specific server hosts a ROS 2 server, environmental model, and run-time selected algorithm plugin. Modularity implies that none or multiple task-specific servers can run at the same time to compute actions. The entry point for BT navigator nodes is the ROS 2 server which is also capable to handle cancellation, preemption, or new information requests. The algorithm plugin processes new requests to complete the task.

Behavior tree navigator

The highest-level component of Nav2 is the behavior tree navigator, which hosts the tree that implements navigation behaviors. In the BT navigator server (Fig. 3.5) the user-defined BTs are loaded and run. Each node calls a server, below, to complete a task. Fig. 3.6 shows the standard Nav2 BT, called also "Navigate To Pose With Replanning and Recovery", which has been used in the experiments of the authors. Symbols in the tree represent two basic components of the behavior tree: '?' represent a fallback node, '→' represent a sequence node. More information on behavior trees can be found on their site [64] or in this paper [27].

Starting from the bottom left and following the logical flow towards the right, a policy ticks the global planner action at a rate of 1 Hz. If the planner fails, the fallback node proceeds to activate the child that clears the global environment of prior obstacles. This operation can solve failures in the perception system. The parent

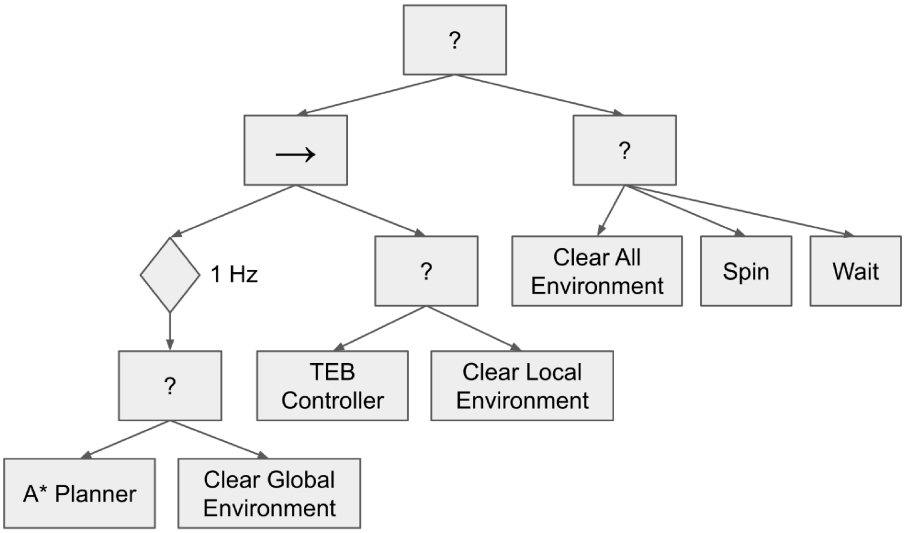


Figure 3.6: Behavior tree used in the marathon done by Nav2 authors.
Source: Nav2 [67].

sequence node ticks then the controller or the clearing behavior in the same way as for the previous branch. If both the controller and the planner fail, the root node calls the right child that implements a more aggressive recovery behavior. It performs in sequence three actions, from the mildest to the most aggressive. It first tries to clear all the environment. If this fails the robot spins in place to reorient local obstacles. As last resort, the robot waits for dynamic obstacles to give way.

3.2.5 Transform library

The tf2 library is used to keep track of the coordinates using coordinates frames related to each other [17]. By defining the relations between the various components of the robots and the world we can get the coordinates of any object relative to any frame. This is a big advantage that simplifies dealing with coordinates. Another feature of tf2 is knowing the positions in an arbitrary past time.

3.2.6 Robot perception

ROS can take in input information provided by many sensors like LiDAR, GPS, camera, IMU, and encoders. For a robot used in an MAPD task is necessary to know its position relative to the map. This is achieved typically by fusing the information coming from a LiDAR to scan the environment, wheels encoders to know how many rotations they have performed, and an IMU that gives accelerations and velocities.

3.2.7 TurtleBot3

The robots used in this thesis are TurtleBot3 (Fig. 3.7). TurtleBot is the most popular ROS standard platform robot and it is designed to teach people who are new to ROS through TurtleBot. The choice of this platform is motivated by the fact that it is a small, affordable, programmable, ROS-based mobile robot that does not cost thousands and includes the majority of the sensors needed in robotics.

The robot is pretty compact as it is 13.8 cm long, 17.8 cm wide, and 19.2cm tall. TurtleBot is a mobile differential robot powered by two motors, with encoders, that control the wheels. It mounts a Raspberry Pi 3 that handles the ROS 2 stack and an OpenCR (Open-source Control module for ROS) board to control the hardware and to read IMU data. A 360° LiDAR is present on top of the structure and allows the robot to perform SLAM and Navigation with ease.

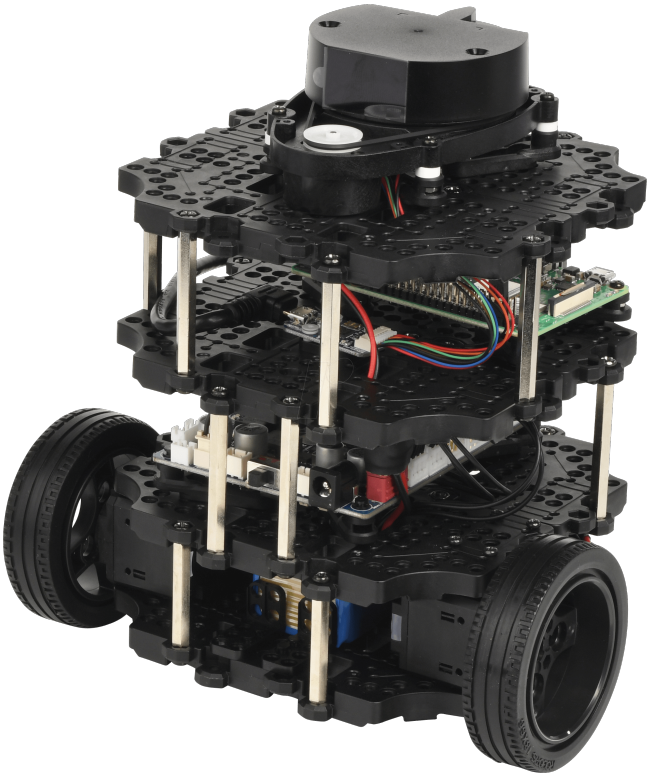


Figure 3.7: Turtlebot3. From top to bottom are present: LiDAR, Raspberry Pi 3, OpenCR board, motors with encoders and wheels, and LiPo battery.
Source: reichelt elektronik [63].

4 Software Architecture

Our system is composed of different parts, that do different operations and need to "talk" to each other. Every part is programmed with the most suitable programming language for the given application, so we end up using C++, Python, TypeScript, and SQL. To make them communicate we used different solutions like HTTPS, Web Sockets, JSON files, and ROS 2 interfaces. In Fig. 4.1 the final structure of the software can be seen.

The source code of the system can be found in a GitHub repository (github.com/lorenzinigiovanni/logistics-robots) while the source code of the MAOF planner can be found in a git repository hosted on Bitbucket (bitbucket.org/lorenzinigiovanni/maof). To implement the system, 156 files were written which result in ~ 12000 lines of code and 84 classes.

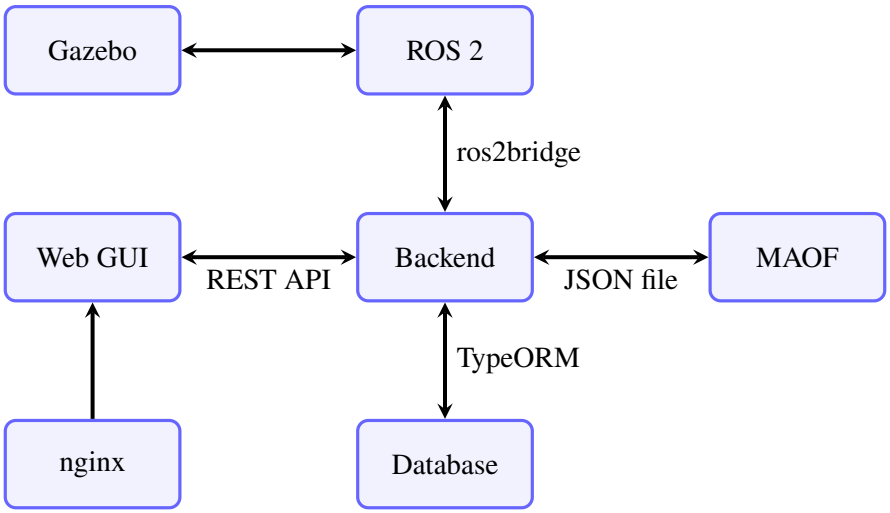


Figure 4.1: Architecture of the software.

The following is a brief list of the system modules and their modes of communication, which will be discussed in more detail later:

- *MAOF* (Sec. 4.1) is the planner, written in C++. Our contributions amount to ~ 3200 lines of code and 8 classes spanning over 21 files. It computes a plan for each robot given a list of goals and a graph. Different planning algorithms can be used such as A* (Sec. 6.1) and ICTS (Sec. 6.2). The input containing the graph, the list of goals, the positions of the robots, and the required algorithms is given as a JSON file. The output plans are also written in a JSON file. The *MAOF* is invoked by the *Backend* as needed.
- *Database* (Sec. 4.2) stores all the information that needs to be persistent. It is based on PostgreSQL and is accessed by the *Backend* through an ORM.
- *Backend* (Sec. 4.3) is the core of the system and manages all the system. It is written in TypeScript and runs on Node.js. It consists of ~ 2500 lines of code and 25 classes written in 33 files. A script for elaborating a map image is written in Python 3 and takes in input an image and gives in output a JSON file with the needed information. Here, 1 class was defined and the number of lines is equal to ~ 500.
- *Web GUI* (Sec. 4.4) is the graphical interface that the users see from their browser. It is written in TypeScript for the Angular 2 framework. The written code is made up of ~ 3600 lines of code resulting in 43 classes and 83 files. It is hosted by an nginx server and communicates to the *Backend* over HTTPS using the REST API that we have defined.
- *ROS* (Sec. 4.5) is the Robot Operating System that runs in the robots and manages the low-level navigation by the means of Navigation 2. It is also used along with Gazebo to simulate the map and the robots.

It communicates the positions and the plans of the robots with the *Backend* using WebSockets. The ROS nodes are written both in C++ and Python. Here our contributions are mainly for startup and configurations files, with ~ 1900 lines of code in 11 files.

4.1 MAOF - Multi-Agent Open Framework

MAOF is a state-of-the-art framework aimed at testing various multi-agent path finding algorithms. Its development is mainly done by Enrico Saccon that created the initial framework (bitbucket.org/chaff800/maof) and implemented some SAPF and MAPF algorithms. Our contribution is mainly composed by single-agent A* (Sec. 5.2) [9], multi-agent A* with Standley's improvements (Sec. 6.1) [74] and ICTS (Sec. 6.2) [23].

The framework has been designed in such a way that while maintaining a formal structure, the code that composes the various building blocks can be replaced or expanded (Fig. 4.2). The code can be compiled with a set of SAPF and MAPF solvers and at runtime, the suitable pair be chosen. It is achieved by having virtual `SAPFSolver` and `MAPFSolver` classes that will be the parents of all the implemented algorithms. Both kinds of solver needs a `solve()` method that is used to compute a plan for one or more agents.

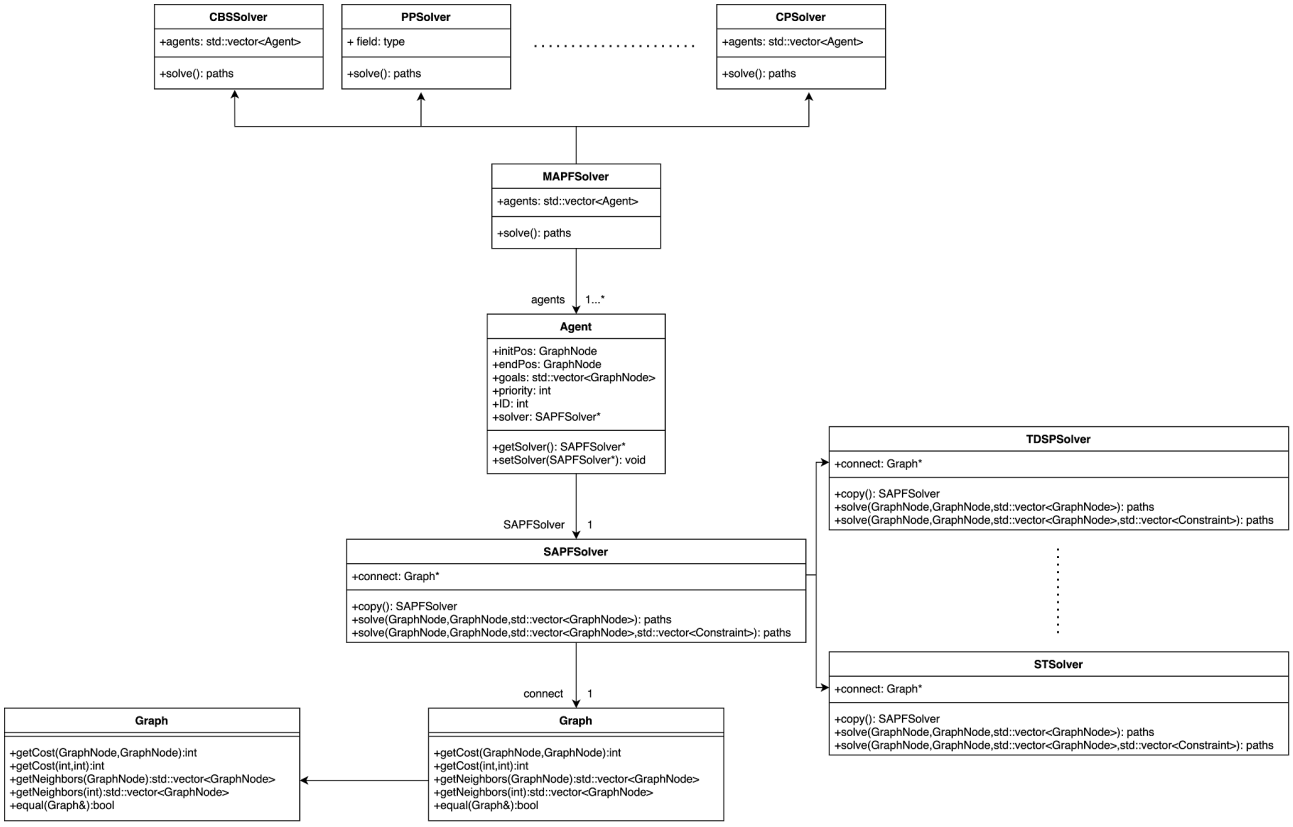


Figure 4.2: MAOF structure.
Source: Enrico Saccon [11]

The main classes that compose the MAOF are:

- `Graph` is an abstract class that stores information about the structure of the map. Only the method to obtain a cost of a link between two nodes and the method to get the neighbors of a node are exposed. It can be implemented both as a linked list or as a connectivity matrix. In this case the second approach is the chosen one.
- `Agent` represents a robot that needs to compute a task in the `Graph`. Every `Agent` has a `SAPFSolver`, a list of goals, and an ID.
- `SAPFSolver` is an abstract class that needs to be inherited from to implement a SAPF solver like A*. It finds a path for the single agent possibly accounting for constraints on the occupation of vertexes or edges at a given time.

- `MAPFSolver` is an abstract class that needs to be inherited from to implement a MAPF solver like A* or ICTS. It exposes the `solve()` method that is in charge of finding a global plan without conflicts that works for all the agents. Usually, the `MAPFSolver` uses, to some extent, the single agents `SAPFSolvers` in the construction of the global plan.

The framework takes in input a JSON file that has all the necessary information needed to compute a plan for the robots. The file is structured as follows:

- `nodes` is a list of nodes that compose the graph. Each node is represented by a vector that contains the `ID`, `value`, `x`, and `y` coordinates.
- `connect` is a connectivity matrix that stores how the `nodes` are connected between each other.
- `agents` is an array of robots where each one is formed by: `ID`, `initPos` that is the ID of the graph node that represents the initial position of the robot, `endPos` that is the ID of the graph node that represents the final position of the robot, `goalPos` that is a list of graph nodes that the agent should visit, `priority` used by the priority planner and `name` for a friendly name of the agent.
- the settings indicate to MAOF how to solve the problem specifying the `MAPF` solver, the `SAPF` solver, the `costFunction`, and the `heuristic`.

The framework outputs a JSON file, that contains a plan, for each robot, consisting of a list of graph nodes' IDs.

4.2 Database

A database is used to store information about robots, maps, users, etc. The typology of database that we have chosen is the relational one [13] and in particular a SQL-compatible one. The data is organized in tables with some defined columns that represent the structure of the data. Particular attention is given to storing only once a given information, by splitting the data into normalized tables that are linked by relationships as needed.

Each table represents a distinct type of object and should have at least one primary key column that is unique to each row. The primary key is used for linking together the data from other tables. As a primary key, we choose to utilize a UUID [54] for every table as it gives some advantages with respect to an incremental integer: it is unique across tables and DBs. Also, it does not reveal the incremental position of the entity, which can be a potential security problem if used inside an URL.

As a *Relational Database Management System* (RDBMS) we choose to use PostgreSQL, which is open source, powerful, and compatible with the *Structured Query Language* (SQL) [26]. The database is not managed directly but through an ORM, as explained in Sec. 4.3.1.

4.2.1 Architecture

The first step was to determine which data we have to store. We end up with this list: robots, tasks (what the user input), plans (what the robots should do), settings, users, and the map. For each entity, a set of columns needs to be identified. For example, a user can be represented by name, email, and password.

As the second step, we needed to determine the relationships between the various tables, for example, that a robot has a plan that in turn needs to know of which graph nodes it is composed. The relationships can be one of these types: *one-to-one*, where a record in a table relates to one and only one record in another table; *one-to-many*, in which a record in a table relates to zero, one or many records in another table; *many-to-many*, that is constructed as two *one-to-many* relationships and a join table to store them [38].

The third step is to normalize the DB (or check if it is already normalized). It consists in reducing the tables to normal form which is necessary to provide data integrity and reduce redundant data. An ideal table should: represents a single subject, not contain multi-valued fields, not contain calculated fields, and not contain duplicate fields [38]. The normalization that can be achieved is one of the incremental normal forms available (1NF [13], 2NF, 3NF [14], BCNF [15], 4NF [59], 5NF, 6NF). Our database should at least satisfy the third normal form (3NF).

In Fig. 4.3 the final structure of the database can be seen.

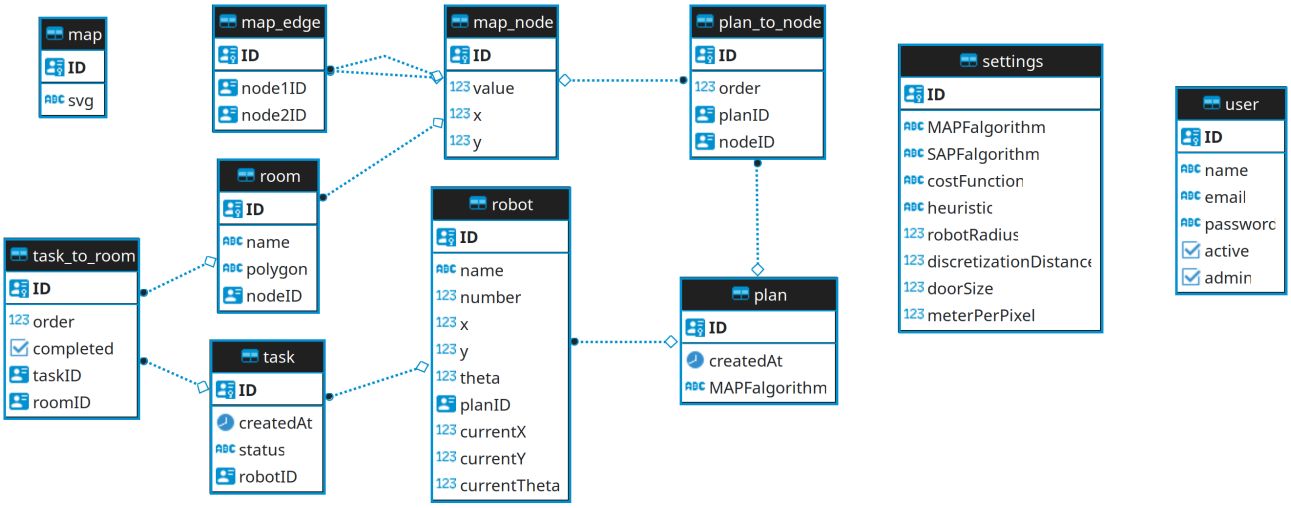


Figure 4.3: Structure of the database.

User

The API and therefore the Web GUI, are protected by logins. To achieve this, a table with the `users`' information is needed. In particular, the columns are: `name` with the name of the user, `email` (unique) that is used during the login and to reset the password, `password` contains the password hashed by Argon2 [1], `active`, a flag for activating the user, and `admin`, a flag to set the user as admin.

Map

The map is represented by a graph (a set of nodes connected by edges) and some other information.

To store the graph two tables are needed: one to store the nodes (`map_node`) and one to store the edges (`map_edge`). The columns for the former are the `value` that is an incremental number required in MAOF and the position coordinates `x` and `y`. The latter has two columns to store the link between two nodes, in order to represent a *many-to-many* relationship between the same entity.

To store the other information the table `map` is needed. Here the only column is the `svg` one, which stores a representation of the map used by the Web GUI. The last needed table is `room` which stores the name and the coordinates of the rooms/offices that are selected by the user to form a `task`.

Robot

Each robot is represented by the followings: `name`, `number` used by MAOF (unique), the initial coordinates of the simulation, and the current coordinates. The relations are a *one-to-one* with `plan` (a sequence of nodes that the robot should visit) and a *one-to-many* with `task` (the tasks required by the user).

Plan

A plan is an ordered list of `map_node`s with the possibility to have repetitions. This is therefore a *many-to-many* relationship as a node is linked to multiple plans and each plan is composed of multiple nodes. The particularity of the join table is that there is a `order` column used to maintain the sequence ordered.

Task

A task is an ordered list of `rooms` with the possibility to have repetitions. This is therefore a *many-to-many* relationship as a room is linked to multiple tasks and each task is composed of multiple rooms. The particularities of the join table are that there is a `order` column used to maintain the sequence ordered and a `completed` column to know which rooms the robots have visited. The `task`s table stores the `createdAt` time in order to serve the users in a FIFO manner and a `status` to know if the task is assigned to a robot, is in execution, or is completed.

Settings

This is a table with multiple columns and a single row used to store some settings used by the system, like the algorithms to use, the resolution of the map, and other properties.

4.3 Backend

The Backend is the core of our system. It manages communications with the MAOF planner, ROS 2, the database, and the Web GUI. This piece of software is written in TypeScript and runs under Node.js. TypeScript is a superset of JavaScript that introduces typing [76]. Node.js is a JavaScript environment, based on the V8 Engine, that works outside of a web browser. We used it to execute the Backend server-side. Node.js includes the *Node Package Manager* (NPM) that connects to a rich online database of user-provided packages that helps in developing a program [55].

4.3.1 ORM

The DB was not directly created and managed by hand, but TypeORM is used instead. TypeORM maps an object of a certain class in TypeScript to a row in a certain table in the database [77]. To write a class using TypeORM is necessary to indicate, by using the decorator `@Column`, which property should be mapped to a DB column. It is also possible to indicate the required type for a certain column and if it could be `nullable` or unique. There are appropriate decorators to indicate relationships, for example, `@OneToMany` is used to declare a *one-to-many* relation in which we need to specify the type of the other object and what property of that holds the reference to this object.

During the use of a class mapped to DB, it is necessary to call the save method when the object should also be saved in DB. To load one or more objects from the DB, it is possible to use some simplified methods like `find` or to use the more powerful `QueryBuilder`. In the end, the request will always be converted into a SQL query but with the advantages of being simpler to write and that the fields are automatically sanitized.

The models that we have created are reported in Sec. 4.2.

4.3.2 REST API

To interface the Backend with the Web GUI, a set of REST APIs was defined. REST APIs are used to provide an interface between physically separated components in a client-server architecture. Our APIs are based on the HTTP methods GET (to retrieve one or more resources), POST (to create a new resource), PUT (to update a resource), and DELETE (to delete a resource) [52]. For transmitting generic objects we have chosen to use the JSON format. When a request arrives at the Backend through the API it will be processed and if necessary, a read or write operation to the DB will be performed.

While most of the endpoints simply retrieve, update or create resources, we will report hereinafter the most interesting ones.

User authentication

Access to the Web GUI and the API is restricted to logged-in users. Firstly, the user should register and then an admin should enable their account. The login process works as follows:

- The user transmits its email and password to POST `api/auth/login`. The password is in "plaintext" but is protected by HTTPS which encrypts it using *Transport Layer Security* (TLS).
- The password is verified against the hash saved in the database, using the state-of-the-art Argon2 hashing function [1].
- If the credentials are valid, then a *JSON Web Token* (JWT) is issued. The JWT can store information like the name or the role of the user. It cannot be modified by a third party as it is signed with a secret only known by the Backend [4].
- If the credentials are not valid then an HTTP 401 error code is given in response.

The user should attach the issued token to every request and the Backend will check its validity and if the user is authorized to access the given resource. At every request, the token is replaced with a new one because it has an expiration date.

Map

The input map is represented by an image with white pixels for free space and black pixels for obstacles or walls. The map, therefore, needs to be elaborated to be encoded in a graph as explained in Sec. 4.3.3. The POST `api/map` endpoint accepts in input an image and through a Python script converts it into a graph, which is then stored in the database.

In the Web GUI, we also need to visualize the map along with its rooms and the plans of the robots. By performing a GET operation on `api/map/svg` an SVG file is returned, that contains the vectorized image of the map, the rooms that are represented by polygons with an ID used to identify them, and the lines that compose the plans of the robots.

Tasks and Plans

A task is a list of rooms that is submitted by the user that wants to have a robot that performs a pickup and delivery task for them. A task is created by sending a list of rooms to POST `api/tasks`. What happens next is that a robot is chosen to perform the task as explained in Sec. 4.3.4. Then the MAOF planner is invoked to compute the plans for every assigned robot, which are then stored in the database.

4.3.3 Map elaboration

The map elaboration is performed in a separate Python 3 script as it is more suitable for using OpenCV.

What we have as input of the algorithm is an image where the black pixels represent the walls and the white pixels represent the free space. The input image, in our case, has been constructed as follows:

1. Create a 2D map of the floor, in meters, using LibreCAD. Then save it in the DXF format.
2. Import the DXF file in Inkscape with a manual scale of 1. Now it is possible to fill the walls with black and leave the free space as white.
3. Export to PNG with resolution 1000 DPI $\simeq$ 0.0254 m/px. This parameter is important as it has to be inserted into the Web GUI settings page.

What we need in output is a list of rooms with their coordinates and a graph. The graph is built over the discretized free space, ignoring the space occupied by rooms. However, a node for each room is added to the graph in order to allow the robots to access them.

The algorithm is reported in Alg. 1 and will be explained hereinafter. The first step consists in inverting the white and black pixels using the `bitwise_not` operation. It allows us to simplify the following computations. Two types of morphological operators [46] are applied to the input image to obtain two new images:

- `roomsImg` is obtained by applying the closure operation. The rooms in the images do not have a door, the closure is applied with a size proportional to the size of a door and has the effect of filling the gaps in the contour. Therefore, the result is an image with the rooms composed of 4 filled walls, that can be easily detected as rectangles.
- `wallsImg` is obtained by applying the dilation operation. Here, the walls are dilated by the radius of the robots, ensuring that a robot will never be requested to go too near to the walls, avoiding a possible crash.

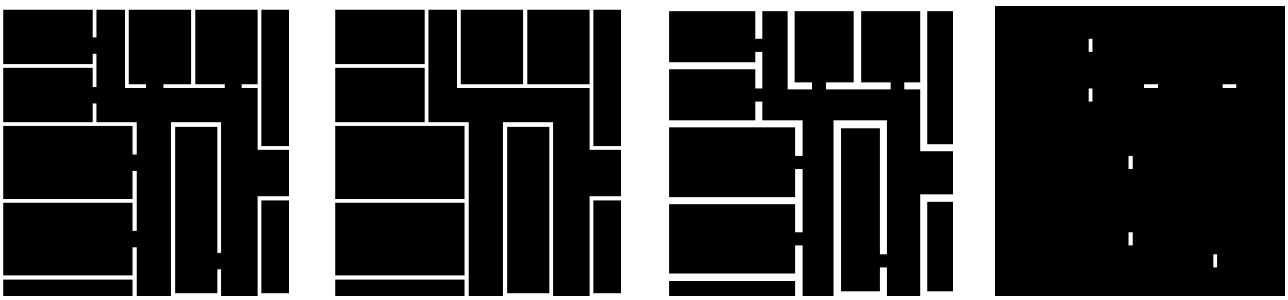
The position of the doors has to be identified for future operations. To do so, the subtraction between `roomsImg` and `wallsImg` is performed resulting in `doorsImg`. The newly created figure will contain only a rectangle of white pixels in correspondence to every door. In Fig. 4.4 the original image which has been inverted (a) and the 3 derived images (b, c, and d) are shown.

The contours from `roomsImg` are extracted. If a contour is composed of 4 edges then it represents a room and will be inserted in the `rooms` array, otherwise, it is a corridor and it will be discarded. The contours of the rooms will be used in this script to add a graph node inside them and in the Web GUI as a click area used to compose a task.

A grid of points with distance `discretizationRes` is applied over all the image and then refined by removing the points inside rooms or that touch a wall. To construct a `graph` the remaining points, which are in the free space, are used as nodes. Edges are added to connect neighbor points when a line between them does not intersect an obstacle.

Algorithm 1: Map elaboration

Input: `map`: the binary image representing the map (0: free space, 1: walls)
Input: `resolution`: the resolution of the image [m/px]
Input: `discretizationRes`: the discretization resolution [m]
Input: `doorsSize`: the width of the doors [m]
Input: `robotRadius`: the radius of the robot [m]
;
Data: `rooms`: a list of rooms
Data: `doors`: a list of doors
Data: `graph`: graph composed of nodes and edges
;
begin
    `roomsImg` ← closure operation of `map` by `doorsSize/resolution`;
    `wallsImg` ← dilation operation of `map` by `robotRadius/resolution`;
    `doorsImg` ← `roomsImg` - `wallsImg`;
    ;
    `contours` ← find rooms contours in `roomsImg`;
    foreach *`contour` in `contours`* **do**
        if *size(`contour`) is 4* **then**
            Append `contour` to `rooms`;
    ;
    Put a `grid` of graph nodes all over the image with spacing `discretizationRes`;
    foreach *`node` in `grid`* **do**
        if *`image[node]` is white or `image[node]` is inside `room`* **then**
            Delete `node`;
    foreach *`node` in `grid`* **do**
        foreach *`neighbor` of `node`* **do**
            if *`line(node,neighbor)` does not intersect a white pixel of `wallsImg`* **then**
                Connect `node` to `neighbor`;
    ;
    `doors` ← find doors contours in `doorsImg`;
    foreach *`door` in `doors`* **do**
        `newNode` ← graph node near the `door` inside the `room`;
        `nearestNode` ← graph node near `newNode`;
        Connect `newNode` to `nearestNode`;
        Add `newNode` to `graph`;
    ;
    Generate an `output.json` file that contains `graph` and `rooms`;

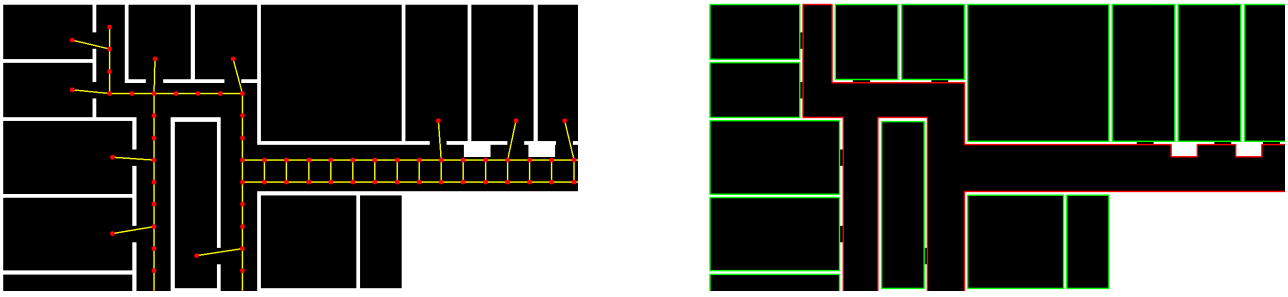


(a) inverted input (b) rooms (c) walls (d) doors

Figure 4.4: Morphological operators applied to input map.

A single graph node near the door inside the rooms is required to permit the robots to access them. To achieve this, for each door a new graph node near it is created and then connected with an already existing graph node.

In the end, the information about the `graph` and the `rooms` are encoded in a JSON file that will be used by the Backend. In Fig. 4.5 the final result with the graph and the rooms is shown.



(a) graph with nodes (red) and edges (yellow) (b) rooms (green) and corridors (red)

Figure 4.5: Final result.

4.3.4 New task management

When a user inserts a new task it needs to be assigned to a certain robot. The robot that will execute the task is chosen in order to start the task as soon as possible. This is achieved by looking at the current position and the currently assigned tasks of each robot, to pick the one that minimizes a heuristic of the assigned tasks plus a heuristic to reach the start of the new task. The algorithm that selects the robot for a new task is detailed in Alg. 2.

4.4 Web GUI

The logistic system is developed to be used by end users that do not necessarily have advanced computer capabilities, therefore it is necessary to provide a simple interface to interact with the system. To achieve this, a graphical user interface intended to be used from a web browser has been developed. The Web GUI provides access to the map to insert new tasks, manage robots and general settings.

The web app is programmed using Angular 2 which is a framework that provides the tools necessary to develop single-page applications. The used programming language is TypeScript, a superset of JavaScript that supports typing [76]. An Angular application is mainly composed of *components* that are a TypeScript class combined with an HTML template. A component, therefore, defines how it is rendered and how it behaves. Every page can be composed of multiple components and other HTML elements. The application is single-page and the right content is shown by the router depending on the URL. The other parts of the application are the services used to make API requests to the Backend. The services are automatically injected where required by a component [25].

The compiled web application needs to be served by a web server such as nginx. We choose to use nginx as it is open source, light, and fast [7]. We also used it as a reverse proxy, that takes a request and forwards it to the

Algorithm 2: New task management

```
Input: newTask: new task that needs to be assigned
Input: robots: a list of robots
Input: robots[i].tasks: a list of tasks assigned to the robot i
;
Data: minHeuristic: the minimum value of the heuristic
Data: bestRobot: the robot that will be assigned the task
;
begin
    minHeuristic ← +∞;
    foreach robot in robots do
        heuristic ← 0;
        foreach task in robot.tasks do
            foreach goal in task do
                heuristic ← heuristic + H(previousGoal,goal);
                previousGoal ← goal;
        heuristic ← heuristic + H(previousGoal,newTask[0]);
        if heuristic < minHeuristic then
            minHeuristic ← heuristic;
            bestRobot ← robot;
```

right service. It, therefore, enables the hosting of multiple services or applications with a single public IP [69].

4.4.1 Pages

Users can navigate through different pages by selecting the wanted page in the menu on the left. Users and administrators will see different available pages. Users can only access *Dashboard*, *Map*, and *Tasks* pages. Instead admins have complete access to all pages, namely *Dashboard*, *Map*, *Tasks*, *Robots*, *Settings* and *Users*. Every user has access to their *User profile*.

Login and Registration

When a user navigates to the URL of the Web GUI a login page is presented that requires a valid account composed of an email address and a password. If the user is not already registered, he can go to the registration page where a name, an email address, and a password are required. Before logging in with the new account an administrator needs to enable the account.

Dashboard

The dashboard (Fig. 4.6) is the home page where some usage statistics are shown. The statistics are organized into 5 widgets and are related to robots, tasks, plans, and planners. The widgets are the following:

- The first one is composed of three numbers representing the number of robots, the number of planned tasks, and the total number of goals defined in the tasks.
- The second one is a horizontal bar chart where the Y axis represents the robots and the X axis the number of tasks. The tasks are colored differently depending on their status.
- The third one is a vertical bar chart that shows how the number of tasks changes in a given period of time. The periods can be a day (with an hourly resolution), month (with a daily resolution), or year (with a monthly resolution).
- The fourth is a vertical bar chart where the X axis is the length of the plans and the Y axis is the number of occurrences of a plan length.

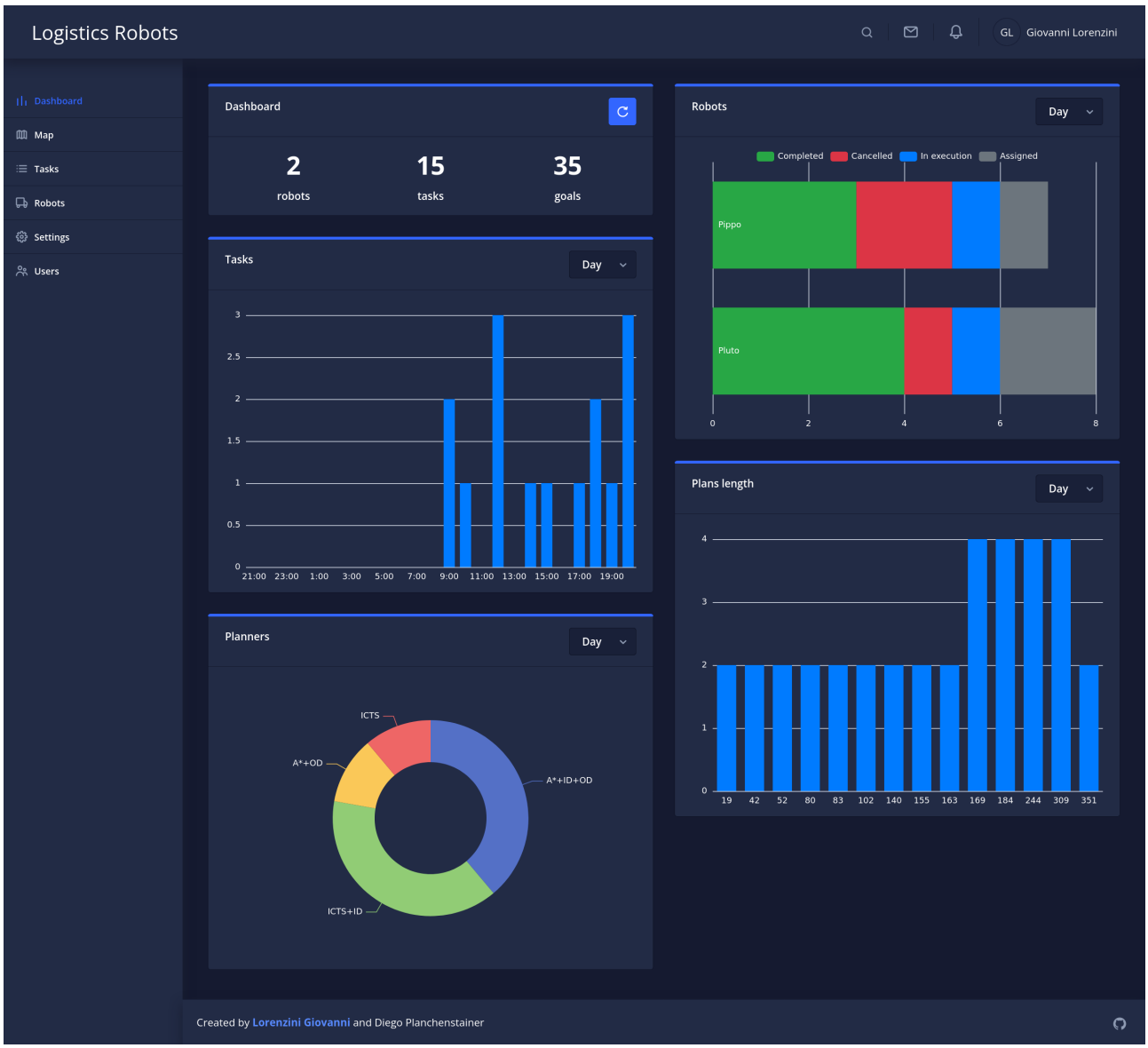


Figure 4.6: Dashboard with the widgets for the statistics.

- The fifth one is a doughnut chart that shows the usage of the various typologies of planners.

Map

The map page (Fig. 4.7) is the most important page for the end user. Here, the user can create a new task by clicking on the *NEW* button and then clicking on the offices that they want the robot to travel by. Every time a room is selected it will be added to the list below the map. Here are reported in order all the offices by which the robot needs to travel. If a wrong room has been added, it can be deleted by clicking on the trash bin icon at the right side of its entry in the list. At last, when the task is completed, by clicking the button *SEND* the task is elaborated by the system and the robots start moving.

By hovering with the mouse over the rooms the office name will be shown. These names are autogenerated but can be changed in order to match the given map.

Over the map, the current location of the robots is shown along with their plans. Each robot and its plan have a unique color to be easily distinguished.

Tasks

The tasks page (Fig. 4.8) summarizes all the tasks presented to the systems. They are ordered from the newest task at the top of the list to the oldest task at the bottom of the list. Each task presents four information and an

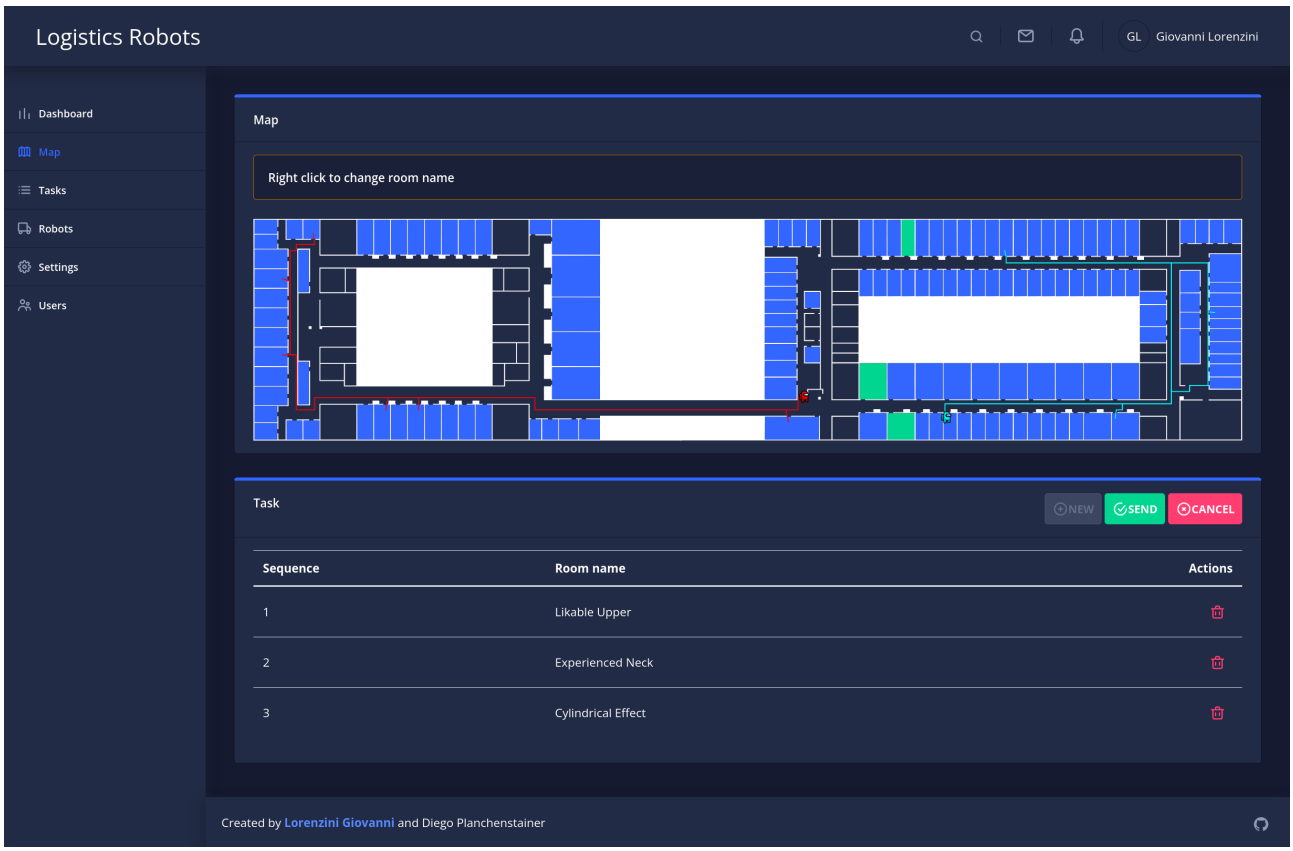


Figure 4.7: Map page with the maps and the robots icons, during the creation of a new task.

action:

- **Status:** the status of the task that can be: not assigned, assigned, in execution, completed, or canceled.
- **Progress:** a progress bar that fills when the goals in the task are accomplished. It is blue when the task is in execution and becomes green when the task is completed or red when the task is canceled.
- **Robot:** robot to which the task is assigned.
- **Goal set:** list of rooms by which the robot has to travel.
- **Actions:** a stop icon that enables the user to cancel the execution of a task is shown for tasks not already canceled or completed.

Robots

On this page administrators can add new robots to the system by defining their parameters: number, name, and position in (x, y, θ). Note that the robot number must be incremented by 1 for every new robot and the names should be different. The properties of a robot are changed on the robot page.

Settings

The settings page (Fig. 4.9) is the most important page for the setup of the system. The settings are divided into 3 sections namely: *Algorithm*, *Robot*, and *Map*.

In the algorithm section, the MAPF algorithm can be selected. Then the rest of the parameters, which are the SAPF algorithm, the cost function, and the heuristic, should be set accordingly.

The robot section consists only in one entry that gives the radius dimension of the robot. This parameter is used in the map decomposition algorithm explained in Sec. 4.3.3 to inflate the map.

In the map sections, the parameters of the map are set and the possibility to upload a new map image is given. The image will then be elaborated to obtain a map as explained in Sec. 4.3.3.

Figure 4.8: Tasks page with the list of tasks and their properties.

At the bottom, there are buttons to save the settings, cancel the changes, reset the settings, and erase the tasks and the plans.

Users

On the users' page a list of the users is displayed along with their properties. In the user page, admins, can activate new users and/or set users to admin. The users can also be deleted. The logged-in user on the other hand can change his password.

4.5 ROS - Robot Operating System

Since the aim of these theses is to integrate MAPF planners into a real system with real agents, ROS is needed to orchestrate all the information required by the robots. First, files for launching the system are needed. Those files describe how the different packages connect among themselves. When launching the system, two possible options are present: either loading a simulated environment or running the robot in the real world. In both cases, two pieces of information need to be provided: the position of the robots and the floor map of the environment. In the case of a simulated environment, also the 3D mesh of the environment is required. These data need to be configured manually whenever the system is deployed in a different location.

The launch file of the robots brings up the Nav2 stack and the localization package. The first provides the capability to move within the environment, and the second takes data from the sensors mounted on the robot and elaborates them to estimate the position of the robot within the environment through *Adaptive Monte Carlo Localization* (AMCL). AMCL uses a particle filter to track the pose of a robot against a known map.

A bridge [3] has been used to communicate between ROS and the Backend.

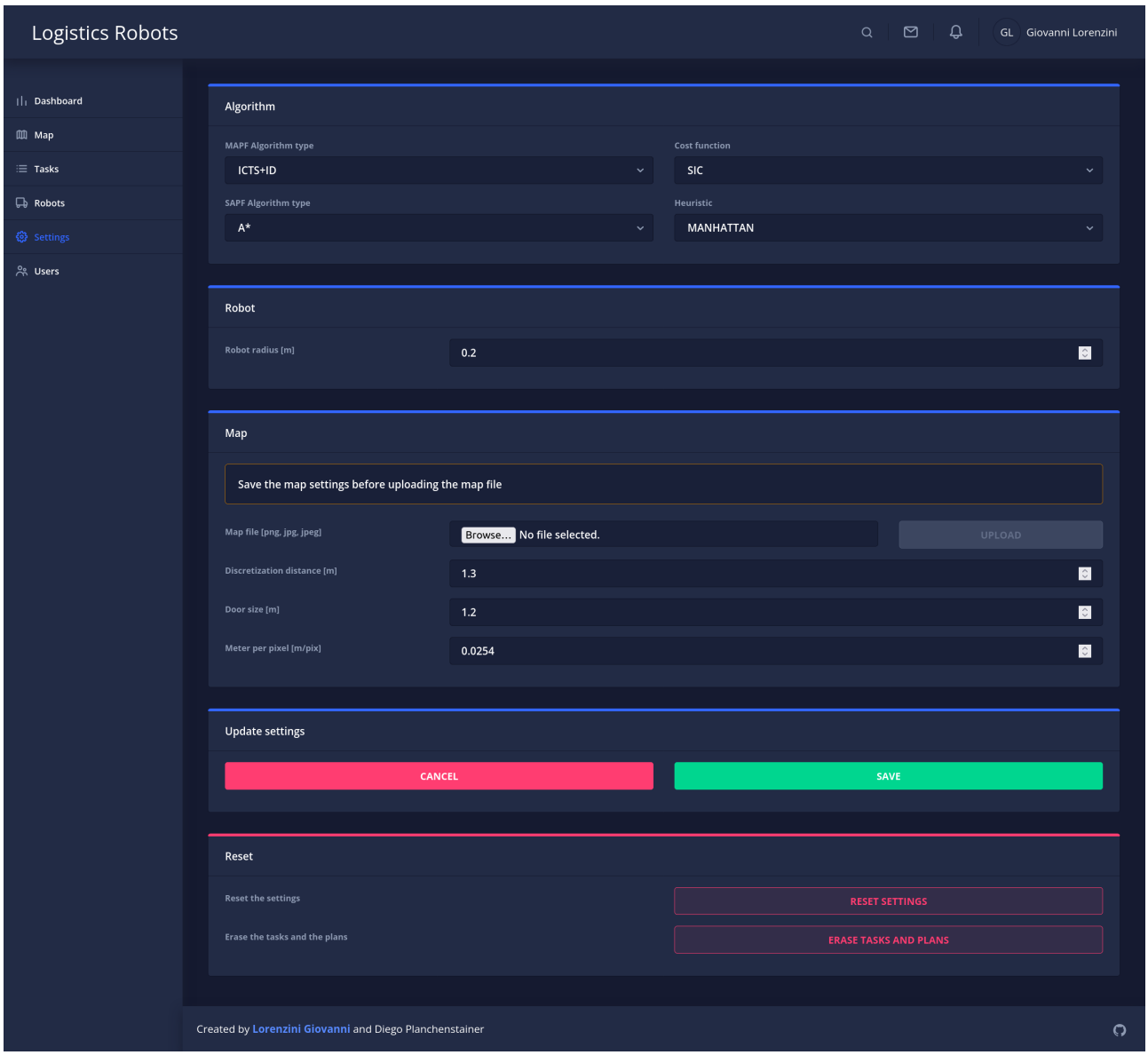


Figure 4.9: Settings page.

4.5.1 Architecture

The ROS architecture part of this project starts from the ROS core that manages all the messages that travel through the system. If the system is simulated ROS needs to interact with Gazebo. Then, for each robot, a series of packages are instantiated: `robot_state_publisher`, `localization`, and `Nav2` stack. It is also possible to swap localization with `SLAM`, but this is not the case. If simulated, another node has to be brung up: `spawn_entity`.

`robot_state_publisher` is responsible for publishing the state of the robot to `tf2`, `localization` implements AMCL to track the pose of the robot, and `Nav2` manages navigation tasks. `spawn_entity` is needed to generate the robot into the simulation environment.

4.5.2 Bridge to Backend

At the time when we needed to build up the communication between ROS and the Backend, different approaches have been tested. In the beginning, rosbridge suite was examined, but it does not provide action interfaces, even though they can be simulated using the hidden topic and the hidden service that composes the action. Then, a *Pull Request* (PR) taken from rosbridge's code repository was inspected. The author proposed to add support for actions, but we found this version buggy. The third approach was to test rclnodejs, this library's drawback was that it needed ROS to compile. In the end, we found a package called ros2bridge that with some modifications to do was the right code for our case.

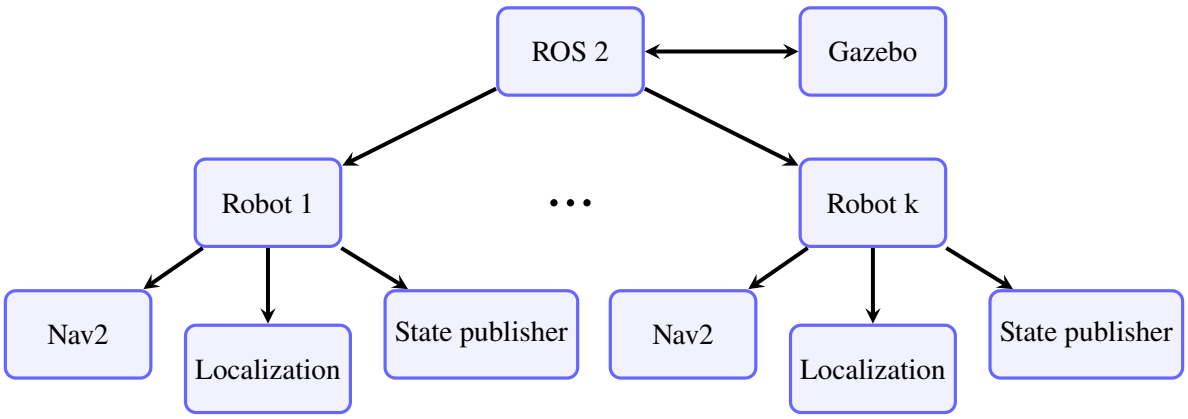


Figure 4.10: ROS architecture.

Rosbridge suite

Rosbridge suite is a meta-package that contains rosbridge_library, rosbridge_server, and rosapi. It supports publication and subscription to ROS topics, ROS parameter settings, and calls to ROS services through a JSON interface. rosbridge_library is the core of the meta-package and is responsible to control ROS interfaces accordingly to the content of the JSON message. rosbridge_server exposes the rosbridge_library by the means of a WebSocket server. In our experiments, we connected the Backend to the exposed WebSocket server on port 9090, sending JSON messages through a client developed in TypeScript by ourselves that will be explained below.

The problem with this approach was that no action interface support was implemented. Recall from Sec. 3.2.3 that actions are built on top of topics and services. With this information, one could try to simulate an action by using these interfaces. Unfortunately, for us, this was not the case.

Rosbridge suite PR with action support

Searching for different approaches to solve the issue of action support, we discovered a PR that added action support. After some testing, it was clear that some bug was present as initially when sending actions to different robots, the sent action overwrote the active action of the other robot. After the author fixed this problem, another issue arose. Feedback and result messages did not come with the right timing, sometimes the system has even lost some messages. This was a major issue that we could not fix, so other approaches were considered.

rclnodejs

Another option that we tried was to employ rclnodejs. This package requires ROS source files to be built. Even though action interface support exists, this option was discarded as we wanted to separate the container of the Backend from the ROS container.

Ros2bridge

In the end, we opted for ros2bridge [3], as it was easier and supported action messages. However, some bugs were present when any type of message carried a list of any type. The code was not able to convert the JSON file to the right message type. Thus, a change in the code to support at least the actions needed has been done. Also, a modification was added to support action deletion. We simply added an if statement for `FollowWaypoints` action and `CancelGoal` service request. In these two cases, we convert the information into the right format and then send it to ROS.

Node.js client

Both rosbridge suite and ros2bridge act as WebSocket servers, therefore a WebSocket is needed to communicate with them. The client has been programmed in TypeScript inside the Backend, using the `ws` library [43]. What we have exposed are some simple classes that wrap the ROS communications, like Topics, Services, and

Actions, creating the right JSON WS messages. In the case of an action, for example, the exposed methods are: `open` that creates a new connection and registers the action client, `call` to send an action message, `cancel` to interrupt an action, `registerCallbacks` that is used to register functions as callbacks for response, feedback, and result messages.

4.5.3 Nav2

Nav 2 [67] was ready to work out of the box. As previously said in Sec. 3.2.3, actions are the preferred interface for long-running tasks, as navigation. By sending the action message interface `FollowWaypoints.action` to the system, robots were able to follow multiple waypoints one after the other. This action interface requires a list of `geometry_msgs/PoseStamped`, returns as feedback the number of the current waypoint, and gives the list of missed waypoints as the result. The `geometry_msgs/PoseStamped` message is composed by an `header` and a `pose`. The first comprehends the `timestamp` of the message and the `frame_id` related to the `pose`. The second carries the information related to the `position` in (x, y, z) coordinates and the `orientation` expressed in quaternions. The position coordinates are extracted from the graph node information. The orientation is computed by the Backend by converting to quaternions the angle between the current position and the next position.

`FollowWaypoints` action module implements waypoint following using the `NavigateToPose` action server. In addition to waypoint navigation, it also hosts a waypoint task executor plugin, used to perform custom behavior at the waypoint. The three standard plugins are `WaitAtWaypoint`, `PhotoAtWaypoint`, and `InputAtWaypoint`. One possible option is to stop the navigation if one location is not reachable, we decided to disable this option.

Recall that it is assumed that the robots move synchronously between points in the graph. Whenever the list of poses contains the same pose twice or more, a major problem emerges. This condition arises because of the MAPF implementation that in some cases, i.e. when a collision will happen, tells one robot to stay at its previous position to avoid crashes, resulting in duplicated poses. Here, the action processes the first waypoint and then, since the next pose is equal, marks the second waypoint as completed and immediately moves on. This causes the robots to collide with each other. Since the objective of the planner is to compute collision-free paths, we do not want this to happen.

Two main solutions to the problem were pursued. The first was to implement a behavior tree navigator able to discern whether the next waypoint is equal to the current one and, in that case, wait in position. This approach was too complex, but we acknowledge that it is the right way to solve this task. The second solution was to modify the already existent `FollowWaypoint` action and check for duplicate poses. This was the approach that we choose. Internally, the action checks if the current goal (the waypoint towards which it is navigating) is completed. If so, the plugins, are activated. In this location of the code, we added a checking operation that triggers a waiting timer if there are repeated waypoints.

Both approaches have a major drawback: the time that the robot waits in place. As a consequence of the synchronous movements, it is fundamental that the waiting time is equal to the time taken by a robot to perform a movement between two waypoints. Time misalignment could cause severe problems to the system, resulting in collisions. The time that a robot should wait in position is encoded in a parameter called `wait_time_if_equal`. This parameter value was computed through an experiment. Two robots performed the same set of movements: the first starts immediately, the second waits in position for a n number of time steps. Then, the difference between the two times of arrival $(\Delta t_1, \Delta t_2)$ is divided by the number of time steps n waited. This is done to obtain a more accurate estimation of the waiting time. The overall formula is:

$$\texttt{wait\_time\_if\_equal} = \frac{\Delta t_1 - \Delta t_2}{n}$$

As a consequence of this process, the parameter value depends on the discretization distance of the map. This implies that every time the discretization distance is changed the parameter needs to be modified. At this stage of development, the system is able to work, but will eventually desynchronize as the estimation it is not perfect and there could be some delays in the robot's movements for different reasons. An improvement could be to communicate with other robots and wait for the completion of all the actions before starting a new one.

Many parameters to configure Nav2 are available. We have added a plugin called "Rotation Shim Controller" that makes the robot spin in place heading to its next location and then travel to its goal. This was done because we require that robots travel along the edges of the graph. If this controller is not added the robots stop a wheel

to turn in the desired direction, moving far from the node. Also, the movements result less smooth, hence the decision to add the plugin.

4.5.4 Simulator

The software used to simulate the environment is open source and is called Gazebo [56]. It is a 3D simulator that integrates *Open Dynamins Engine* (ODE) physics engine, *Open Graphics Library* (OpenGL) rendering, and support code for sensor simulation and actuator control. Interesting sensors often modeled are cameras, LiDARs, and Kinect-like sensors.

ODE is a physics engine that is composed of a rigid body dynamic simulator and a collision detection engine. It has been used for different applications, from games to robotics simulations. For the last category, ODE presents some drawbacks as poor support for joint-damping and the friction approximation method.

OpenGL is an API for rendering 2D and 3D vector graphics used to leverage GPU capabilities, to obtain hardware-accelerated rendering. The development started in 1991 and the last update went back to 2017, since the company producing it (Silicon Graphics, Inc.), dropped it in favor of Vulkan API [60]. Thus, advanced GPU capabilities such as ray tracing are not supported. OpenGL capabilities allow Gazebo to obtain realistic rendering of the environment. The 3D rendering engine used is called OGRE.

Gazebo has been adopted for multiple challenges that required complex tasks and environments, such as DARPA Robotics Challenge, NASA Space Robotics Challenge, and Toyota Prius Challenge.

Since ROS needs to exchange information with Gazebo, a bridge that converts Gazebo's protobuf messages to ROS messages and vice-versa was created since ROS Melodic.

To simulate robots and the environment, 3D meshes are needed. For the TurtleBot3 meshes are provided within the turtlebot_description package within their repository [65]. The meshes of the university's floor were provided to us at the start of the thesis project. It was an STL file generated after a SLAM operation of the building. If one wants to obtain the meshed it can either build them with CAD and convert them into a `.dae` file or generate the map through a SLAM procedure. To convert it into a `.dae` file, the STL was imported into Blender and then exported into the correct format. In Blender, one could also perform different operations such as refining the mesh in case it is not satisfactory. In Fig. 4.11, the simulated environment of the university is shown together with two TurtleBots. Blue lines represent the laser beams produced by the simulated LiDAR sensors mounted on top of the robots.

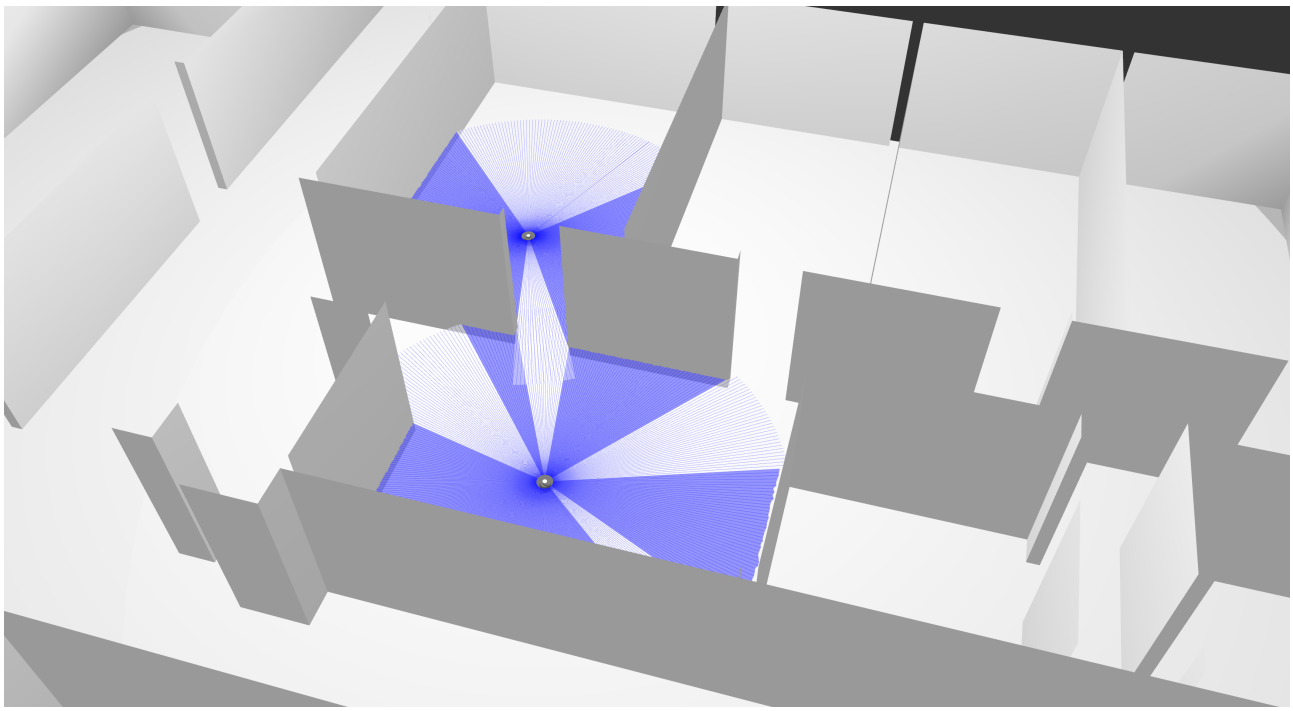


Figure 4.11: Gazebo simulation environment with two robots.

5 SAPF Algorithms

Here we are going to explain how the *Single-Agent Path Finding* (SAPF) algorithms work and how they have been implemented in the Multi-Agent Open Framework.

A single-agent path finding algorithm aims at searching the best path between two graph vertexes. The found plan is therefore composed of a list of vertexes that the agent has to follow.

In the MAOF framework, a pickup and delivery task is considered, which is composed of multiple goals that should be visited in sequence. It can be accomplished by applying the SAPF algorithm to pairs of goal vertexes (Alg. 3).

Algorithm 3: Multi-goal single-agent path finding

```
Input: G=(V,E): graph
Input: goals: list of goals
;
Result: path: list of graph vertexes
;
begin
    for i ← 1; i < size(goals); i ← i+1 do
        tmpPath ← SAPF(G,goals[i-1],goals[i]);
        Append tmpPath to path;
    return path;
```

5.1 Dijkstra

Dijkstra is an optimal SAPF algorithm developed by E. W. Dijkstra in 1956 and published in 1959 [78]. It accounts for the weights (length or cost) of the graph edges in order to find the optimal path. In particular, a path to reach a vertex is discarded if a better (shortest) one is found. The algorithm chooses to expand always the node with the smaller cost. In our case, the graph edges are typically of unitary length, therefore the algorithm degenerates to the *Breadth-First Search* (BFS).

In more details the algorithm (Alg. 4) works as follow:

1. The initial node `I` is added to the open set (`openSet`) that is implemented by using a priority queue.
2. A map (`gScore`) for tracking the cost to reach a node n_i from the initial node `I` is used. Initially, it only contains the cost to reach `I` from `I`, which is equal to 0.
3. Until the open set is not empty, the node (`C`) with the lowest cost, present in the priority queue, is extracted and deleted from the open set. The node `C` is also inserted in the closed set, so the algorithm is not going to expand it again.
4. It is necessary to check if the current node `C` is the same as the desired final node `F`. In case it is true, the optimal path can be reconstructed with the help of a map between a node n_i and the node n_j used to reach n_i with the least cost. The founded path is returned and the execution terminates.
5. Now, it is time to look at every neighbor `N` of the current node `C`. In particular, the cost `gScoreProposal` to reach `N` from `I` through `C` is calculated. If this cost is less than an eventually present cost in the map `gScore` or if a previous cost is not available, then:
    - the node `C` is set as the best predecessor of `N`;

- the cost to reach `N` is also stored in the `gScore` map;
- the neighbor `N` is added to open set with priority `gScoreProposal`;

Algorithm 4: Dijkstra

```
Input: G=(V,E): graph
Input: I: initial vertex
Input: F: final vertex
;
Data: openSet: priority queue with nodes to be expanded
Data: closedSet: set of already expanded nodes
Data: cameFrom: maps predecessor of node n_i to node n_j
Data: gScore: maps cheapest cost to reach node n_i
;
Result: path: list of graph vertexes
;
begin
    openSet[I] ← 0;
    gScore[I] ← 0;
    ;
    while openSet not empty do
        C ← first node in openSet;
        Delete C from openSet;
        Add C to closedSet;
        ;
        if C = F then
            while C ≠ I do
                Prepend C to path;
                C ← cameFrom[C];
            return path;
        ;
        foreach neighbor N of C not in closedSet do
            gScoreProposal = gScore[C] + cost(C,N);
            if gScoreProposal < gScore[N] then
                cameFrom[N] ← C;
                gScore[N] ← gScoreProposal;
                openSet[N] ← gScoreProposal;
    return path not found;
```

The Dijkstra algorithm has been implemented in MAOF using C++. The following optimizations have been applied:

- The open set, closed set, came from map, and costs map can be represented as an array each, with as length the number of vertexes of the graph. We choose to not pursue this path but to use other data structures that memorize only the interesting nodes as typically only a small part of the graph is visited by an agent.
- The closed set is implemented using the set from the library unordered_dense [48], the fastest hashmap implementation for C++ [49]. It is typically possible to access an element in $O(1)$ instead of in $O(\log n)$ by a regular set.
- The costs map `gScore` is a map where the key is a graph node and the value is the least cost to reach that node from `I`. The `cameFrom` map has as key a graph node n_i and as value the graph node n_j that allows

reaching n_i with the least cost starting from `I`. Both these maps are implemented using the hashmap library unordered_dense [48].

- The priority queue is implemented using the one provided by C++ standard library. The queue is inversely ordered with respect to the cost to reach a certain node n_j from `I`. The ties in cost are broken by the time of insertion of a node in the queue, in order to resemble a *Depth-First Search* (DFS) for equal costs.

5.2 A*

If the coordinates of the graph vertexes are available, as in our case, it is smart to use them. This enables us to construct a heuristic that is an estimate of the length of the path connecting two vertexes. The usage of a heuristic speed up the search process. The heuristic needs to be admissible to guarantee optimality, in practice it means that the heuristic never overestimates the cost of reaching a goal [9]. The time complexity depends on the used heuristic. An upper bound to the number of expanded nodes is given by:

$$O\left(b^d\right) = O\left(5^d\right)$$

where b is the branching factor, in our case 5, and d is the depth of the solution [71]. A good heuristic aims at keeping the effective branching factor low, approaching the optimal branching factor $b^* = 1$. In our case we have implemented two heuristics:

- **Euclidean distance:** is the shortest distances between two nodes n_i and n_j, computed as:

$$dist = \sqrt{(n_i^x - n_j^x)^2 + (n_i^y - n_j^y)^2}$$

- **Manhattan distance:** is defined as the sum of the absolute differences of the coordinates of two graph nodes n_i and n_j. Our robots move only parallel to the X and Y axes and therefore the Manhattan distance is more accurate. It is calculated as follows:

$$dist = \left|n_i^x - n_j^x\right| + \left|n_i^y - n_j^y\right|$$

Both are admissible heuristics and therefore the solution found by the A* algorithm will be optimal. In A*, the heuristic is computed between the neighbor `N` of the current node and the final node `F` and used to direct the search towards the goal.

While A* is similar to Dijkstra (Sec. 5.1), there are the following differences, as can be seen in Alg. 5:

- A* exploits the heuristic between the neighbor `N` of the current node and the final node `F`. The priority of the node `N` in the open set priority queue represents the cost to reach node `N` from node `I` via node `C` plus an estimation (heuristic) of the length of the path between node `N` and node `F`.
- Dijkstra can be seen as an A* with the heuristic always equal to 0 [37].
- In Dijkstra the node that is going to be expanded is chosen as the one that can be reached with the least-cost path. In A*, the node that is going to be expanded is chosen as the one that seems to have the most promising (cheap) path to reach the goal.

This single-agent A* algorithm is used as a base for the multi-agent A* algorithm (Sec. 6.1) and is also called by both multi-agent A* (Sec. 6.1) and ICTS (Sec. 6.2) to be used as a heuristic.

Algorithm 5: A*

Input: `G=(V,E)`: graph
Input: `I`: initial vertex
Input: `F`: final vertex
;
Data: `openSet`: priority queue with nodes to be expanded
Data: `closedSet`: set of already expanded nodes
Data: `cameFrom`: maps predecessor of node n_i to node n_j
Data: `gScore`: maps cheapest cost to reach node n_i
;
Result: `path`: list of graph vertexes
;
begin
    `openSet[I]` ← `heuristic(I,F)`;
    `gScore[I]` ← 0;
    ;
    while *`openSet` not empty* **do**
        `C` ← first node in `openSet`;
        Delete `C` from `openSet`;
        Add `C` to `closedSet`;
        ;
        if *`C` = `F`* **then**
            while *`C` ≠ `I`* **do**
                Prepend `C` to `path`;
                `C` ← `cameFrom[C]`;
            return *`path`*;
        ;
        foreach *neighbor `N` of `C` not in `closedSet`* **do**
            `gScoreProposal` = `gScore[C]` + `cost(C,N)`;
            if *`gScoreProposal` < `gScore[N]`* **then**
                `cameFrom[N]` ← C;
                `gScore[N]` ← `gScoreProposal`;
                `openSet[N]` ← `gScoreProposal` + `heuristic(N,F)`;
    return *path not found*;

6 MAPF Algorithms

In this chapter, an in-depth discussion about the implemented MAPF solvers is presented. Details on the implementation of the algorithms and tricks used to obtain these results will be illustrated here. In this team effort, two planners were developed: multi-agent A* and ICTS. While Lorenzini worked on multi-agent A*, Planchenstainer developed ICTS. The reader is advised to check the thesis on the planner of their interest.

6.1 A*

The multi-agent A* is the algorithm on which my thesis is focused. This algorithm and this analysis have been written by myself. This implementation consists of $\sim$ 1000 lines of code spanning over 4 classes. The usage of C++, as the programming language, is due to the integration to Saccon's code and its high performance.

Multi-agent A* works similarly to single-agent A* (Sec. 5.2), but with some differences as the state representation and the heuristic. The state is the union of the states of every agent leading to a k-array of positions. In a 4-connected graph, and considering the operation of standing still, the possible moves for every agent are 5. Therefore, it leads to a total number of possible future states of 5^k. However, not all moves are legal, for three reasons: there is not an edge connecting the vertex to another one, a swap conflict happens meaning that two agents exchange their position along the same edge, or there is a vertex conflict meaning that two agents go to the same vertex at the same time.

Another difference, compared to the single-agent variant, is the computation of the heuristic and the moving cost. What it is considered is not a distance between vertexes expressed in meters, but the number of edges that an agent should cross to go from one vertex to another one. Therefore the heuristic, to be admissible, should be the lowest number of hops between two vertexes. To do this estimation, mainly two options are available, where the first one is the Manhattan distance with a factor of conversion between meters to number of edges. This metric can be computed fastly but is inaccurate in particular on large and complicated maps with many obstacles or walls. So, the choice fell on the second option that performs a single-agent search, using an algorithm such as Dijkstra or A*, to get the distance between two vertexes expressed as the number of edges composing the shortest path connecting them.

The MAOF framework is designed to accept in input, for each agent, a list of goals that it must reach in the given order. This adds a level of difficulty that cannot be simply solved by concatenating multiple plans. Each agent takes a different amount of steps to reach a goal, leading to the fastest agents waiting for the slowest ones. This is clearly sub-optimal, as one agent cannot proceed to the next goal until all the agents have reached the current goal. To solve this problem, the idea of using some information to store the goal towards which an agent should go is taken from [16, 2]. This is achieved by storing, in the state, a *label*, so the index of the goal, for each agent.

In the following sections are reported a detailed explanation of the implementation of the multi-agent A* algorithm, the adopted solutions, and optimizations.

6.1.1 State representation

During the search, the algorithm needs to know the present state, the visited states, the states to be visited, and the cost to visit a state. Therefore, the creation of a class to store this information is natural, I called it `MAStarNode`. The state does not store directly if it is visited or not and the cost to reach it, but it can be used as the key in a map structure to retrieve the required piece of information. As stated above, the locations of the agents are required to compose a state. The location of every agent is stored in a vector of graph nodes, with size the number of agents k. A vector of integers with size k is used to store the labels, where each label is the goal index j that the agent a_i is trying to reach. The last piece of information stored in the state is an integer called `partialNodeAgentIndex` used by the Operator Decomposition improvement explained below.

The class `MAStarNode` contains many methods, which are mostly getter and setter, and the one for obtaining a list of neighboring nodes. In Alg. 6 the algorithm used to compute the neighbors of a state is reported. This method computes all the possible valid combinations of the graph nodes for every agent. Therefore, it is possible

to estimate the number of neighbors as $\sim 5^k$. Typically, this should be solved using a recursive algorithm, but I choose to use the approach from [36]. It basically transforms the algorithm from recursive to iterative. The remainder between the current index (stored in a temporary variable) and the size of the current neighbor is used as the index for the graph node used to compose the neighbor. Then, the temporary variable is divided by the number of single-agent neighbors and it is used in the next cycle to compute the new index. Using this approach, a speedup of the generation of the neighbors has been noticed, improving the performance of the whole multi-agent A* algorithm, since the method to compute the neighbors is called many times.

Algorithm 6: Method to get a list of neighbors

```
Input: G=(V,E): graph
;
Data: sNeighbors: matrix of single-agent neighbors (for each agent)
Data: states: matrix of multi-agent neighbors
;
Result: neighbors: list of neighbors of type MAStarNode
;
begin
    foreach node in nodes do
        Append neighbors of node to sNeighbors;
    ;
    numberOfStates ← 1;
    foreach sNeighbor in sNeighbors do
        numberOfStates ← numberOfStates * size(sNeighbor);
    ;
    for i ← 0; i < numberOfStates; i ← i+1 do
        temp ← i;
        state ← {};
        foreach sNeighbor in sNeighbors do
            index ← temp % size(sNeighbor);
            temp ← temp / size(sNeighbor);
            Append sNeighbor[index] to state;
        Append state to states;
    ;
    foreach state in states do
        Append new MAStarNode(state, labels) to neighbors;
    return neighbors;
```

A last interesting method of the `MAStarNode` class is the one used to compute the hash. All the information present in the class should be part of the hashing function so the objects can be distinguished from others. A hash function is a function that maps data of arbitrary size to a single value. It is used to store and access data stored in a hash table, and therefore should be designed to limit the probability of collision. A collision happens when two different objects are mapped to the same value. Therefore, a function used to combine all the class properties has been written, taking inspiration from [66]. This code is designed to start with the hash of a class property, and then progressively combine it with the hash of the other class properties, in order to obtain a single hash. The found hash should possibly be unique depending on the combination of the input data.

6.1.2 Basic algorithm

In this section, the standard multi-agent A* and its implementation details are explained, while Standley enhancements, multiple goals support, and dynamic weighting are explained later. The structure of code remains similar to the one of the single-agent A*, but it is adapted to deal with the different number of agents, the new state structure defined in Sec. 6.1.1, and the conflicts.

In Alg. 7 the skeleton algorithm of this multi-agent A* implementation is reported. It works as follows:

Algorithm 7: Multi-agent A*

```
Input: G=(V,E): graph
Input: I: list of initial vertexes
Input: F: list of final vertexes
;
Data: openSet: priority queue with states to be expanded
Data: closedSet: set of already expanded states
Data: cameFrom: maps predecessor of state n_i to state n_j
Data: gScore: maps cheapest cost to reach state n_i
;
Result: solution: list of paths composed of graph vertexes
;
begin
    openSet[I] ← heuristic(I,F);
    gScore[I] ← 0;
    ;
    while openSet not empty do
        C ← first node in openSet;
        Delete C from openSet;
        Add C to closedSet;
        ;
        if C = F then
            while C ≠ I do
                Prepend C to solution;
                C ← cameFrom[C];
            return solution;
        ;
        foreach neighbor N of C not in closedSet do
            if conflicts(C,N) then
                continue;
            gScoreProposal = gScore[C] + cost(C,N);
            if gScoreProposal < gScore[N] then
                cameFrom[N] ← C;
                gScore[N] ← gScoreProposal;
                openSet[N] ← gScoreProposal + heuristic(N,F);
    return solution not found;
```

1. The initial state `I` is added to the open set (`openSet`) with priority equal to the heuristic between `I` and `F` (the number of edges in the shortest path connecting them, obtained with a SAPF search). The open set is implemented using a priority queue.

2. A map (`gScore`) for tracking the cost to reach a state n_i from the initial state `I` is used. Initially, it only contains the cost to reach `I` from `I`, which is equal to 0.

3. Until the open set is not empty, the state (`C`) with the lowest cost, present in the priority queue, is extracted and deleted from the open set. The state `C` is also inserted in the closed set, so the algorithm is not going to expand it again.

4. It is necessary to check if the current state `C` is the same as the desired final state `F`. In case it is true, the optimal path can be reconstructed with the help of a map structure that maps a state n_i to the state n_j used to reach n_i with the least cost. The found path is returned and the execution terminates.

5. Now, it is time to look at every neighbor `N` of the current state `C`. Firstly, it is necessary to check for the presence of vertex or swap conflicts between `N` and `C`. If the outcome indicates that some conflicts are present, the neighbor `N` is skipped and the execution continues with the next neighbor. If there are no conflicts, the cost `gScoreProposal` to reach `N` from `I` through `C` is calculated. If the cost `gScoreProposal` is less than a possibly present cost in the map `gScore` or if a previously computed cost in the map `gScore` is not available, then:

   - the state `C` is set as the best predecessor of `N`;
   - the cost to reach `N` is also stored in the `gScore` map;
   - the neighbor `N` is added to open set with priority `gScoreProposal`.

In the Alg. 7 the computation of the cost and of the heuristic are shadowed behind two functions that in this case are different with respect to the single-agent A*. Both the cost and the heuristic can be calculated using the SIC or MKS techniques. SIC is the sum of individual costs where all the agents' costs/heuristics are considered and summed up. MKS is the makespan and it considers only the cost/heuristic that is higher than the others. Now, only the computation of the SIC variant is explained, as the other one is similar.

The cost is the sum of the cost of every agent. The SIC metric can be computed as:

$$SIC(\Pi) = \sum_{1 \leq i \leq k} |\pi_i|$$

where π_i is the path of an agent and Π the joint plan. Here the actual distance of the movements of the agent is not considered. What is considered is if the agent moves or not. Initially, the cost to move was set to 1 while the cost to stand still was set to 0. Later, they have been both set to 1, so to entice robots to move towards the goal. This decision is motivated by these observations that have been made, changing the cost of staying still to:

- **0:** the algorithm finds a path in a very small period of time but the path is sub-optimal because the agents can stand still as it cost nothing to them. An optimal solution is desirable therefore this option has been discarded.
- **0.001:** the algorithm takes a lot of time and produces a sub-optimal solution.
- **0.999:** the execution time is more or less like the above options and the path is optimal.
- **1:** the chosen option. The execution is neither fast nor slow and the found solution is optimal.

The multi-agent heuristic is the sum of the single-agent heuristic of every agent. The single-agent heuristic is expressed as the number of movements required by the agent to reach the goal vertex, resulting in a multi-agent heuristic computed as:

$$H(A) = \sum_{1 \leq i \leq k} h(a_i) = \sum_{1 \leq i \leq k} |\pi'_i|$$

where π'_i is the shortest path between the current vertex and the goal vertex. Therefore, a SAPF solver is used to compute a path between the current location of the agent and its goal. In the end, what is considered is the length of the found path.

To make the algorithm run fast these code optimizations have been adopted:

- The priority queue is implemented using the one provided by C++ standard library. The queue is inversely ordered with respect to the cost to reach a certain state n_j from `I`. The ties in cost are broken by the time of insertion of a state in the queue, in order to resemble a *Depth-First Search* (DFS) for equal costs.
- A map from the library unordered_dense [48] is used to store the computed heuristics in order to simply access it, instead of re-computing it, in case it is needed again. This trick has a huge positive impact on performances since a single-agent graph node is analyzed many times during the search.
- The closed set is implemented using a set of unordered_dense library. It is typically possible to access an element in $O(1)$ instead of in $O(\log n)$ by a regular set.
- The costs map `gScore` is a map where the key is a state and the value is the minimum cost to reach that state from `I`. The `cameFrom` map has a state n_i as key and the state n_j that allows reaching n_i with the least cost starting from `I` as value. Both these maps are implemented using the hashmap library unordered_dense.

To compute an estimate of the time complexity of the A* algorithm is first necessary to identify the branching factor. In this standard algorithm, the branching factor is:

$$b = m^k = 5^k$$

where $m = 5$ are the possible actions and k is the number of agents. Then the time complexity can be estimated in:

$$O\left(b^d\right) = O\left(5^{k^d}\right)$$

where b is the branching factor and d is the depth of the solution [71]. The effective branching factor b^* should be lower than b thanks to the usage of the heuristic.

6.1.3 Multiple goals

In the MAOF framework a list of goals, for every agent, is given, and it needs to be considered during the search. A label for every agent, memorizing this piece of information, is therefore required. In fact, in the state representation class `MAStarNode`, an array of labels is present.

How it works is quite simple: the labels are initialized all to 1, meaning that the agents should reach the first goal in their list. Then the standard algorithm is modified in order to update the label when a neighbor is being expanded. This works by checking if the goal that the agent is navigating to is the same as the agent's position in the neighbor node. If the check is positive, then the label for that agent is incremented by one.

Another change to the standard algorithm is necessary for the part that checks if the goals are satisfied. This is accomplished by looking at the labels. If the label of each agent is equal to the size of the list of goals of that agent, then the agents have completed the planning.

The last thing that needs to be changed is the way the heuristic is computed. The single-agent path from which the length is taken should pass through all the goals that it has not already reached. This is accomplished by taking into consideration only the goals with index i that satisfy $i \geq l$ where l is the label.

6.1.4 ID - Independence Detection

Independence Detection (ID) is one of the improvements proposed by Standley in [74]. The idea is to plan separately the path of each agent in order to have a problem that grows linearly, instead of exponentially, with respect to the number of agents. This clearly cannot be applied to all agents indiscreetly, but only to the ones that do not have a conflict with others. To have the most benefits, a partitioning of the agents into disjoint sets is necessary. In Alg. 8 the implemented algorithm is reported. It works as follows:

1. The single-agent path is computed for each agent. Then, it is stored in the `solution` list.
2. The conflicting groups' list is initialized with each agent in a separate group.
3. A do-while cycle, running until no more conflicts are present, is employed. Inside the cycle the operations are:

(a) Check if there are some conflicts between the found paths. If there is a conflict, the groups of which the conflicting agents are part are merged into a single group.

(b) If no conflicts have been found then the solution is valid and can be returned.

(c) If there are conflicts, a new solution for the conflicting groups should be searched. As typically more than one group is present and they are not related, this task can be easily parallelized. In fact, for each conflicting group, a new thread is created in which the function to find the paths is called.

Algorithm 8: Independence detection

```
Input: agents: list of agents
;
Data: conflictingGroups: group of conflicting agents
Data: conflicts: boolean variable indicating if there are conflicts
;
Result: solution: list of paths composed of graph vertexes
;
begin
    foreach agent in agents do
        solution[agent.ID] ← SAPF(agent);
        Append agent.ID to conflictingGroups;
    do
        conflicts ← false;
        for i ← 0; i < size(solution); i ← i+1 do
            for j ← 0; j < size(solution); j ← j+1 do
                if conflict(solution[i],solution[j]) then
                    groupI ← findGroup(i);
                    groupJ ← findGroup(j);
                    In conflictingGroups merge groupI and groupJ;
                    conflicts ← true;
        ;
        if not conflicts then
            return solution;
        ;
        foreach conflictingGroup in conflictingGroups do
            paths ← conflictingAgentsSolve(conflictingGroup);
            for i ← 0; i < size(conflictingGroup); i ← i+1 do
                solution[conflictingGroup[i]] ← paths[i];
    while conflicts;
```

6.1.5 OD - Operator Decomposition

Operator Decomposition (OD) is the other improvement proposed by Standley in [74]. This optimization aims at reducing the branching factor from 5^k to 5, however increasing the depth of the search by a factor k. In this A* algorithm with OD, the branching factor is:

$$b = m = 5$$

where $m = 5$ are the possible actions. Then the time complexity can be estimated in:

$$O\left(b^{kd}\right) = O\left(5^{kd}\right)$$

where b is the branching factor and d is the depth of the solution [71, 74]. The effective branching factor b^* should be lower than b thanks to the usage of the heuristic.

This is achieved by employing a new representation of the state space, where only one agent at a time can make a move. In particular, each state stores inside itself a variable called `partialNodeAgentIndex` to keep track of which agent should move. At each time step, the agent with index `partialNodeAgentIndex` can make a move, while others stand firm in their positions. Then the index variable is increased so that the next agent can move. After moving the last agent the variable is reset to the index of the first agent. Two kinds of states can be identified, the intermediate states where only a few agents have taken a move, and the standard states where all the agents have taken a move.

In Alg. 9 the method used to get the neighbors of the current state, when the operator decomposition is enabled, is reported. It works as follows:

1. A new index is computed by increasing the index of the current state by 1 and then diving it by the number of agents and keeping the remainder.
2. The neighbors of the vertex on which the selected agent is present are inserted into the `sNeighbors` list.
3. A for cycle is used to consider every neighbor present in the above list. To construct the new state, a copy of the current state is made. Then, the position of the selected agent is changed with the found neighbor. Another change to the state regards the `partialNodeAgentIndex` that is set equal to the index used to find the neighbors. Lastly, the new state is added to the list of neighboring states.

Algorithm 9: Method to get a list of neighbors with OD enabled

```
Input: G=(V,E): graph
;
Data: index: index of the current agent
Data: sNeighbors: list of single-agent neighbors (for current agent)
;
Result: neighbors: list of neighbors of type MAStarNode
;
begin
    index ← (partialNodeAgentIndex + 1) % size(agents);
    sNeighbors ← neighbors of nodes[index];
    ;
    foreach sNeighbor in sNeighbors do
        neighbor ← this state;
        neighbor.agentsLocation[index] ← sNeighbor;
        neighbor.partialNodeAgentIndex ← index;
        Append neighbor to neighbors;
    return neighbors;
```

To guarantee an optimal solution, the OD implementation cannot be naive, meaning that the conflicts check cannot work in a simple way. In particular, the agents should be allowed to make some moves that seem to be illegal. The agents that have not already taken a move should not be part of the conflicts check, as they can later move to another position to avoid the collision. The algorithm to check the collisions has been modified in order to check for vertex conflicts between the current agent and each agent with an index lower than the current one. The check for the swap conflicts is performed looking not at the previous state but at the previous standard state, considering that the actual moves are then always performed considering the standard states.

With OD enabled, the calculation of the cost is simplified given that only a single agent can move at a given time. A different cost can be assigned in case the agent performs a movement or stays in its position. After some tests, it was decided to set the cost to 1 for both types of actions.

The last change required to the standard algorithm is in the part of the reconstruction of the solution. Initially, all the states are considered and a path is built, then the problem is that only one agent moves in each step, therefore the intermediate states need to be filtered out, leaving only the standard states.

6.1.6 Dynamic Weighting

The algorithm written until now is optimal, meaning that it eventually finds the shortest paths used by the agents to reach their goals. A drawback is that the search speed is quite low leading to a high computational time. Therefore, to all the previously applied optimizations, the usage of dynamic weighting is added.

If in the computation of the cost function the heuristic is scaled by a factor $\epsilon > 1$ it is then possible to control how important is the future estimated cost with respect to the past real cost. The cost function for the weighted A* [32] is computed as:

$$f(n) = g(n) + \epsilon h(n)$$

By increasing ϵ the heuristic is no anymore admissible but it gains importance in the calculation of the cost function. The agents move faster towards their goals, leading to a path that is at most ϵ times longer than the optimal one.

To have the speedup benefits without obtaining a too-long plan Pohl invented the dynamic weighting technique [33]. It starts with weight ϵ, which is then decreased during the search as the goal is approached. Therefore the weight $w(n)$ is a function that depends on the state n. The cost function for the dynamic weighted A* is computed as:

$$f(n) = g(n) + (1 + \epsilon w(n))h(n)$$

where $w(n)$ is computed as:

$$w(n) = \begin{cases} 1 - \frac{d(n)}{N} & \text{if } d(n) \leq N \\ 0 & \text{otherwise} \end{cases}$$

where $d(n)$ is the depth of the search and N is the expected length of the path.

The dynamic weighting technique has been used with the parameter $\epsilon = 0.5$ which gave good results in some empirical tests. All the benchmarks reported in Chapter 7 are performed with the A* using the dynamic weighting technique, in fact, it is possible to see that some of the paths found are slightly longer with respect to the ones found by ICTS.

6.2 ICTS - Increasing Cost Tree Search

See Planchenstainer's thesis for an in-depth analysis of our implementation of the ICTS algorithm.

7 Experimental Results

A series of tests have been carried out to assess the performances of the different planners. The parameters that influence the performances are mainly three: the number of agents, the number of tasks, and the number of conflicts. Two main types of experiments have been run out: one in which goals are selected randomly among the offices present in the building and one where the problem scenario is handcrafted to evaluate performances in specific configurations. Since the system is used to transport packets, it must respond to requests fast enough for the user not to prefer to deliver the packets themselves. This aspect is taken into account by setting a timeout in the experiments. Whenever the planner takes more than one minute, the process is halted and the task is considered failed. The discretized map results in a graph of 1183 nodes, with 3312 edges.

Different data are collected during the execution. The most relevant one is the time taken to complete the task (capped at 1 min). Then, path length, number of conflicts, and the dimension of the largest independent subgroup are recorded. The path length is meaningful because it gives a measure of the number of nodes that the planner needs to expand. The number of conflicts gives an insight into how difficult the problem is. This value is computed by looking at how many repetitions are present in the single-agent plans since usually, a robot stays still whenever a conflict is about to happen. In this way, only a lower bound on the conflicts is estimated, since an agent can also move to free the way to another and then return to its position. The last metric is the dimension of the largest independent subgroup. This information is extremely important for cases where *Independence Detection* (ID) is active. In fact, the search complexity is the complexity of a problem with a number of agents equal to the largest group. Other groups with lower agents number are negligible in time with respect to the biggest group.

A metric that could be interesting is the number of nodes explored by the different algorithms. However, it was not included, since it is difficult to compare that value for different algorithms. As an example, ICTS expands $O(b^{\Delta})$ nodes on its high-level search, where b is the branching factor and Δ is the difference between the optimal solution cost and the root cost. A* with OD instead expands $O(bnt)$, where n is the number of agents and t is the depth of the optimal solution. It is clear that these two pieces of information do not come from the same parameters, thus is not relevant to compare them.

In the experiments, different MAPF algorithms are tested, but other parameters such as SAPF algorithm, cost function, and heuristic remain constant. For the following, the single-agent path finding algorithm is A* with Manhattan heuristic and the cost function for the MAPF algorithm is SIC.

The test has been run on a computer with the following software and hardware characteristics:

- **Operating system:** Ubuntu 22.04.2 LTS with Linux kernel 5.19.0-32.
- **C++ toolchain:** GCC 12.1.0 and CMake 3.22.1.
- **IBM ILOG CPLEX:** Optimization Studio version 22.1.1.
- **CPU:** AMD Ryzen 7 1800X (8 cores, 3.6 GHz).
- **RAM:** 32 GB 3200 MT/s.

7.1 Random problems in Povo map

In these kinds of problems, the goals are chosen randomly among all the offices in the building. This choice has the consequence that usually robots have to travel long distances. The number of goals is a parameter, chosen between [1, 2, 5], used to generate the problems. It increases the difficulty of the problem by requiring the robots to visit more offices.

Another parameter is the number of robots present in a test, which is chosen between [1, 2, 5]. As the complexity is typically exponential in the number of robots, this parameter has a huge impact on performance.

For every type of problem composed of a number of goals and a number of robots, 20 random tests have been generated to have a mean of the performances. Each test was fed to each type of algorithm and its variants:

[A*, A*+ID, A*+OD, A*+ID+OD, ICTS, ICTS+ID, ICR+A*+OD, ICR+ICTS, ICR+CP]. Therefore the total number of executions is 1620.

7.1.1 Random problems with 1 robot

This test aims at assessing how the various algorithms deal with a single-agent path finding problem. In this experiments, a baseline for comparing the algorithms with more agents is reported. However, an important clarification is needed: when ID is selected the test is completed through the single-agent path finding algorithm selected at that time, in this case, single-agent A*. This results in a speed-up of many orders of magnitude since SAPF algorithms use the Manhattan heuristic instead of the SIC heuristic as MAPF solvers. This implies that MAPF algorithms have to run multiple times a SAPF and use the length of the found path as a guide. Obviously, since it is only one robot, a single SAPF solution is enough but MAPF algorithms do not have the capabilities to discern this case if ID is not activated. ICR behaves similarly to the ID options, therefore it computes the single-agent paths and then looks for conflicts, that in this case are not present. Hence, it is very fast.

All the algorithms are capable of solving single instances in less than 2s for a path 417 steps long. It can be noted that ICTS is a bit slower than A*.

7.1.2 Random problems with 2 robots

This test aims at assessing how the various algorithms deal with two agents. It is the first where MAPF solver capabilities are actually tested. As shown in Tabs. 7.4 to 7.6, all tests are completed except for one case in which A* has to solve a 5-goal instance. Also ICR with A*+OD as a sub-solver struggle to compute some paths where 5 goals are given. In Tabs. 7.4 and 7.5 it is clear the advantage brought by Independence Detection (ID). In fact, for these problem instances, there are many independent paths solved only with SAPF algorithms, and this results in lower average running times.

A* *Operator Decomposition* (OD) seems to bring no benefits, also slowing a bit down the search process. We think that this is due to having a large map with few robots that cross their path only a few times.

With only one goal per robot to satisfy (Tab. 7.4), the majority of the problems are solved in under 1 second, with an average time of 487ms for A* and 124ms for ICTS. Here, ID results are not reported since they evaluate only SAPF algorithm performances. Therefore, ICTS is four times faster than A* in solving these instances. This is due to the fact that the map is very large and there is plenty of space to move on. In this setting, ICTS is advantaged because it has to explore few ICT nodes since the number of conflicts is low. Instead, A* needs to expand one by one each state resulting in higher computational time.

Starting from Tab. 7.5 we can see that the number of conflicts, and therefore the average largest group, increases. Here, considering ID is more meaningful as it increases the number of completed tests under 10s to 20. Here, ICTS still has a great advantage over A* as before.

When the number of goals increases to 5, the number of independent paths lowers. Therefore, ID's power is reduced, as only 5 instances weigh the average. This brings the opportunity to inspect better MAPF algorithms performances. A* in its sub-cases is slower than ICTS by a factor of 2, reducing a bit the gap, as the environment became denser (of conflicts).

ICR with ICTS or CP as a sub-solver shows that solving conflicts locally gives a big advantage. The number of considered agents and the search space are reduced. In fact, in the case of CP, the time needed to solve the instances lowers to 1 second. For ICTS instead, it drops to 11 ms, making it the faster option available among the tested ones.

7.1.3 Random problems with 5 robots

This test aims at assessing how the various algorithms deal with five agents. Here, the probability of conflicts increases as more robots are introduced into the map. A* struggles in this setting as it completed 14 tasks with 1 goal, 5 with 2 goals, and none for 5 goals.

Also, A* without ID is not capable of dealing with kind of problems, but can when ID is enabled as the complexity is decreased. We can also see that the largest independent group size is low for the solved problems. In conclusion, A*+ID solved only the simplest problems where the agents are mostly independent. The difference between A* with or without OD can be seen in Tab. 7.8, where it reduces the computational time by 32%. In Tab. 7.7 this does not happen, since OD power is unleashed where more conflicts are present as in the previous table. Unfortunately, A* is not capable of dealing with problems with 5 robots and 5 goals, as the search space is too wide.

Algo \ T/O	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	20	20	20	96	96	96	-	-	-	0	0	0	92	92	92
A*+ID	20	20	20	1	1	1	1	1	1	0	0	0	92	92	92
A*+OD	20	20	20	96	96	96	-	-	-	0	0	0	92	92	92
A*+ID+OD	20	20	20	1	1	1	1	1	1	0	0	0	92	92	92
ICTS	20	20	20	124	124	124	-	-	-	0	0	0	92	92	92
ICTS+ID	20	20	20	1	1	1	1	1	1	0	0	0	92	92	92
ICR+A*+OD	20	20	20	1	1	1	-	-	-	0	0	0	92	92	92
ICR+ICTS	20	20	20	1	1	1	-	-	-	0	0	0	92	92	92
ICR+CP	20	20	20	1	1	1	-	-	-	0	0	0	92	92	92

Table 7.1: Robots: 1. Goals: 1.

Algo \ T/O	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	19	20	20	226	278	278	-	-	-	0	0	0	163	171	171
A*+ID	20	20	20	1	1	1	1	1	1	0	0	0	171	171	171
A*+OD	19	20	20	226	277	277	-	-	-	0	0	0	163	171	171
A*+ID+OD	20	20	20	1	1	1	1	1	1	0	0	0	171	171	171
ICTS	18	20	20	223	315	315	-	-	-	0	0	0	158	171	171
ICTS+ID	20	20	20	1	1	1	1	1	1	0	0	0	171	171	171
ICR+A*+OD	20	20	20	1	1	1	-	-	-	0	0	0	171	171	171
ICR+ICTS	20	20	20	1	1	1	-	-	-	0	0	0	171	171	171
ICR+CP	20	20	20	1	1	1	-	-	-	0	0	0	171	171	171

Table 7.2: Robots: 1. Goals: 2.

Algo \ T/O	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	8	20	20	406	1642	1642	-	-	-	0	0	0	266	417	417
A*+ID	20	20	20	3	3	3	1	1	1	0	0	0	417	417	417
A*+OD	8	20	20	404	1646	1646	-	-	-	0	0	0	266	417	417
A*+ID+OD	20	20	20	3	3	3	1	1	1	0	0	0	417	417	417
ICTS	7	20	20	380	1917	1917	-	-	-	0	0	0	251	417	417
ICTS+ID	20	20	20	3	3	3	1	1	1	0	0	0	417	417	417
ICR+A*+OD	20	20	20	3	3	3	-	-	-	0	0	0	417	417	417
ICR+ICTS	20	20	20	3	3	3	-	-	-	0	0	0	417	417	417
ICR+CP	20	20	20	3	3	3	-	-	-	0	0	0	417	417	417

Table 7.3: Robots: 1. Goals: 5.

Algo \ T/O	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	18	20	20	252	487	487	-	-	-	0.1	0.1	0.1	85	85	85
A*+ID	20	20	20	40	40	40	1.2	1.2	1.2	0.1	0.1	0.1	85	85	85
A*+OD	18	20	20	264	523	523	-	-	-	0.1	0.1	0.1	85	85	85
A*+ID+OD	20	20	20	40	40	40	1.2	1.2	1.2	0.1	0.1	0.1	85	85	85
ICTS	18	20	20	124	124	124	-	-	-	0.1	0.1	0.1	85	85	85
ICTS+ID	20	20	20	32	32	32	1.2	1.2	1.2	0.1	0.1	0.1	85	85	85
ICR+A*+OD	20	20	20	62	62	62	-	-	-	0.3	0.3	0.3	85	85	85
ICR+ICTS	20	20	20	18	18	18	-	-	-	0.3	0.3	0.3	85	85	85
ICR+CP	19	19	19	27	27	27	-	-	-	0.2	0.2	0.2	83	83	83

Table 7.4: Robots: 2. Goals: 1.

Algo \ T/O	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	7	18	20	430	2479	3566	-	-	-	0	11	9.9	146	182	183
A*+ID	16	20	20	28	815	815	1.1	1.3	1.3	5.9	9.9	9.9	175	183	183
A*+OD	7	17	20	438	2234	3877	-	-	-	0	7.1	6.0	146	177	181
A*+ID+OD	16	20	20	29	888	888	1.1	1.3	1.3	5.9	6.0	6.0	175	181	181
ICTS	14	20	20	420	726	726	-	-	-	6.9	4.9	4.9	160	181	181
ICTS+ID	19	20	20	133	220	220	1.3	1.3	1.3	5.1	4.9	4.9	178	181	181
ICR+A*+OD	15	20	20	30	1173	1173	-	-	-	0.2	0.7	0.7	177	181	181
ICR+ICTS	20	20	20	3	3	3	-	-	-	0.7	0.7	0.7	181	181	181
ICR+CP	20	20	20	133	133	133	-	-	-	0.7	0.7	0.7	181	181	181

Table 7.5: Robots: 2. Goals: 2.

Algo \ T/O	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	1	16	19	741	4348	7411	-	-	-	0	0.4	5.3	272	385	410
A*+ID	5	16	19	4	3600	6844	1.0	1.7	1.7	0	0.4	5.3	326	385	410
A*+OD	1	15	19	763	4202	7995	-	-	-	0	0.4	14	272	385	415
A*+ID+OD	5	14	19	4	2887	7471	1.0	1.6	1.7	0	0.2	14	326	381	414
ICTS	1	19	20	732	3561	4059	-	-	-	0	2.8	2.7	272	397	409
ICTS+ID	5	19	20	4	3114	3626	1.0	1.7	1.8	0	2.8	2.7	326	397	409
ICR+A*+OD	6	9	17	7	2594	17455	-	-	-	0.5	0.8	1.7	330	358	387
ICR+ICTS	20	20	20	11	11	11	-	-	-	1.7	1.7	1.7	410	410	410
ICR+CP	11	20	20	322	1119	1119	-	-	-	0.8	1.7	1.7	376	410	410

Table 7.6: Robots: 2. Goals: 5.

	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
T/O / Algo	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
A*+ID	13	13	14	75	75	1088	1.5	1.5	1.6	0.3	0.3	3.5	78	78	78
A*+OD	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
A*+ID+OD	13	13	14	72	72	1162	1.5	1.5	1.6	0.5	0.5	3.6	78	78	78
ICTS	18	19	19	500	574	574	-	-	-	0.4	0.6	0.6	80	80	80
ICTS+ID	17	19	19	171	280	280	1.9	2.0	2.0	0.7	0.6	0.6	80	80	80
ICR+A*+OD	13	18	19	73	499	1018	-	-	-	3.8	3.6	3.4	77	77	79
ICR+ICTS	19	20	20	19	112	112	-	-	-	3.6	3.5	3.5	79	80	80
ICR+CP	14	17	17	138	435	435	-	-	-	1.0	1.1	1.1	78	79	79

Table 7.7: Robots: 5. Goals: 1.

	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
T/O / Algo	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
A*+ID	2	5	5	7	681	681	1.0	2.2	2.2	0	1.0	1.0	174	151	151
A*+OD	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
A*+ID+OD	4	5	5	284	457	457	2.0	2.0	2.0	1.3	1.2	1.2	154	152	152
ICTS	6	19	19	787	2933	2933	-	-	-	0.7	0.9	0.9	129	155	155
ICTS+ID	9	19	20	554	1795	4075	2.7	3.5	3.5	0.7	0.9	1.3	138	155	155
ICR+A*+OD	2	10	19	10	2706	12829	-	-	-	0.0	3.9	5.2	174	139	158
ICR+ICTS	20	20	20	24	24	24	-	-	-	6.5	6.5	6.5	157	157	157
ICR+CP	7	16	16	463	1911	1911	-	-	-	2.6	6.1	6.1	154	160	160

Table 7.8: Robots: 5. Goals: 2.

	Test completed			Average run time [ms]			Average largest group			Average conflicts			Average length		
T/O / Algo	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s	1s	10s	60s
A*	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
A*+ID	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
A*+OD	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
A*+ID+OD	0	0	0	-	-	-	-	-	-	-	-	-	-	-	-
ICTS	0	1	6	-	4793	25147	-	-	-	-	2.0	1.3	-	280	345
ICTS+ID	0	1	5	-	5111	19932	-	5.0	4.8	-	2.0	1.0	-	280	355
ICR+A*+OD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ICR+ICTS	14	16	16	203	813	813	-	-	-	21	40	40	401	396	396
ICR+CP	0	0	5	-	-	16740	-	-	-	-	-	18	-	-	366

Table 7.9: Robots: 5. Goals: 5.

ICTS instead solves easily 19/20 problems with one goal with an average time of 574ms. In the case of two goals, the computational time goes up to a few seconds with ICTS which find a solution to 19 problems, while ICTS with ID solves 20 problems.

In Tab. 7.9 it appears clearly the difficulty of these problems, as ICTS solves only 6 tasks while A* none. This is highlighted by the number of independent groups (when ID is active) that is near 5 and the length of the path that is demanding. In this case, ID slows down the search, as ICTS with ID solved one problem less than ICTS. This can be caused by a combination of two factors. One is the time needed to solve the task with plain MAPF. The other derives from the functioning of ID. ID groups and solves conflicting agents together in an iterative way. Suppose there are 3 agents and at the beginning 2 of them collide while the other is independent. After solving the group of agents, we find that the third agents collide in this solution. So, ID groups together all the agents, and another MAPF problem needs to be solved. Thus, the search is slowed down since more than one search is performed. If the problem solved with plain MAPF use quite all the time at disposal and ID has to run more than one search it can happen that the algorithm is not capable of solving it in the time constraints.

ICR with ICTS as a sub-planner is the best solver also for this class of problems. It manages to find a solution to 16 of the hardest problems, while ICTS completes only 6 searches and A* none. With CP as sub-solver, the path is not always found, but the limitation here is given more from the memory requirements than from the timeout. In fact, CP uses a huge amount of memory when the space to explore is big. Note that sometimes ICR finds a path that is slightly longer than the one found by ICTS. Recall that ICR uses a local portion of the map to solve conflicts, thus the solution could be not optimal. This is the cause of the issue explained before.

7.2 Targeted problems in Povo map

In this section, specific handcrafted problem instances are tested to highlight the performances in relevant scenarios. A total of 9 targeted experiments are inspected. In the upcoming images, agents are defined as colored circles. Their starting point is defined by the letter 'S' and the ending point by the letter 'E'. In Figs. 7.4(b) and 7.4(c), agents start from a position in which other agents end. This happens also in other tests, but in this case, another approach had to be used for space constraint. Starting and ending positions are represented as a circle cut in half. The upper half is the starting position and the lower half represents the ending position.

Only single-goal problems are tested, as there is no substantial difference in solving with one or more goals. In these benchmarks, the length of the path is lower than the one of the tests done in Sec. 7.1. Also, the number of robots is increased. Therefore, the density is higher and conflicts are more likely.

7.2.1 Moving agents in block from corridor to corridor

Here, two tests have been executed. In Fig. 7.1, it can be seen the structure of the problems. The task is the same: moving a set of 8 robots from one location to another. The difference between the two instances is the distance traveled. In the first case (Fig. 7.1(a)), eight agents move from the rooms in the eastern outer corridor of Povo 1 to the rooms present in the south corridor of Povo 1. In the second case (Fig. 7.1(b)), the agents start from the same position and end in the rooms present in the south corridor of Povo 2.

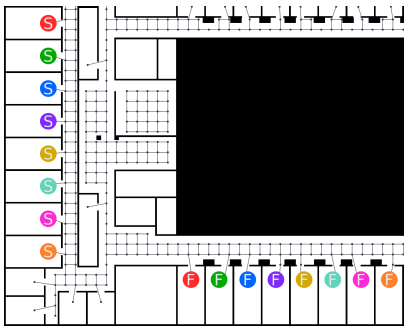
Even though agents move in group, it turns out that the paths are independent, so where ID is active a major improvement in the solving speed is present. In the first task of Tab. 7.10, the difference between A* and A* with OD can be appreciated. In fact, A* expand circa $b^n t$ nodes whether A* with OD bnt with a perfect heuristic. This difference becomes more relevant as the number of robots increases, as A* with OD provides exponential savings.

Since the paths are independent, ICR has the same effects as ID, resulting in a very low computational time.

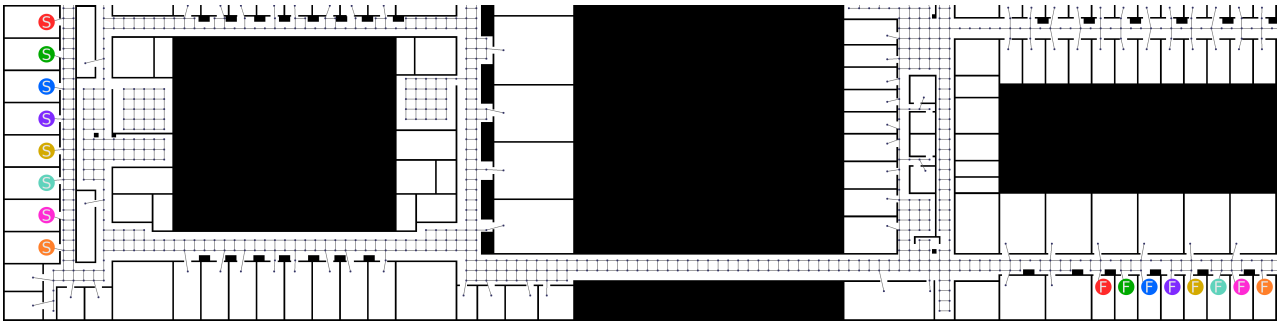
7.2.2 Swap agents in single lane corridor

In Fig. 7.2, four agents swap their position in the north corridor of Povo 2. Here, we find the first scenario in which A* performs better than ICTS. The cause is the increased density of conflict in the environment. In this setting, ICTS has to explore several layers of the *Increasing Cost Tree* (ICT). Instead, since the number of nodes in the graph is low, A* has only a few nodes to explore resulting in a speed-up in the search. Furthermore, OD grants some speed up in conflict resolution, as can be seen in Tab. 7.12.

Here, ICR and variants with ID perform worse than both plain A* and ICTS, due to the fact that all the agents conflict with each other. However, the reasons are different. ID needs to run four SAPFs and then, since all the agent conflicts, it runs one instance of the MAPF algorithm. Similar to ID, ICR solves four SAPF problems



(a) follow 1



(b) follow 2

Figure 7.1: Moving agents in block from corridor to corridor.

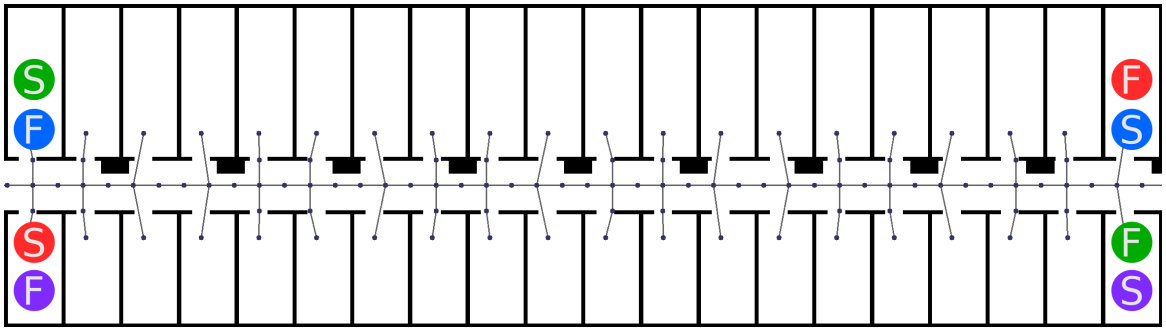


Figure 7.2: Swap agents in single lane corridor (swap 1).

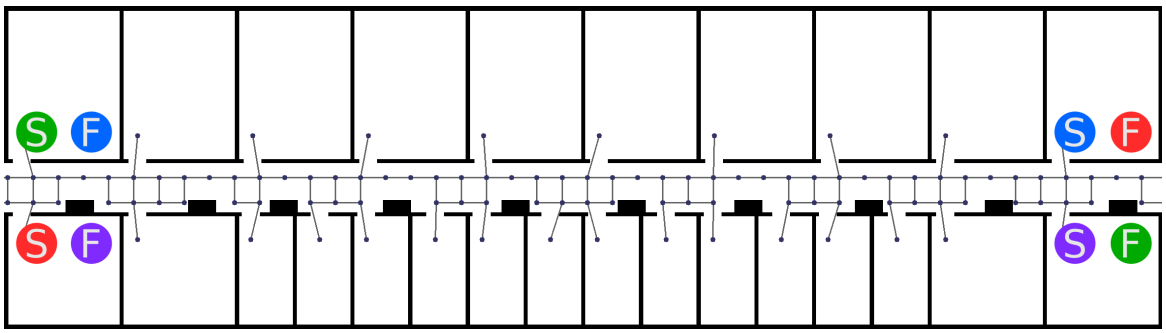
but has to run the MAPF three times to solve the conflicts. The performance depends on the dimensions of the local neighborhood selected by ICR.

7.2.3 Swap agents in quasi-single lane corridor

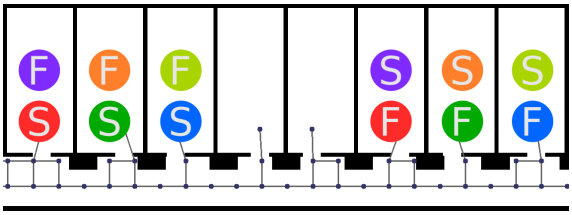
Here, two tests have been executed. In Fig. 7.3 it can be seen the structure of the problems.

In the first case (Fig. 7.3(a)), four agents swap their position in the north corridor of Povo 1. This example is similar to the previous task, but the density of graph points is higher resulting in a map that leads to fewer conflicts. The results in Tab. 7.11 confirm the statement expressed in Sec. 7.2.2. Indeed, here ICTS performs better than A* since the map is wider with fewer collisions. ICR found the solution faster than its sub-planners' complete algorithms, but in the case of CP, the solver ran out of memory.

In the second case (Fig. 7.3(b)), six agents swap their position in the eastern outer corridor of Povo 1. The graph composition is very similar to the previous test, but the number of robots increases, thus the number of conflicts. This scenario is particularly suitable for A* that manages to solve the benchmark in circa 4 seconds when OD is applied, as can be seen in Tab. 7.11. Instead, ICTS can not solve it as the search needs to deepen too much inside the ICT. ICR manages to solve it, but only with A*+OD as sub-planner, taking in any case a couple more seconds with respect to A* with OD.



(a) swap 2



(b) swap 4

Figure 7.3: Swap agents in quasi-single lane corridor.

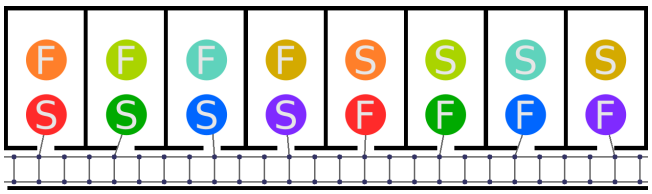
7.2.4 Swap agents in double lane corridor

Two locations for these tests have been chosen. The first location is the outer eastern corridor of Povo 1. Here in Fig. 7.4(a), eight robots swap their position in groups of four agents.

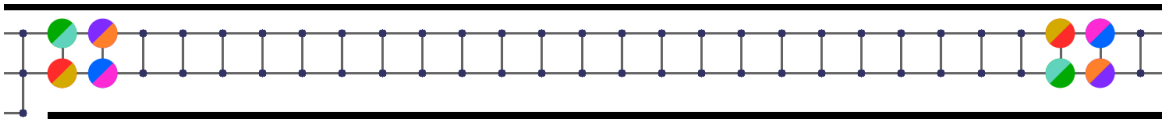
In this test, only ICTS and A* with OD can solve the problem. Strangely, A* with both ID and OD cannot find a solution. In our opinion, this is due to the fact that OD is affected by the order of the agents, especially when the heuristic is not admissible (due to the weight decay). *Independence Detection* (ID) merges the groups of agents in another way leading to a more difficult problem to solve.

The second location is the tunnel between the two buildings (Figs. 7.4(b) and 7.4(c)). Several problems with an increasing number of robots have been inspected. The numbers of tested robots are [6, 8, 10]. The setting is the same: robots are divided into two groups each on one side of the tunnel and they need to swap their positions. Only tests with 6 and 8 robots are completed, the one with 10 agents can not be managed by our planners. The results are presented in Tab. 7.13.

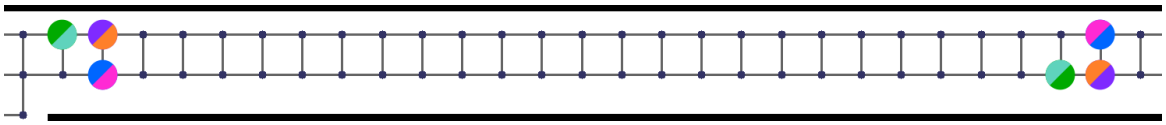
In the 8 agents test, only ICTS and ICR with ICTS are able to find a solution. This test proved to be difficult since it took 21 seconds to solve. In the 6 agents test, also A* with OD and ICR with A*+OD as sub-solver can



(a) swap 5, the image is rotated of 90° clockwise



(b) swap 10



(c) swap 11

Figure 7.4: Swap agents in double lane corridor.

find a solution. As before something strange happens when ID is enabled alongside OD, in this case reducing the computational time of a factor 16. As explained previously, this is caused by OD being sensible by the order of the agents that is modified by ID during the merge of conflicting groups.

7.3 Overall results

In conclusion, an overall comparison of the algorithms is reported to summarize the pros and cons of each technique.

A*

A* proves powerful when the space of states is dense with obstacles, thus in narrow environments with a large number of robots. OD helps to reduce search complexity. Our implementation of A* is relatively memory-hungry since usually after some minutes it saturates the RAM leading to crashes. Anyway, in the inspected tests where the time out was set to 1 minute, this problem never occurred.

ICTS

With its ability to start from the single-agent paths and then refine the solution handling the collisions, ICTS is suitable for wide environments with a low probability of conflicts such as the department's first floor. ICTS is able to run for hours without saturating the memory. This could be an advantage since more memory-heavy optimizations in the code could be pursued.

ID

On top of both algorithms, ID helps to solve efficiently independent sub-problems, saving time and resources. The advantages brought by ID exceed the disadvantages caused when more replanning is needed in case of complex problems.

CP

Before including CP as a sub-planner of ICR, some brief tests were executed. CP could not manage to solve any of the instances presented, crashing soon after starting because it fills up the memory. As concluded by Saccon, CP does not manage big graphs as in this case [12], thus the idea of including it in ICR.

Another drawback of CP is that it requires an optimization software to work. In our case, IBM ILOG CPLEX was used which requires a commercial license or a student status to be used on "large" problems.

ICR

ICR delegate conflict resolution to its MAPF planner, inheriting its strength and weaknesses. Three MAPF algorithms have been tested in this setting: A* with OD, ICTS, and CP. Variants with ID were excluded since they would be redundant with ICR. Also, A* was discarded as A* with OD is more powerful at solving conflicts.

With ICR, CP proved to be at least able to solve some tests, even though in most of the cases it can not compete with the other two planners. A* with OD is able to solve a big portion of the tests, missing the most complex ones, even though it excels in cluttered environments where others fail. The best configuration is when ICR is combined with ICTS.

7.3.1 Final consideration

Obviously, it does not exist an algorithm that is better than any other one. Each algorithm is good at solving a specific type of problem. In our scenario, where few robots have to pick up and deliver packages in a large building with wide corridors and the tasks required by the system do not number in the hundreds in an hour, we believe that ICR with ICTS is the best algorithm between the inspected ones to handle the system. The motivations are that in this scenario ICTS is the best tested MAPF planner, and by combining it with ICR that solves conflicts only locally, this results in an increment in the performances. Furthermore, ICR does not hinder too much ICTS when complex problems are presented, as shown in the targeted tests.

Test	Algorithm	Run time [ms]	Largest group	Conflicts	Length
Follow 1	A*	29274	-	0	39
	A*+ID	2	1	0	39
	A*+OD	122	-	2	40
	A*+ID+OD	2	1	0	39
	ICTS	74	-	0	39
	ICTS+ID	2	1	0	39
	ICR+A*+OD	2	-	0	39
	ICR+ICTS	2	-	0	39
	ICR+CP	2	-	0	39
Follow 2	A*	-	-	-	-
	A*+ID	5	1	0	130
	A*+OD	-	-	-	-
	A*+ID+OD	4	1	0	130
	ICTS	569	-	0	130
	ICTS+ID	4	1	0	130
	ICR+A*+OD	8	-	0	130
	ICR+ICTS	8	-	0	130
	ICR+CP	9	-	0	130

Table 7.10: Moving agents in block from corridor to corridor.

Test	Algorithm	Run time [ms]	Largest group	Conflicts	Length
Swap 2	A*	203	-	1	45
	A*+ID	260	4	1	45
	A*+OD	91	-	7	46
	A*+ID+OD	104	4	7	46
	ICTS	63	-	0	45
	ICTS+ID	64	4	0	45
	ICR+A*+OD	90	-	8	47
	ICR+ICTS	42	-	4	46
	ICR+CP	-	-	-	-
Swap 4	A*	26125	-	7	21
	A*+ID	27821	6	7	21
	A*+OD	3324	-	17	22
	A*+ID+OD	4806	6	17	22
	ICTS	-	-	-	-
	ICTS+ID	-	-	-	-
	ICR+A*+OD	5163	-	9	22
	ICR+ICTS	-	-	-	-
	ICR+CP	-	-	-	-

Table 7.11: Swap agents in quasi-single lane corridor.

Test	Algorithm	Run time [ms]	Largest group	Conflicts	Length
Swap 1	A*	89	-	6	48
	A*+ID	117	4	6	48
	A*+OD	55	-	6	48
	A*+ID+OD	67	4	6	48
	ICTS	764	-	6	48
	ICTS+ID	799	4	6	48
	ICR+A*+OD	160	-	6	48
	ICR+ICTS	812	-	6	48
	ICR+CP	-	-	-	-

Table 7.12: Swap agents in single lane corridor.

Test	Algorithm	Run time [ms]	Largest group	Conflicts	Length
Swap 5	A*	-	-	-	-
	A*+ID	-	-	-	-
	A*+OD	7301	-	7	17
	A*+ID+OD	-	-	-	-
	ICTS	2036	-	0	15
	ICTS+ID	1905	8	0	15
	ICR+A*+OD	-	-	-	-
	ICR+ICTS	-	-	-	-
	ICR+CP	-	-	-	-
Swap 9	A*	-	-	-	-
	A*+ID	-	-	-	-
	A*+OD	-	-	-	-
	A*+ID+OD	-	-	-	-
	ICTS	-	-	-	-
	ICTS+ID	-	-	-	-
	ICR+A*+OD	-	-	-	-
	ICR+ICTS	-	-	-	-
	ICR+CP	-	-	-	-
Swap 10	A*	-	-	-	-
	A*+ID	-	-	-	-
	A*+OD	-	-	-	-
	A*+ID+OD	-	-	-	-
	ICTS	21326	-	2	26
	ICTS+ID	28726	8	4	26
	ICR+A*+OD	-	-	-	-
	ICR+ICTS	21202	-	2	26
	ICR+CP	-	-	-	-
Swap 11	A*	-	-	-	-
	A*+ID	-	-	-	-
	A*+OD	9065	-	8	28
	A*+ID+OD	567	6	4	27
	ICTS	21	-	1	26
	ICTS+ID	22	6	1	26
	ICR+A*+OD	5172	-	10	28
	ICR+ICTS	11	-	1	26
	ICR+CP	-	-	-	-

Table 7.13: Swap agents in double lane corridor.

We think that when the system will be in full use, situations such as the handcrafted ones will be unlikely. Also, we estimate the average path length to be around 100 steps. Therefore, a planner like ICR can solve user-inserted tasks within a short time, leading to a good user experience.

8 Conclusions and Future Work

8.1 Conclusions

Nowadays, the intralogistics sector is moving from a structure based on human labor to one where AGVs are employed to solve its tasks. *Autonomous Ground Vehicles* (AGVs) need to optimize the distance traveled, while avoiding crashes with other elements present in the environment, them being either other robots or dynamic obstacles, managing battery power, and returning to their base at the end of the tasks. The problem of planning the paths of multiple robots that avoid crashing between each other is referred to in the literature as *Multi-Agent Path Finding* (MAPF) problem. When agents need to pick up and deliver packages, the problem is referred to as *Multi-Agent Pickup and Delivery* (MAPD).

Firstly, a review of the literature is carried out highlighting the most interesting contribution in this field, discovering the differences between centralized and distributed algorithms, and between combinatorial and reinforced-base ones. Since the orchestration of robots, both in academia and industry, is achieved by the means of *Robot Operating System* (ROS), a presentation of the new version (ROS 2) is given.

In this thesis, we implemented an intralogistics system in all its stack, as a team effort between Lorenzini Giovanni and Planchenstainer Diego. The system is employed to carry packages between offices in the first floor of the building of Povo, headquarters of the Department of Information Engineering and Computer Science (DISI). Different components have been developed to manage the MAPD tasks. Lorenzini focused on the implementation of the Backend and the Database, while Planchenstainer concentrated on the Web GUI and the ROS infrastructure. The last part, the MAPF planner, is based on the work of Enrico Saccon, where we added new planners. We arbitrarily choose to implement two centralized combinatorial planners: multi-agent A* for Lorenzini and ICTS for Planchenstainer.

Then, a review of the performances has been carried out. Four algorithms with their variation are inspected: A*, ICTS, CP, and ICR. A* has four variations which add to the plain A* planner OD and/or ID. The best combination of A* is with OD and ID, which is fast at solving problems where many robots are present in a small area. These methods are overcome when the environment enlarges and the density of robots lowers. In this situation, ICTS works better. CP has some disadvantages regarding memory consumption and the necessity of a solver software. ICR is the best algorithm, but it requires a sub-solver chosen from the above algorithms from which it inherits its strength and weakness, typically improving them.

8.2 Future work

Even though the system is capable of solving the tasks, some improvements could be made.

One is to include charging stations in the map and handle battery capacity. A trivial way could be to set two thresholds: a working one and a charging one. This approach could lead to many robots charging in rush hours, or many robots idle in the working area when few tasks need to be executed. A more complex approach could be to model the problem as a Markov-Decision Process and solve it with Deep Reinforcement Learning, as in [79]. Also, the system implemented stores in the database the history of all the tasks, ordered by time. This data can be exploited to predict the probability of an incoming task, integrating this information into the battery management algorithm.

Then, other algorithms can be implemented and tested. We think that distributed algorithms could work faster in this environment, where the density of robots is not high. Considerations about different edges' lengths is an option that allows using more complex maps. In fact, A* with minor changes can consider edges with different weights, while ICTS should be adapted using one of the more complex variants. This enables the possibility of optimizing the graph which consists in removing useless nodes between certain edges, leading to a simplified graph.

As an addition, since we are not fully satisfied with the code that handles the communication with ROS, and the code that stops the robots when two subsequent nodes are equal, nicer ways to solve the problem could be pursued in the future. In particular for the second case the condition could be modeled with behavior trees,

making it more robust and scalable. A further addition could be to wait for a signal that tells that all robots are in position, instead of waiting a predefined constant time.

Lastly, we could evaluate whether it is possible to optimize the code, making algorithms faster.

Bibliography

[1] Biryukov A., Dinu D., and Khovratovich D. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy*, pages 292–302, 2016.

[2] Ferwerda A. Extending the multi-label a* algorithm for multi-agent pathfinding with multiple waypoints, 2020.

[3] Babu B. ros2bridge. https://pypi.org/project/ros2bridge/.

[4] Jones M. B., Bradley J., and Sakimura N. JSON Web Token (JWT). *RFC*, 7519:1–30, 2015.

[5] Stout B. Smart moves: Intelligent pathfinding. *Game Developer Magazine*, April 1996.

[6] Silver D. Cooperative pathfinding. In *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'05, page 117–122. AAAI Press, 2005.

[7] DreamHost. Web server performance comparison. https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison.

[8] Boyarski E., Felner A., Stern R., Sharon G., Tolpin D., Betzalel O., and Shimony E. ICBS: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 740–746. AAAI Press, 2015.

[9] Hart P. E., Nilsson N. J., and Raphael B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[10] Saccon E. Increasing conflict resolution (ICR). https://bitbucket.org/chaff800/maof/src/dev/include/MAPF/ICR/.

[11] Saccon E. Multi-agent open framework. https://bitbucket.org/chaff800/maof/.

[12] Saccon E. Comparison of multi-agent path finding algorithms in an industrial scenario. Master's thesis, University of Trento, 2022.

[13] Codd E. F. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970.

[14] Codd E. F. Further normalization of the data base relational model. *IBM Research Report*, 1971.

[15] Codd E. F. Recent investigations into relational data base systems. *IBM Research Report*, 1974.

[16] Grenouilleau F., Hoeve W. J., and Hooker J. N. A multi-label a* algorithm for multi-agent pathfinding. *Proceedings of the International Conference on Automated Planning and Scheduling*, 29(1):181–185, 2021.

[17] Tully Foote. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, Open-Source Software workshop, pages 1–6, April 2013.

[18] Pardo-Castellote G. OMG data distribution service: architectural overview. In *IEEE Military Communications Conference, 2003. MILCOM 2003.*, volume 1, pages 242–247 Vol.1, 2003.

[19] Sartoretti G., Kerr J., Shi Y., Wagner G., Kumar T. K. S., Koenig S., and Choset H. PRIMAL: Pathfinding via reinforcement and imitation multi-agent learning. *IEEE Robotics and Automation Letters*, 4(3):2378–2385, 2019.

[20] Sartoretti G., Shi Y., Paivine W., Travers M., and Choset H. Distributed learning for the decentralized control of articulated mobile robots. In *Proceedings of (ICRA) International Conference on Robotics and Automation*, pages 3789 – 3794, May 2018.

[21] Sartoretti G., Wu Y., Paivine W., Kumar T. K. S., Koenig S., and Choset H. Distributed reinforcement learning for multi-robot decentralized collective construction. *DARS*, pages 35–49, 2018.

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

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

[24] Wagner G. and Choset H. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219, 11 2014.

[25] Google. What is Angular? https://angular.io/guide/what-is-angular.

[26] PostgreSQL Global Development Group. PostgreSQL: About. https://www.postgresql.org/about/.

[27] Abiyev R. H., Günsel I., Akkaya N., Aytac E., Çağman A., and Abizada S. Robot soccer control using behaviour trees and fuzzy logic. *Procedia Computer Science*, 102:477–484, 2016. 12th International Conference on Application of Fuzzy Systems and Soft Computing, ICAFS 2016, 29-30 August 2016, Vienna, Austria.

[28] Ma H., Harabor D., Stuckey P. J., Li J., and Koenig S. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, AAAI'19/IAAI'19/EAAI'19. AAAI Press, 2019.

[29] Ma H., Li J., Satish Kumar T. K., and Koenig S. Lifelong multi-agent path finding for online pickup and delivery tasks. *CoRR*, abs/1705.10868, 2017.

[30] Ma H., Li J., Kumar T. K. S., and Koenig S. Lifelong multi-agent path finding for online pickup and delivery tasks. *CoRR*, abs/1705.10868, 2017.

[31] Lutkebohle I., Gamarra B. O., Goenaga I. M., Losa J. M., and Vilches M. V. micro-ROS: ROS 2 on microcontrollers. In *ROSCon.*, Open Robotics, October 2019.

[32] Pohl I. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3):193–204, 1970.

[33] Pohl I. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 12–17, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[34] IBM Corportation. CPLEX® optimizers. https://www.ibm.com/docs/en/icos/12.8.0.0?topic=cplex.

[35] Barraquand J. and Latombe J. Robot motion planning: A distributed representation approach. *International Journal of Robotic Research - IJRR*, 10:628–649, 12 1991.

[36] Coffin J. Create all possible combinations of multiple vectors. https://stackoverflow.com/a/48271759.

[37] De Smith M. J., Goodchild M. F., and Longley P. *Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools*. Troubadour Publishing, 2007.

[38] Hernandez M. J. *Database Design for Mere Mortals: A Hands-on Guide to Relational Database Design*. Addison-Wesley Professional, third edition edition, 2013.

[39] Yu J. and LaValle S. Structure and intractability of optimal multi-robot path planning on graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1):1443–1449, Jun. 2013.

[40] Cai K., Wang C., Cheng J., de Silva C. W., and Meng M. Q.-H. Mobile robot path planning in dynamic environments: A survey. *CoRR*, abs/2006.14195, 2020.

[41] Karur K., Sharma N., Dharmatti C., and Siegel J. E. A survey of path planning algorithms for mobile robots. *Vehicles*, 3(3):448–468, 2021.

[42] Simonyan K. and Zisserman A. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1906.08291, 2014.

[43] Pinca L. ws: a node.js websocket library. https://github.com/websockets/ws.

[44] Damani M., Luo Z., Wenzel E., and Sartoretti G. PRIMAL_2: Pathfinding via reinforcement and imitation multi-agent learning - lifelong. *IEEE Robotics and Automation Letters*, 6(2):2666–2673, 2021.

[45] Erdmann M. and Lozano-Perez T. On multiple moving objects. In *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, volume 3, pages 1419–1424, 1986.

[46] Haralick R. M., Sternberg S. R., and Zhuang X. Image analysis using mathematical morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(4):532–550, 1987.

[47] Lavalle S. M. *Planning Algorithms*. Cambridge University Press, 2006.

[48] Leitner-Ankerl M. ankerl::unordered_dense::{map, set}. https://martin.ankerl.com/2022/08/27/hashmap-bench-01/.

[49] Leitner-Ankerl M. Comprehensive c++ hashmap benchmarks 2022. https://martin.ankerl.com/2022/08/27/hashmap-bench-01/.

[50] Quigley M., Conley K., Gerkey B., Faust J., Foote T., Leibs J., Wheeler R., and Ng A. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, volume 3, 01 2009.

[51] Ryan M. Constraint-based multi-robot path planning. In *2010 IEEE International Conference on Robotics and Automation, ICRA 2010*, pages 922–928, United States, 2010. Institute of Electrical and Electronics Engineers (IEEE). 2010 IEEE International Conference on Robotics and Automation, ICRA 2010 ; Conference date: 03-05-2010 Through 07-05-2010.

[52] Sujian Y. M., Shashidhara H. R., and Rohini N. Survey paper: Framework of REST APIs. *International Research Journal of Engineering and Technology*, 2020.

[53] Yu J.and LaValle S. M. Planning optimal paths for multi-agent systems on graphs. *CoRR*, abs/1204.3830, 2012.

[54] MySQLtutorial.org. MySQL UUID Smackdown: UUID vs. INT for primary key. https://www.mysqltutorial.org/mysql-uuid/.

[55] Chhetri N. A comparative analysis of node.js (server-side javascript). *Culminating Projects in Computer Science and Information Technology*, 2016.

[56] Koenig N. and Howard A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[57] Surynek P., Felner A., Stern R., and Boyarsk E. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence*, ECAI'16, page 810–818, NLD, 2016. IOS Press.

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

[59] Fagin R. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst.*, 2(3):262–278, sep 1977.

[60] Sanders R. The future of opengl. https://community.khronos.org/t/the-future-of-opengl/106317.

[61] Stern R. *Multi-Agent Path Finding – An Overview*, page 96–115. Springer-Verlag, Berlin, Heidelberg, 2022.

[62] Stern R., Sturtevant N. R., Felner A., Koenig S., Ma H., Walker T. T., Li J., Atzmon D., Coher L., Satish Kumar T. K., Boyarski E., and Barták R. Multi-agent pathfinding: Definitions, variants, and benchmarks. *CoRR*, abs/1906.08291, 2019.

[63] reichelt elektronik. Tb3 burger robotis turtlebot3 burger. https://www.reichelt.com/it/it/robotis-turtlebot3-burger-tb3-burger-p264201.html.

[64] Auryn Robotics. Behaviortree.cpp 4.0. https://www.behaviortree.dev/.

[65] ROBOTIS. Ros packages for turtlebot3. https://github.com/ROBOTIS-GIT/turtlebot3.

[66] Howlett S. How to create a good hash_combine with 64 bit output. https://stackoverflow.com/a/8980550.

[67] Macenski S., Martín F., White R., and Clavero J. G. The marathon 2: A navigation system. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2718–2725, 2020.

[68] Macenski S., Foote T., Gerkey B., Lalancette C., and Woodall W. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[69] Ravoof S. How to set up a reverse proxy (step-by-steps for nginx and apache). https://kinsta.com/blog/reverse-proxy/.

[70] Raymond E. S. Basics of the unix philosophy. http://www.catb.org/ esr/writings/taoup/html/ch01s06.html.

[71] Russell S. and Norvig P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition edition, 2003.

[72] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, pages 417–431, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[73] LaValle S.M. and Hutchinson S.A. Optimal motion planning for multiple robots having independent goals. *IEEE Transactions on Robotics and Automation*, 14(6):912–925, 1998.

[74] Standley T. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, AAAI'10, page 173–178. AAAI Press, 2010.

[75] Walker T. T., Sturtevant N. R., and Felner A. Extended increasing cost tree search for non-unit cost domains. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 534–540. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[76] Ars Technica. Microsoft TypeScript: the JavaScript we need, or a solution looking for a problem? https://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/.

[77] TypeORM. TypeORM - Amazing ORM for TypeScript and JavaScript (ES7, ES6, ES5). Supports MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, WebSQL databases. Works in NodeJS, Browser, Ionic, Cordova and Electron platforms. https://typeorm.io/.

[78] Dijkstra E. W. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, dec 1959.

[79] Mu Y., Li Y., Lin K., Deng K., and Liu Q. Battery management for warehouse robots via average-reward reinforcement learning. In *2022 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 253–258, 2022.

[80] Čáp M., Vokřínek J., and Kleine A. Complete decentralized method for on-line multi-robot trajectory planning in well-formed infrastructures. *Proceedings of the International Conference on Automated Planning and Scheduling*, 25(1):324–332, Apr. 2015.