Final Master's Thesis

Master's Degree in Automatic Control and Robotics (MUAR)

Multi-robot Task Allocation System to Improve Assistance in Domestic Scenarios

THESIS

June 19, 2019

Autor: Louise Bonnefoy

Director: Joan Aranda López

Call: June 2019

ETSEIB

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

UPC

Abstract

The AURORA project aims at developing new strategies to take assistive robotics a step further. In order to provide extended services to potential users, engaging a team of simpler robots is often preferable to using a unique, super-capable robot. In such multi-robot systems, the coordination for task execution is one of the major challenges to overcome. The present thesis proposes a solution to the specific issue of multi-robot task allocation within an heterogeneous team of robots, with additional inter-task precedence constraints.

The main elements of the state of the art that support this project are reported, including useful taxonomies and existing methods. From the analyzed solutions, the one that better fits with the constraints of the project is an iterated auction-based algorithm able to manage precedence constraints, which has been modified to handle heterogeneity and partial scheduling.

The selected solution has been designed and implemented with the particular purpose of being applied to the robotized kitchen setup of the AURORA project; it is however flexible and scalable and can therefore be applied to other use-cases, for instance vehicle-routing problems. Several evaluation scenarios have been tested, that demonstrate the good functioning and characteristics of the system, as well as the possibility to integrate humans into the task assignation process.

Contents

Glossary and Acronyms

API	Application Programming Interface.
APSP	All-Pair-Shortest-Path.
DCOP	Distributed Constraint-Based.
DOF	Degree Of Freedom.
GUI	Graphical User Interface.
IK	Inverse Kinematics.
KDL	Kinematics and Dynamics Library.
MILP	Mixed Integer Linear Programming.
MRS	Multi-Robot System.
MRTA	Multi-Robot Task Allocation.
PRM	Probabilistic Roadmap Methods (motion planner).
ROS	Robot Operating System.
RRT	Rapidly-exploring Random Trees (motion planner).
STL	Standard Template Library. An important C++ library that provides a set of generic algorithms and useful data structures.
STN	Simple Temporal Network.
STP	Simple Temporal Problem.
TCSP	Temporal Constraint Satisfaction Problem.
URDF	Unified Robot Description Format, an markup language format used to represent a robot model.
VRP	Vehicle-Routing Problem.
YAML	YAML Ain't Markup Language. A human-readable data serialization language, which is commonly used for configuration files.

ETSEIB

Chapter 1

Introduction

1.1 Motivation

Robot assistants are undeniably gaining ground. For instance, the development of Industry 4.0 comes with a technological shift: the classical industrial robots safely guarded behind fences are now replaced by *cobots* which work in close collaboration with human operators. This new paradigm does not only affect the industry; on the contrary, it plays a major role in the redefinition of many activities from the service sector. Besides the need to improve the accuracy and efficiency of the industrial tasks, helping customers and visitors, automating household chores, providing support to an aging population and other is boosting the development of intelligent and human-friendly assisting robots. As an illustration of the recent technological evolutions, one can mention the robots Tiago (PAL Robotics), Pepper (SoftBank Robotics), PR2 (Willow Garage) and HRP4 (Kawada), among many others.

In particular, robots are gaining importance in the healthcare sector. This area indeed tends to suffer from workforce and resource shortage while facing an ever-increasing demand. Robots can, therefore, be used profitably to assist both caregivers and patients.

In the latter case, one can distinguish between various levels of support. Depending on the deterioration of the user's cognitive and/or motor abilities, the robotic system can be used for stimulation (to maintain cognitive or physical activity), rehabilitation (to progressively enhance hampered skills) and assistance (to compensate irreversible losses). Each level requires the study and development of safe and adaptive systems that can act or be commanded according to the user's needs and wishes.

In parallel to the progress made in such assisting systems, much effort has been put in the last decades to develop multi-robot systems (MRS). Teams of mobile robots can for instance be used advantageously to achieve exploration or multi-delivery missions. Another key motivation lies in the fact that working with a team of simpler robots is often preferable to using a unique, super-capable robot: each item is often easier to develop, and provides more flexibility to solve complex problems.

Obviously, the two topics that have been mentioned in this introduction intertwine in some applications. For instance, one could mention the work by [2], in which a team of social robots assists caregivers in a retirement home; the robots autonomously organize in order to visit, interact and play with each resident. As in this project, the work carried out for this Final Master's Thesis stands at the crossroad of these two topics: assisting applications and multi-robot systems.

1.2 Background

This thesis was written in partial fulfillment of the **Master's Degree in Automatic Control and Robotics** of the Barcelona School of Industrial Engineering (Escola Tècnica Superior d'Enginyeria Industrial de Barcelona, ETSEIB) which is part of the Polytechnic University of Catalonia (Universitat Politècnica de Catalunya, UPC).

It was carried out within the laboratory of the Intelligent Robotics and Systems Research Group[1] (GRINS) of the UPC. The group gathers researchers from the former research group in Robotics and Vision, from the Biomedical Engineering Research Center (CREB) as well as the Bioengineering Institute of Catalonia[2] (IBEC).

Its main topics of interest are the study, evaluation and development of robotic systems with perception capabilities, in order to obtain flexible and intelligent behaviours. More specifically, the research is centered around three axes:

- design and development of vision and image-processing systems;
- cooperation in multi-agents systems;
- robotic systems adapted to specific tasks or surroundings (underwater robotics, industrial and services robotics or more recently medical robotics).

Recent projects[3] include for instance:

- the development of a low-cost rehabilitation system to assist children affected by severe motor disorders;
- the development of a smart walker (for rehabilitation as well);
- an important project in collaboration with two hospitals and several other academic institutions to improve fetal surgery procedures.

In particular, the work presented in this report is situated within the frame of the **AURORA project**, which aims at developing new distributed control strategies and human-robot cooperation in assistive setups [4]. This national project is financed by the Spanish Ministry of Economy and Business and led by the GRINS with the collaboration of the Bioengineering Institute of the Miguel Hernández University.

The AURORA project is centered around the study of health care systems endowed with distributed intelligence elements, in order to enhance their global intelligence, efficiency and interpretation ability, therefore offering higher levels of cooperation and assistance. To do so, the goal is to achieve greater parallelism to improve the efficiency and reliability and to be able to establish optimal control strategies.

Attaining theses objectives calls for significant advancement in the interpretative and control schemes of the system, with the development of new algorithms and methods to estimate the intention and state of the user and of new interfaces to increase adaptive and proactive abilities of the setup, always keeping in mind the high safety requirements.

Since it is intertwined with the AURORA project, the work achieved in this thesis focuses on healthcare and assistive setups, and more specifically on **domestic environments**. Part of the laboratory dedicated to this project therefore consists of a replica of a kitchen working surface, equipped with hobs and a sink, placed against a set of three cupboards. The surface is high enough so that a disabled person using a wheelchair may move around. This can be seen in figure 1.1.

[1]https://grins.upc.edu/en
[2]https://www.ibecbarcelona.eu/
[3]https://grins.upc.edu/en/projects/research-projects

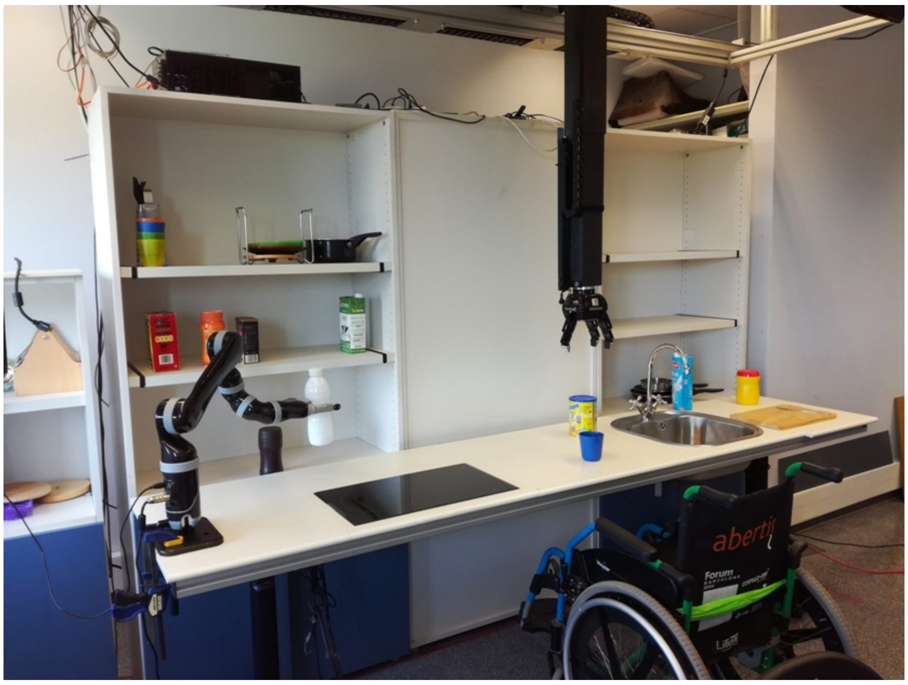


Figure 1.1: Kitchen setup

The kitchen is furnished with several household items such as pots, pans, glasses and cartons that can typically be found in such domestic environments. The setup is equipped with different robots, which architecture and characteristics will be presented in chapter 5, and several Kinect cameras to monitor the scene.

The remaining of this report is organized as follows:

- Chapter 2 presents the objectives and the work plan.
- Chapter 3 is dedicated to the literature review about the specific multi-robot task allocation problem, preceded by a brief overview of multi-robot systems.
- Chapter 4 analyzes the requirements and presents the final solution design.
- Chapter 5 describes the implementation of the proposed solution.
- Chapter 6 gathers the description and results from the evaluation tests.
- Chapter 7 examines the project cost and environment impact.
- Chapter 8 concludes the thesis and gives suggestions about future work.

Chapter 2

Objectives

Using the tools and resources associated to the AURORA project, the main goals of the thesis are the **design, implementation and evaluation of a Multi-Robot Task Allocation (MRTA) system adapted to an heterogeneous team of robotic manipulators.**

In the setup considered for this project, the team of robots is made up of several robots that should coordinate to help a disabled user performing daily tasks in a kitchen. The nature of the tasks has a considerable impact on the design of the solution. In our case, several task levels can be considered. The first level is constituted by the *high-level tasks*, such as "making a coffee" or "pouring a glass of water". These types of actions are the ones that will most likely be triggered by the user through any relevant user-machine interface. One understand that these tasks or *goals* are actually compound or even complex tasks which requires the execution of various sub-tasks. For instance, "pouring a glass of water" will require (1) to fetch an empty glass, (2) to grasp a bottle of water, (3) to pour the water from the bottle into the glass. These tasks of lower level can themselves be decomposed into sub-subtasks such as "picking the glass from its location", "placing the glass at a given location"...

In order to narrow the problem, the focus was set on the task allocation only. This thesis does not cover the task decomposition problem. Because of its importance, it will be explained in the report when presenting the ins and outs of the task allocation issue; however, we shall assume that the input to the MRTA algorithm is given and does not need any further pre-processing, i.e. the **task decomposition is given and fixed**. Because working with high-level tasks is not meaningful nor relevant to perform adequate task allocation, the tasks we will work with are the tasks from the following layers. In terms of task classification, they are **simple or elemental tasks** (see 3.2.1.3). For the decomposition to be correct however, one need to consider some **ordering constraints**, in order not to pour the water before the glass has been placed on the table!

It results that the constraints we will be working with are of the following types:

- **capability constraints**: because the robots are **heterogeneous**, certain tasks may be performed by some of the robots and not others. For instance, in the AURORA setup, the MICO robot offers a greater variety of grasp orientation and can reach the shelves, while the CAPDI is the only one that can move an object from one side of the table to the other.
- **capacity constraints**: each robot can perform a limited number of tasks at the same time, merely one.
- **precedence constraints** on the tasks.

The final solution to be implemented should therefore be able to handle these constraints. Ideally, it should also have the following characteristics:

- **flexibility**: the implementation should be flexible so that the system may be used on various setups with no or only minor alterations.
- **scalability**: the system should be adaptable to a various number of robots and not be limited to the current AURORA setup.
- **robustness**: the system should be fault-tolerant, i.e. in the case of a failure, either be able to recover and manage the error, or properly warn the user.
- **functionality**: it should be possible to run the system online, with consumer-level equipment, so that it can be used in practice.

Finally, the development should take into account that the MRTA solution has to be integrated in a wider system with assistive purpose, which implies the possibility to have a **human in the loop**.

Work plan

The project was developed over a five-month period, following the steps described below:

1. **Study of the state of the art**: as an initial objective, it is required to review the literature about MRS and MRTA in order to collect relevant data and research results. The information obtained from the bibliographic research is indeed one of the cornerstones of the analysis of the problem and potential solutions.
2. **Development/Enhancement of the multi-robot system**: from the already-existing elements of code, development of the code and modules necessary to control the different manipulators.
3. **Design of the MRTA solution**: from the study of the state of the art and the analysis of the problem specificities and constraints, selection of the relevant paradigms and techniques to use.
4. **Development of the MRTA system**: actual development and implementation of the algorithms for the task allocation process.
5. **Evaluation**: definition and set up of the test cases in order to evaluate the implemented tool and its characteristics.
6. **Work documentation** and redaction of the thesis.

Chapter 3

State of the art

Before going into the details of the task allocation issue, it is necessary to understand where the problem comes from. That is why the first section of the chapter is dedicated to multi-robot systems in general. After this short overview, we will introduce the elements and methods essential to understand and solve the MRTA problem.

3.1 Multi-Robot Systems

As a large and broad problem will gain at being divided into smaller sub-problems, or a huge project in a list of small and precise tasks, it is often preferable to operate with several simple robots than designing one super-powerful robot. Working with a team of robots provides several advantages [16], the first one being the possibility to combine the capacities and abilities of multiple robots to accomplish complex tasks that would be out of the reach of one individual. Multi-robots systems will then often be more cost-effective that single-robot systems. What's more, they are usually designed in a distributed and decentralized fashion and often exhibit redundancy, which considerably improves the robustness and the reliability of the global system, and enables parallelism. [9][13]

3.1.1 Overview and Classification

Broadly speaking, a MRS is a system composed by two or more robots, that can be similar or not. It is possible to imagine a great variety of MRS. To better apprehend this topic and ease the research, several classifications have been proposed so far.

Most papers classifies MRS according to some characteristic architectural features. For instance, in [12], Farinelli et al. organize systems around two axes: the *coordination dimension* and the *system dimension*. The former describes the type of coordination that is achieved (in terms of *cooperation*, *knowledge*, *coordination strength*, and *organization*), while the latter gathers the most relevant system features (*communication*, *team composition*, *system architecture* and *team size*).

Another classification, proposed by Darmanin and Bugeja in [9], organizes the MRS according to their application domain. The authors distinguish between six main functions, most of which can be found in [12] as well:

- *Surveillance, Search and Rescue* is an application that has gained in popularity with the development of unmanned aerial vehicles. These drones and other robots can for instance be used to patrol and monitor outdoor areas or for humanitarian assistance purposes.
- *Foraging and Flocking* are usually associated with swarm robotics, taking inspiration in the natural behaviour of bees or ants.

- *Formation and Exploration* are contrary requirements imposed to a team of robots. In *Formation*, the robots must preserve a strict arrangement and manage to avoid obstacles. In *Exploration*, they should distribute themselves to optimally cover the area.
- *Cooperative Manipulation* is required in *box-pushing* or similar problems, in which several robots should cooperate and coordinate to move an object.
- *Team Heterogeneity* refers to MRS platforms made up of heterogeneous components, usually selected for their complementary abilities. In their paper, Darmanin and Bugeja also consider scenarios involving human agents.
- *Adversarial Environment* designates applications in which a team of robots has to compete against some opponents. The context can either be sport/play, like in the RoboCup robot soccer competition, or military missions, with interventions on the battlefield.

3.1.2 Challenges

Since the late 1980s, much effort has been put to develop multi-robot systems. These setups come of course with challenges to be tackled, notably:

Control and coordination The burden of multi-robot systems lies in the control of the several entities. To really make the most of this particular design, it is important to design control such that the different agents are used optimally. In swarm exploration for instance, if the robots have to examine a set of given targets, it would be useless to have all agents explore the same target. In another type of example, if one want several mobile manipulators to lift a box and move it to another location, it is crucial that all robots act in a coordinated fashion. The control can be performed by one master that rules the whole system or can be implemented in a distributed way.

Communication For the first point to work properly, it is essential to ensure good communication between the different elements of the system. As for the control, there is no one-size-fits-all solution: communication can be centralized by one top entity or be managed by each team member. In swarm applications, it can even be implicit: the transmission of information is performed through the environment, as when insects leave pheromone trails to mark explored routes [13]. Communication also becomes an important challenge in exploration task, where distance between robots and battery levels can complicate the process.

Localization and mapping In many applications, it is necessary for the robots to know their environments in order to compute paths and move from one target to another, to avoid collisions or to retrieve the localization of an object. Depending on the problem, the robots can either be provided with extensive information about their surroundings or may have to build their own map of the environment. The latter is known as *mapping*. *Localization* refers to finding one's self location in the environment. These two problems are of course not restricted to multi-robot setups. However, it can be underlined that, when working with several robots, the paradigm changes a little. On the one hand, each robot may have to know the localization of its peers, which can be required to avoid collisions but also to perform certain tasks. On the other hand, depending on their perception abilities, the different team members can help one another to retrieve data about the environment and/or locate themselves more accurately.

3.2 Multi-Robot Task Allocation

3.2.1 The Multi-Robot Task Allocation problem

3.2.1.1 Problem statement

As mentioned in the previous chapter, coordination is one of the most important theoretical aspects of MRS. As in any team, it is indeed crucial to think about how to distribute the work and payload in an optimal manner, in order to be as efficient as possible and to make the most of the MRS. That is why multi-robot task allocation (MRTA) has become a key research topic.

MRTA aims at answering the following question:

Which task(s), and in which order, should execute which robot to complete the goal?

Formally, the MRTA problem instance can be stated as: *Finding an optimal allocation A of a set of tasks T to a subset of robots R, that will be in charge of carrying it out:*

$$A : T \longrightarrow R$$

3.2.1.2 Problem modeling

The MRTA problem may be modeled in various manners, among which one can find [16]:

- *ALLIANCE Efficiency Problem*: The ALLIANCE algorithm is an optimization algorithm used to solve NP problems. The algorithm is inspired by the situation in which several *tribes*, each with given skills (which model capacities), attempt to conquer necessary resources that can be found in the environment. The tribes should carry out *alliances* to organize themselves and optimize the resources retrieval.
- *Fair Division Problem*: In this problem, the set of tasks T is expected to be divided into *fair shares* between R robots, so that each of them may accomplish an equivalent job. In this problem, the tasks can be considered indivisible and assign entirely to one agent only, or divisible. In that case, the division is not necessarily homogeneous, which allows to work with heterogeneous robot team.
- *Multiple Traveling Salesman Problem*: It consists in a variant of the Traveling Salesman Problem, in which there are several salesmen. The Traveling Salesman Problem is a classical optimization problem in which a salesman should determine the best possible route to visit a list of cities, starting and ending at a the same given origin city. In the multiple version, the aim is to determine for each of the salesmen a route so that they all at least visit one city and return to the base city, while covering as a group all the cities and minimizing the total cost (usually distance or time).
- *Optimal Assignment Problem*: The MRTA problem may be seen as an instance of a fundamental combinatorial optimization problem. Given a set of tasks and a set of agents, one agent should be assigned to each task in order to complete the whole set. Usually the problem requires to optimize some cost. If the number of agents is equal to the number of tasks and the total cost is the sum of the cost for each agent, the assignment problem is said to be *linear*.

The MRTA problem can be solved by merely looking for a feasible solution, i.e. one that enables to accomplish the given set of tasks without violating any of the potential constraints. In most cases, however, the objective is also to maximize/minimize some function of the reward/cost, in order to optimize the resources and/or the efficiency of the system. Problems can naturally be multi-objectives. Among the common **optimization objectives**, one may find [15]:

- Minimize the sum of path costs over all the robots (MiniSUM). The most common cost to be considered is the distance traveled.
- Minimize the maximum path cost over all the robots (MiniMAX). Similarly, the objective can be to minimize the makespan, i.e. the time needed to accomplish the complete set of tasks.
- Minimize the average per task cost of the path over all the robots (MiniAVE).
- Minimize lateness or tardiness.
- Maximize the number of tasks that are eventually completed.
- Minimize the number of robots that are used.
- Maximize the profit, i.e. the difference between the reward gained and the cost paid for each task.

As an example, consider the basic instance of the problem, in which any robot r_i, $i \in [1, m]$ may perform any task t_j, $j \in [1, n]$ and can perform a maximum of N_i tasks. Let c_{ij} denotes the cost of r_i performing t_j and let a_{ij} be the binary variable equal to 1 if t_j is assigned to r_i and 0 otherwise. Then the problem can be formulated as a linear programming instance:

$$\min_{a_{ij}} \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} c_{ij}$$

subject to

$$\begin{cases} \sum_{i=1}^{m} a_{ij} & = 1, \quad \forall j = 1, ..., n \\ \sum_{j=1}^{n} a_{ij} & = N_i, \quad \forall i = 1, ..., m \\ a_{ij} & \in \{0, 1\} \quad \forall i, j \end{cases}$$

3.2.1.3 Task decomposition

As mentioned in introduction when stating the objectives, it is assumed in the present work that the final task decomposition is given to us. Still, it seemed important to give some insight about task decomposition, in order to understand at which point in the process our work integrates itself.

The starting point of the problem is the necessity to achieve a given **overall goal**, for instance "move the box B to point P" or "make a coffee". As one can see, these instructions are quite high-level, and they require more than one action to be completed. The first part of the analysis is therefore to break down the global goal into more specific subgoals. In [28], Zlot considers the following task decomposability properties:

- **Decomposability**: a task is decomposable if there exists a set of subtasks which, satisfying some specified combinations, achieve the given task.
- **Multiple decomposability**: a task is multiply decomposable if it admits more than one possible decomposition.
- **Full decomposability**: a task is fully decomposable if there exists a set of simple subtasks that can be derived within a finite number of decomposition steps.

Based upon these definitions, Zlot defines the task classification reported below:

- **Elemental (or atomic) task**: task that is not decomposable.

- **Decomposable simple task**: task that can be decomposed into elemental or decomposable simple subtasks, provided that there exists no decomposition of the task that is multirobot-allocatable.
- **Simple task**: task that is either an elemental task or a decomposable simple task.
- **Compound task**: task that can be decomposed into a set of simple or compound subtasks, provided there is exactly one fixed full decomposition of the task.
- **Complex task**: task that is multiply decomposable and for which there exists at least one decomposition that is a set of multirobot-allocatable subtasks. The subtasks may be simple, compound or complex.

One can easily understand that the task decomposition has an impact on the MRTA results and the performance of the MRS. Therefore, an additional question to be answered prior to the allocation is this one: *which simple tasks should be considered?* This is of particular importance when dealing with complex tasks that admit more than one decomposition.

The initial user orders most likely consist in complex or compound tasks. In our work, it is assumed that they have been split into **elemental tasks**, that make up the input to the system to be developed.

Note that this hypothesis implies another implicit assumption: it is that the final system follows the *decompose-then-allocate* paradigm, in which the tasks are decomposed into subtasks that are then allocated. Another approach is the *allocate-then-decompose*, in which the tasks are assigned to robots and then decomposed locally [16]. Both approaches have drawbacks that the other does not present. For instance, the decompose-then-allocate technique can be less flexible since it can lead to costly plans that cannot be changed. The allocate-then-decompose method, however, can suffer from the fact the task decomposition will be dependent on the agent, yielding sub-optimal solutions. Consequently, it may not be suited for heterogeneous teams - which supports our choice of approach.

3.2.2 MRTA problem taxonomies

The multi-robot task allocation problem is quite general, and can be found within a great variety of configurations. It is therefore useful to be able to **classify the specific problem** to be tackled, in order to narrow down the scope of possibilities. This is the object of the different taxonomies presented here.

3.2.2.1 Gerkey and Matarić's taxonomy

Gerkey and Matarić were the firsts to present a formal study of MRTA problems, in an attempt to provide grounding to the research on multi-robot coordination. Their work had a significant impact on the community, and most of the research in MRTA is based on their original analysis. In [14], they proposed a general taxonomy of MRTA problems along three axes (fig. 3.1):

- **Single-task robots (ST)** vs **multi-task robots (MT)**: each robot can execute at most one task at a time (ST) or some robots can execute multiple tasks simultaneously (MT).
- **Single-robot tasks (SR)** vs **multi-robot tasks (MR)**: each task requires exactly one robot to achieve it (ST) or some tasks can require multiple robots (MR).
- **Instantaneous assignment (IA)** vs **time-extended assignment (TA)**: the information about the robots, tasks and environment may only allow an instantaneous assignment of tasks (IA); if more information is available, like how will tasks arrive over time, it may be possible to plan taking into account future allocations (TA).

Each particular problem can then fall into a category denoted by a triple of two-letter abbreviation. For instance **ST-SR-IA** refers to the **Single-Task Robots. Single-Robot Tasks, Instantaneous Assignment** problem. This is

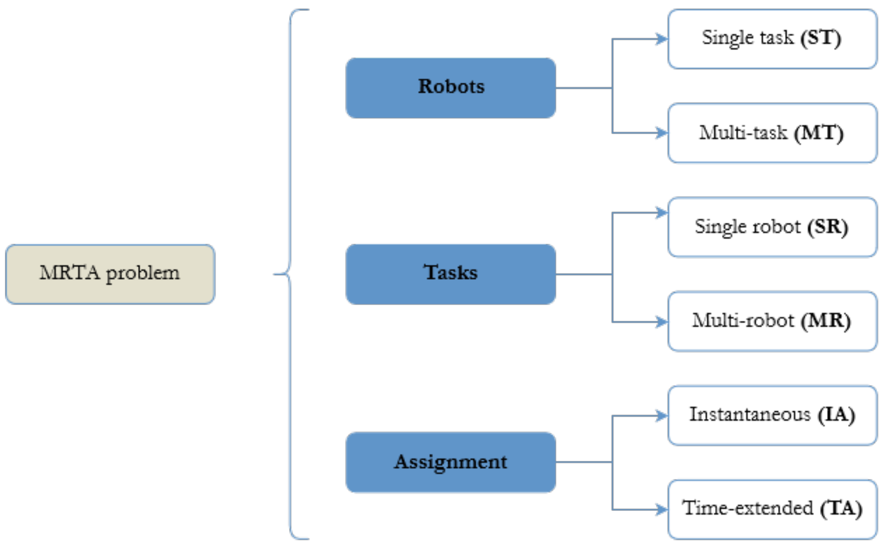


Figure 3.1: Gerkey and Matarić's taxonomy

the simplest situation which can be considered as an instance of the optimal assignment problem. In their publication, Gerkey and Matarić dwelled on each category, analyzing the theory behind each problem. This also explains why their work was such a founding one.

This taxonomy is however not exhaustive and do not consider some important problems. In their approach, Gerkey and Matarić considered *utility* as the main measure to be optimized, where the utility is defined as:

$$U_{RT} = \begin{cases} Q_{RT} - C_{RT} & \text{if R can execute T and } Q_{RT} > C_{RT} \\ 0 & \text{otherwise} \end{cases}$$

where Q_{RT} is the expected quality of execution and C_{RT} the cost if robot R executes task T. One issue is the fact that the utilities for different tasks may be interrelated, meaning the robots' utility for a task may depend on the other scheduled tasks: in their classification, Gerkey and Matarić assumed independent utilities.

Besides utility independence, a strong assumption of their work is the **independence of tasks**, meaning the task execution sequence has no importance. This is a significant limitation, since it does not allow to consider constraints among the tasks such as precedence or simultaneity ones. As a workaround, the authors suggest to consider a set of constrained tasks as one large multi-robot task, admitting that this considerably complicates the problem.

3.2.2.2 Korsah et al.'s *iTax*

Since constrained tasks are quite common, further research has been carried since the publication of this first taxonomy. Around a decade later, Korsah et al. presented a new taxonomy for MRTA, called *iTax* [17]. This comprehensive taxonomy aims at taking into consideration the key issues mentioned above.

In particular, constraints are defined as functions that restrict the space of feasible solutions. They can be of various types, for instance:

- **Capability** constraints: some robots may not be able to perform certain tasks.
- **Capacity** constraints: a robot can only perform a certain number of tasks at a time.
- **Simultaneity** constraints: two tasks must be performed at the same time.
- **Non-overlapping** constraints: two tasks must *not* be performed simultaneously.
- **Precedence** constraints: one task must be performed before another.

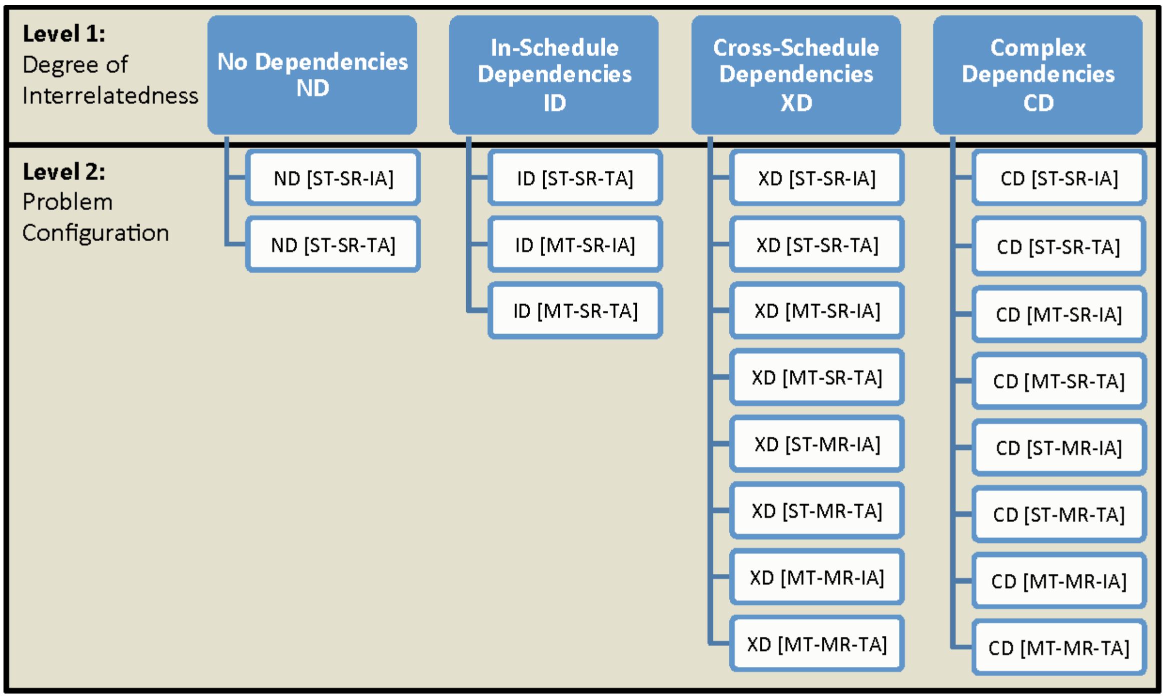


Figure 3.2: iTax task allocation taxonomy [17]

- **Proximity** constraints: two tasks must be performed at a certain distance from each other.

These constraints can be used to perform task decomposition, following the terminology developed by Zlot in [28] (see 3.2.1.3). As a consequence, a problem can be equivalently represented using either a set of independent compound tasks, either a set of simple tasks related by some inter-task constraints.

To tackle the problem of interrelated utilities, the definition is adapted as follows:

$$U_{RT} = \begin{cases} Q_{RT} - C_{RT} & \text{if subteam R can execute subset T} \\ -\inf & \text{otherwise} \end{cases}$$

The *iTax* is built upon the original classification by Gerkey and Matarić. To the three axes, it adds another dimension, namely the *degree of interdependence* of agent-task utilities. This dimension is characterized by a single categorical variable which values can be:

- **No Dependencies (ND)**: problem with simple or compound tasks that have independent agent-task utility.
- **In-Schedule Dependencies (ID)**: problem with simple or compound tasks with intra-schedule dependencies, i.e. the effective utility of an agent for a task depends on its other assigned tasks.
- **Cross-Schedule Dependencies (XD)**: problem with simple or compound tasks with inter-schedule dependencies, i.e. the effective utility of an agent also depends on the schedule of the other agents.
- **Complex Dependencies (CD)**: problem that exhibits inter-schedule dependencies for complex tasks, i.e. the dependencies are determined by the chosen task decomposition.

The final taxonomy is shown in figure 3.2: it consists in two levels. The first level is the additional dimension proposed by Korsah et al., while the second level follows Gerkey and Matarić's work. It is interesting to note that not all the theoretical subcategories are represented; some are indeed not relevant.

3.2.2.3 Nunes et al.'s MRTA/TOC

More recently, Nunes et al. proposed a more specific taxonomy for task allocation problems with temporal and ordering constraints [23]. The MRTA/TOC classification (which name stands for Multi-Robot Task Allocation with Temporal and Ordering Constraints) is also based on Gerkey and Matarić's earlier work. More specifically, it maintains its subcategories but extends the time-extended assignment (TA) part by considering:

- on the one hand, temporal constraints in the form of **time windows (TA:TW)**.
- on the other hand, temporal constraints in the form of **synchronization and precedence constraints (TA:SP)**.

A time window constraint is a constraint on the start or finish time (or both) of a task that is expressed in the form of a temporal interval. It usually sets the earliest start time and latest finish time of the task. Precedence and synchronization constraints are constraints that specify an order for the task sequence. This second type of constraints can be expressed implicitly by time windows, but it is generally not enough.
Additionally, one can distinguish when meaningful between:

- *hard* and *soft* temporal constraints. Hard temporal constraints are constraints that cannot be violated: if they are executed out of their specified time, the gained utility is null. On the contrary, tasks with soft temporal constraints may be started early or finished late with only an additional penalty that decreases the utility (until it eventually reaches a hard deadline).
- *deterministic* and *stochastic* models. In the former, the output is completely determined by the initial conditions while the latter takes into account uncertainty.

3.2.3 Methods for task allocation

Now that the study of the different problem taxonomies helped to clarify the problem, let's present some of the possible solutions for the actual task allocation process. This section is not intended as an exhaustive review, and focuses on the most common methods.

Due to the complexity of the MRTA problem and the broad variety of problem instances, countless approaches have been undertaken in an attempt to solve this question. A common criteria to classify the different techniques is the team organizational paradigm, which distinguishes between centralized and decentralized methods [16]. **Centralized methods** depend on a central controller to perform the task assignment. All the robots must maintain a connection to the central unity, in order to send it all the information they have. On the contrary, in **decentralized methods**, each agent participates to the task allocation process.

3.2.3.1 Centralized methods

In centralized methods, the controller, which can be one of the robot or a ground station, is responsible for the entire MRTA process. Therefore, the autonomy of the other agents in the system is limited or null. This architecture provides some advantages, among which the possibility to better optimize the efforts and resources to save cost and time [16], as well as the fact that the controller usually has complete knowledge about the environment [11]. On the other hand, this approach can be expensive in terms of communication resources. Above all, it suffers from a major drawback which is the single point of failure, i.e. if the central entity breaks down, the whole system is impacted. Also, fully centralized approaches are restricted to small-sized teams, since the central controller is a bottleneck of the system.

The main methods within centralized solutions are optimization-based approaches.

Optimization-based approaches: As mentioned in section 3.2.1.2, the MRTA issue can be modeled as an optimization problem. Given some constraints that reduce the set of feasible solutions, the idea is to find the optimum solution, i.e. the feasible solution that best fulfills the objectives. Optimization is a broad field of mathematics, and there exist lots of algorithms. They can be deterministic, like the classical or numerical methods (such as gradient-based methods, quadratic programming, etc), or stochastic.

When looking for an optimal solution, it may be possible in some cases to find an **exact solution**. Nonetheless, it is most often impractical because of the computational issues. A naive exhaustive search for all the possible allocations that satisfy the problem constraints is intractable: the worst case for $|T|$ tasks and $|R|$ robots indeed leads to a complexity $O(|T|!)^{|R|}$. More efficient methods include the branch-and-bound algorithm and its variants, or specific tools designed to solve Mixed Integer Linear Programming (MILP) formulations. To overcome the computational issues, it is possible to work with metaheuristic approaches or MILP-based heuristics. This **approximate methods** are a good way to reduce the computational time, at the cost of a lower quality solution in general.

3.2.3.2 Distributed methods

Distributed setups are often preferred, since they can provide better reliability, flexibility and scalability. In case one agent fails, the rest of the team may carry on, and a new agent can be more easily added. Less communication is needed, which can be of interest when working with mobile robots or with few resources. In particular, auction-based methods in which each robot maintains its schedule and computes its utility are a good way to cut down on communication costs [15]. These advantages, like in all solutions, come with some weaknesses - the main drawback of decentralized methods being the fact they can generate sub-optimal solutions [16].

Among decentralized approaches, the most popular methods are Distributed Constraint-Based (DCOP) and auction-based ones.

Distributed Constraint-Based approaches: MRTA problems that include constraints may be formulated as Distributed Constraint Optimization Problem and therefore solved using the corresponding tools. Finding an exact solution however is often impractical since it is NP-Hard. Consequently, most approaches are approximate DCOP techniques [15].

Market-Based approaches: Among the most popular methods, one can find auction- and negotiation-based approaches, which are economically inspired. The algorithms mimic an auction, in which each agent receives the set of tasks to be assigned and bids according to its utility. The auction-based algorithm can be summarized as follows [11][16]. Each auction round consists in several phases:

1. **Announcement**: the set of available tasks is communicated to each agent by the *auctioneer*.
2. **Submission**: each robot computes a **bid** for each task put up for auction and sends its best offer to the auctioneer. This bid is a number that reflects the utility or the cost if the robot executes the task.
3. **Selection**: all the bids are collected by the auctioneer which then selects the winning bidder according to the optimization objectives.
4. **Contract**: the result is communicated to all the robots-bidders, and the winner can add the task to its schedule.

The process is repeated until all the tasks have been assigned. Depending on the exact process, the communication needs are more or less demanding. Different auction schemes exist, for instance *combinatorial auctions* or *single sequential item auction*, the latter being more parsimonious.

Market-based approaches have gained in popularity thanks to several advantages [16]. First, they work using local information and the preferences/capabilities of each team member, which make them **efficient**. Since they are based on a distributed paradigm, they offer **robustness** and **scalability** depending on their implementation. What's more, they can be easily implemented to work **online**, which allow to process dynamic arrival of tasks. Of course, they are no miracle solution. As their main flaw, it should be mention that, because each agent is "self-interested", the solution that is obtained can be sub-optimal. Additionally, it may be difficult to properly design the bid calculation functions.

Summary

This chapter covered the review of the state of the art, introducing first multi-robot systems. Within the presented classification and related challenges, our problem falls within the *Team Heterogeneity* category and tackles, notably, the coordination challenge.

The second part of the chapter focused on the specific task allocation issue. The problem, which can be formulated as *finding an optimal allocation of a set of tasks T to a subset of robots R*, can be modeled in various ways and usually involves an optimization objective, for instance minimizing the completion time or the distance covered. MRTA problems are generally classified using a three-axis taxonomy, which distinguishes between single- or multi-task robots (ST/MT), single- or multi-robot tasks (SR/MR), and instantaneous or time-extended assignment (IA/TA). This taxonomy, however, needs to be extended to take into account inter-task relations, in particular precedence constraints. Finally, the chapter also presented the main methods to perform task allocation; among them, auction-based solutions seem the most promising.

Chapter 4

Solution design

Taking into account the information from the state of the art reviewed in the previous chapter, and considering the objectives stated at the beginning of this report, it is now possible to move on to the design of the MRTA solution that better fits the AURORA needs. This chapter presents the chosen method considering the previous analysis. It also dwells on the formal modelization of the MRTA problem instance with precedence constraints.

4.1 Objective reminder and problem classification

As mentioned several times previously, the MRTA problem is a complex issue to solve. That is why in this work, the focus was only set on building a feasible schedule from a given set of elemental tasks. Let us briefly remind the constraints to be taken into account:

- heterogeneous capabilities of the robots,
- capacity of the robots (at most one task),
- intertask constraints due to precedence relationships,
- presence of a human in the loop.

What is more, the system should exhibit the following characteristics:

- flexibility,
- scalability,
- robustness,
- functionality.

From the analysis of the different constraints, one can see that our problem falls within the general **ST-SR-TA** category from Gerkey and Matarić. It is interesting to note that this classification would be totally different, had we considered the higher-level tasks. In that case, the whole set of tasks would not be known prior to the system start up and would be on the contrary dynamically updated according to the user's needs. What's more, a given task would require the intervention of several robots. The problem would therefore have been labeled as ST-MR-IA.

As underlined before, because we work with the lower-level tasks, it is essential to take into account precedence constraints. The gap in Gerkey and Matarić classification can be filled by using the enhanced taxonomy by Nunes et al. which has been presented in section 3.2.2.3. Following [23], our problem can be categorized as **ST-SR-TA:SP**.

4.2 Solution

4.2.1 Chosen method

Despite the plethora of algorithms that can be found in the literature, several critical aspects are given little consideration, in particular complex tasks allocation, dynamic task allocation, constrained task allocation and heterogeneous allocation [16]. Most examples actually deals with variant of the vehicle-routing problem (VRP), or make strong hypotheses about the task independence that limit their reuse.

Now, our problem being an instance from the **ST-SR-TA:SP** category, it is crucial to be able to deal with intertask constraints.

An interesting method is presented in [21], in which the tasks are divided in disjoint sets. To keep to the precedence constraints, the sets are strictly ordered and each robot may perform at most one task per set. The algorithm proposed relies on a distributed auction-based technique and is proven to be complete and sound. The drawback of the method is of course that it strongly constrains the achievable precedence graphs. In particular, the authors assume that all the tasks require the same amount of time to be completed.

For this thesis, it was decided to follow the work by McIntire et al. who presented in [22] an iterated multi-robot auction process for precedence-constrained task scheduling. Despite certain drawbacks, auction-based methods are indeed the most appropriate to work with heterogeneous teams, since they are the most suited to take into account the different capabilities [16].

The algorithm proposed is proven to be sound and complete with a complexity $T(n, m)$ that satisfies

$$O(\frac{n^5}{m^4}) \leq T(n, m) \leq O(\frac{n^6}{m^4})$$

with m the number of robots and n the number of tasks. The experiments reported in the original paper are all based on data sets created for the vehicle routing problem. Currently, it seems that no data set dedicated to MRTA problem with task ordering constraints exists. Consequently, those data sets were modified to include precedence constraints in order to properly test the algorithm.

At first sight, the VRP does not appear as linked to our problem. Still, two comments can be made. First, it is actually possible to draw a parallel between the two problems. In the AURORA setup, it is indeed possible to assign a location to each task: the glass should be placed at point A, the robot must reach point B to pick the bottle... and so on. To a certain extent, it can therefore be conceived as similar to the VRP. Second, the paper has the advantage to provide a general model for task allocation with any type of precedence constraints, meaning its application is not restricted to VRP instances.

4.2.2 Problem formulation

4.2.2.1 Task set

Let $T = (t_i)_i$ be the set of n tasks. Each task is associated a location where it will have to be executed. Some precedence constraints between the tasks may exist: they are represented using a *precedence graph* $G_p = (T, E)$. E is the set of edges that are effectively used to represent the constraints: $(t_i, t_j) \in E$ means that task t_j is a successor of task t_i and can not be started before t_i has been completed. Contrary to [21], any precedence graph may be considered; however, one should note that for the problem to be feasible, G_p must be a directed **acyclic** graph.

4.2.2.2 Robot set

Let $R = (r_i)_i$ be the set of m robots. Each robot has an initial location. In the original paper, R is **considered to be homogeneous**: this is obviously not the case in our scenario. To **take into account heterogeneity**, some modifications are introduced to the algorithm. They will be explained later on.

4.2.2.3 Objective

As mentioned several times when presenting the general MRTA problem, the objective is to find a valid schedule for task execution, that will consist in 1) partitioning T across R and 2) scheduling the tasks assigned to each robot so that the precedence constraints are respected. The optimization objective will be to minimize the makespan, i.e. the total schedule duration.

4.2.3 Mathematical model

Let's consider the following decision variables:

- Z, the makespan
- S_{t_j} and F_{t_j}, the start and finish times of task t_j, $j = 1, ..., n$
- $A_{t_j}^{r_i}$, the binary assignment variables:

$$A_{t_j}^{r_i} = \begin{cases} 1 & \text{if task } t_j \text{ is assigned to robot } r_i \\ 0 & \text{otherwise} \end{cases}$$

- $Y_{t_j}^{r_i}$, the binary assignment variables:

$$Y_{t_j}^{r_i} = \begin{cases} 1 & \text{if task } t_j \text{ is the last task assigned to robot } r_i \\ 0 & \text{otherwise} \end{cases}$$

- $X_{t_j,t_k}^{r_i}$, the precedence ordering variables

$$X_{t_j,t_k}^{r_i} = \begin{cases} 1 & \text{if robot } r_i \text{ performs task } t_j \text{ just before task } t_k \\ 0 & \text{otherwise} \end{cases}$$

Additionally, let's consider:

- $d(t_j, t_k)$, the travel time between the respective locations of task t_j and task t_k
- $dur(t_j)$, the duration of task t_j
- T_d a set of dummy tasks with no duration that are used to represent the starting position of each robot
- T_{prec} the set of task dependencies

The problem can then be formulated as the Mixed Integer Programming instance reported (again following [22]) in the equations below:

$$\text{minimize } Z \tag{4.1}$$

subject to

$$Z \geq F_{tj}, \forall t_j \in T \tag{4.2}$$

$$\sum_{r_i \in R} A_{t_j}^{r_i} = 1, \forall t_j \in T \cup T_d \tag{4.3}$$

$$\sum_{t_j \in T_d} A_{t_j}^{r_i} = 1, \forall r_i \in R \tag{4.4}$$

$$\sum_{t_j \in T} Y_{t_j}^{r_i} = 1, \forall r_i \in R \tag{4.5}$$

$$\sum_{t_j \in T \cup T_d, t_j \neq t_k} X_{t_j,t_k}^{r_i} = A_{t_j}^{r_i}, \forall t_k \in T, r_i \in R \tag{4.6}$$

$$\sum_{t_k \in T, t_j \neq t_k} X_{t_j,t_k}^{r_i} + Y_{t_j}^{r_i} = A_{t_j}^{r_i}, \forall t_j \in T \cup T_d, r_i \in R \tag{4.7}$$

$$F_{t_j} + d(t_j, t_k) - M(1 - X_{t_j,t_k}^{r_i}) \leq S_{t_k}, \forall t_j, t_k \in T, r_i \in R \tag{4.8}$$

$$0 \leq S_{t_j} \leq \infty, \forall t_j \in T_d \tag{4.9}$$

$$0 \leq F_{t_j} \leq \infty, \forall t_j \in T \cup T_d \tag{4.10}$$

$$F_{t_j} - S_{t_j} \geq dur(t_j), \forall t_j \in T \tag{4.11}$$

$$S_{t_k} - F_{t_j} \geq 0, \forall (t_j, t_k) \in T_{prec} \tag{4.12}$$

$$A_{t_j}^{r_i} \in \{0,1\}, \forall t_j \in T \cup T_d, r_i \in R \tag{4.13}$$

$$Y_{t_j}^{r_i} \in \{0,1\}, \forall t_j \in T, r_i \in R \tag{4.14}$$

$$X_{t_j,t_k}^{r_i} \in \{0,1\}, \forall t_j, t_k \in T \cup T_d, r_i \in R \tag{4.15}$$

- Constraint (2) defines the makespan Z as the global latest finish time ($Z = \max_{t_j \in T} F_{t_j}$).
- Constraint (3) ensures that each task is assigned to exactly one robot.
- Constraint (4) ensures that each robot has exactly one dummy task.
- Constraint (5) ensures that each robot has exactly one final task.
- Constraint (6) ensures that each task that is not first in the schedule has a predecessor.
- Constraint (7) states that if robot r_i does task t_j, then t_j is the last task in r_i's schedule or has a successor.
- Constraint (8) ensures that the robot has time to travel between executing two tasks.
- Constraint (9-10) states that the start and finish times of a task should be positive.
- Constraint (11) enforces the task duration to be respected.
- Constraint (12) ensures that the precedence constraints are respected, by enforcing the start time of a task to be greater than the finish time of its predecessor.
- Constraint (13-15) define the admissible values for the binary assignment variables.

4.2.4 Algorithm

4.2.4.1 Auction preparation

As mentioned previously, the algorithm developed in [22] is a form of auction algorithm. In the proposed solution, the tasks are scheduled **in batches**. As a key hypothesis, it is assumed that **in each batch, the tasks are pairwise independent.** This enables to schedule all the tasks from one batch without considering the scheduling of the other tasks in the batch. The batch selection is performed by the auctioneer, which takes as input the precedence graph G_p defined above. At each iteration, it selects the tasks that have either no predecessor or which predecessors have all been scheduled. These tasks are said to be **free**. One can therefore split the precedence graph into **layers** (see figure 4.1):

- the first layer is made by the free tasks T_F;
- the second layer T_L, which are the tasks to be auctioned next, corresponds to the free tasks in $T \backslash T_F$;
- the rest of tasks, which are hidden, is denoted as $T_H = T \backslash \{T_F \cup T_L\}$.

This way of grouping the tasks ensures that the task pairwise independence assumption is respected and that no precedence constraint links two tasks from a same batch.

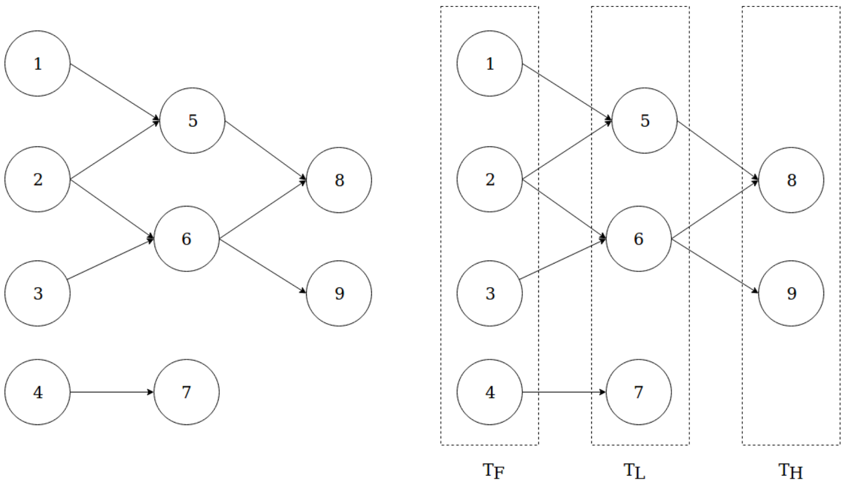


Figure 4.1: Example of precedence graph and layer splitting

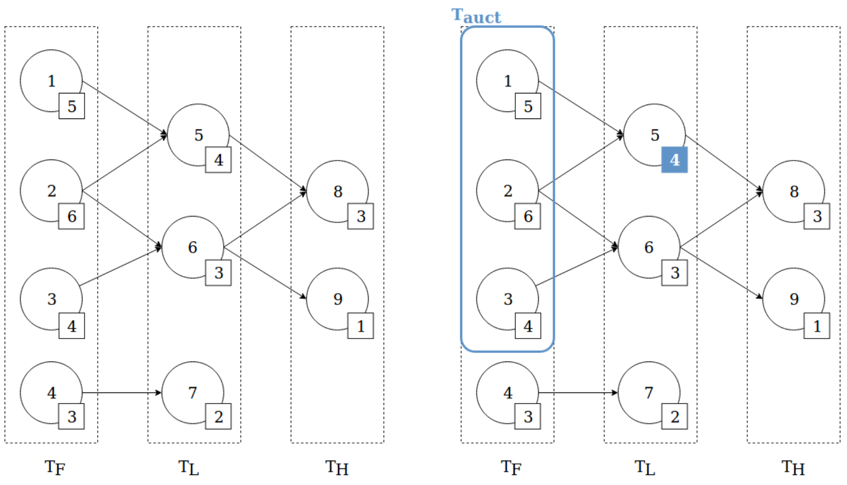


Figure 4.2: Example of precedence graph and layer splitting with priorities

Priorities: In the algorithm proposed by McIntire et al., tasks can also be assigned some numerical **priorities** in order to bias the process. In that case, the auction batch is constituted of the free tasks which priority is greater or equal than the highest priority of the second-layer tasks (see figure 4.2). This additional feature can be used to delay some of the first-layer tasks, in order to complete earlier more important tasks that are currently not free.

4.2.4.2 Auction course

The main body of the algorithm is roughly the same than the process explained in section 3.2.3. Once the batch of tasks to auction has been selected, it is broadcast to all bidders. To take into account heterogeneity, McIntire's solution is modified by adding the following step: before computing the bid for a task, the robot evaluates if it can do it with respect to its capabilities. Each robot then computes *for each **feasible** task in the batch and for each possible position in its current schedule* a bid.The best result of each robot is sent to the auctioneer, which selects the best "offer" and communicates the winner to the robots. The winner updates its schedule and all participants remove the assigned task from the tasks to be auctioned. The process is repeated until all the tasks from the auctioned batch have been scheduled.

Partial schedules: In order to **increase the robustness** of the process, we **add** to the algorithm proposed by McIntire the possibility to **label tasks as unfeasible**. If the offers received by the auctioneer are all null, it means that no bidder may perform the tasks remaining in the current batch. If this happens, these tasks **and their successors** are labeled as "unfeasible" and the auctioneer proceeds with the next batch. The successor tasks, that cannot be performed, are not considered. For instance, in the example of figure 4.1, if no robot can perform task 3, it is labeled as unfeasible, and so are tasks 6, 8 and 9. The process continues with a new batch composed of tasks 5 and 7.

As of now, the crux of the method has not been presented. It has been seen how the task independence assumption in each batch can be respected - however, the precedence constraints should at some point be taken into account: how can one make sure that they will indeed be satisfied? This is the object of the next paragraphs.

4.2.4.3 Task schedules

The process that has been described until now does not consider the ordering constraints on the tasks. The auctioneer will play no role in this, and all the responsibility is delegated to the set of bidders. As it can be inferred from the process explained above, **each robot keeps in memory its own task schedule**. The idea is therefore that each robot makes sure that its schedule remains valid, considering the task start and finish times as well as the possible travel time between locations.

The task schedules are represented using **Simple Temporal Networks** (STN), introduced in [10]. STNs are data structures similar to graphs, in which nodes symbolize events and edges symbolize timing constraints on the events.

In a typical *temporal constraint satisfaction problem* (TCSP), one can consider for each event a continuous variable X that represents a time point while the constraints are represented by a set of intervals. The constraints can be *unary*, i.e. constraining the value of the variable X_i, or *binary*, i.e. restricting the domain of the distance between two time points. Usually the point X_0 represents the beginning of times, to which all time points are relative. In particular, this enables to transform the unary constraint into binary constraints. The same approach will be applied to the MRTA problem, in which a dummy task with no duration will be used to represent the initial location and time.

The set of variables $(X_i)_i$ and unary/binary constraints forms a *network* that may be easily represented by a *directed constraint graph*. When looking for a **solution** to the temporal problem, one therefore looks for an assignment tuple $X = (X_1 = x_1, ..., X_n = x_n)$ that complies with all the constraints. If at least one such solution exists, the network is said to be **consistent**.

In our case, we shall consider only one interval per constraint. It is therefore possible to focus on what Dechter et al. call a *simple temporal problem (STP)*, which is an instance of the TCSP in which all edges are labeled by a single interval representing a constraint of the form:

$$a_{ij} \leq X_i - X_j \leq b_{ij} \iff \begin{cases} X_i - X_j & \leq b_{ij} \\ X_j - X_i & \leq -a_{ij} \end{cases}$$

Considering the right-hand inequalities, it is possible to represent the STP directed constraint graph as another graph in which the edges are doubled and labeled only by one scalar. This graph is called the **distance graph**.

If, in the distance graph, several paths may connect two nodes i and j, it is possible to verify that the distance between two points must be smaller or equal to the shortest path d_{ij} between the two nodes.

As a consequence, it can be stated that **a given STP is consistent if and only if its distance graph has no negative cycles**. This theorem is of interest and it will be used to determine if a robot schedule remains valid after inserting a new task into it.

The corollary of this theorem is at least as important, since it enables to find a solution to the STP:

Let consider a consistent STP. Two consistent scenarios (i.e. solutions) are given by:

$$S_1 = (d_{01}, ..., d_{0n}) \tag{4.16}$$

$$S_2 = (-d_{10}, ..., -d_{n0}) \tag{4.17}$$

which respectively assign its latest and earliest possible time to each variable. For more details, the interested reader can refer to [10].

Figure 4.3 shows a very simple example of robot schedule in which two tasks t_1 and t_2 have been inserted. The nodes of the STN are made up by the start and finish times of the two tasks. The nodes that correspond to the start and finish times of a given task are connected by an edge (in green) that specifies the task duration constraint: it should elapse between the task start and finish at least $dur(t_i)$, hence the semi-infinite interval. The edge between t_1 and t_2 (in yellow) specifies that t_2 must start after t_1, and not before it has elapsed enough time for the robot to travel from one location to the other. For simplicity, the origin point has not been represented. Rather, self-loop arrows (in blue) give the absolute value domain for each time point, i.e. start and finish times.

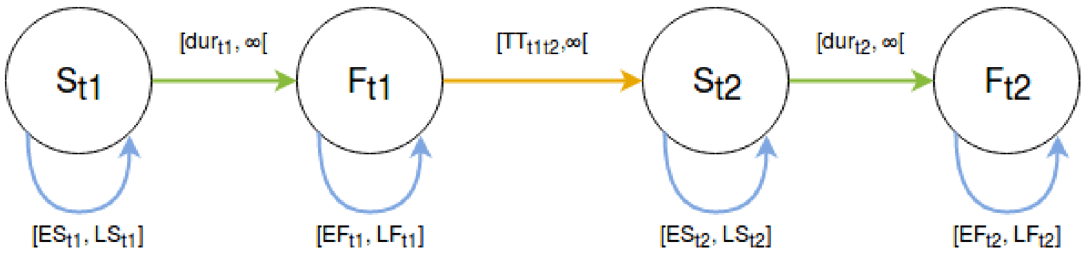
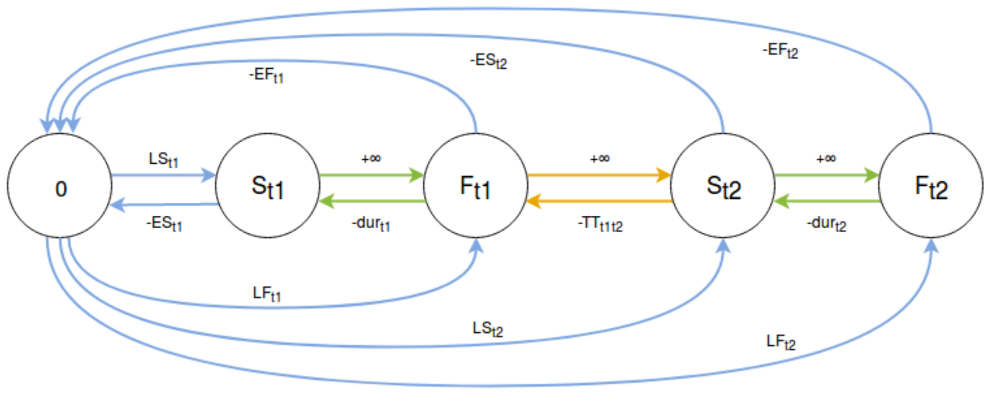


Figure 4.3: Example of robot schedule as a STN

As explained above, it is possible to associate to this network a distance graph: it can be seen in figure 4.4. Two representations are given: the first one is the graphical representation showing the treatment of the edges/constraints; the second is the matrix form $D = (d_{ij})$ of the distance graph. In the latter, the coefficient d_{ij} represents the distance from node i to node j.

Because of its importance, we have in this section lingered over the STN theory and how we shall rely on it, following [24]. Going back to the MRTA problem, let's sum up the main steps to maintain the task schedule:



(a) Graph

	0	S_{t1}	F_{t1}	S_{t2}	F_{t2}
0	0	LS$_{t1}$	LF$_{t1}$	LS$_{t2}$	LS$_{t2}$
S_{t1}	-ES$_{t1}$	0	∞	∞	∞
F_{t1}	-EF$_{t1}$	-dur$_{t1}$	0	∞	∞
S_{t2}	-ES$_{t2}$	∞	-TT$_{t1t2}$	0	∞
F_{t2}	-EF$_{t2}$	∞	∞	-dur$_{t2}$	0

(b) Matrix representation

Figure 4.4: STN distance graph

- Schedule representation:
  - the nodes of the STN represent the start and finish times of each task assigned to the robot.
  - the duration of a task constrains the time between its start and finish times.
  - the travel time of the robot constrains the time between the finish time of a task and the start time of its successor.
  - additional interval constraints specify the absolute values of the start and finish times of a task.
- Task insertion:
  - new task assignment are evaluated by checking for each possible insertion point in the schedule if the new schedule is valid.
  - the schedule is considered valid if no negative cycle occurs in the associated distance graph.
- Bidding:
  - the bid associated to a schedule is obtained by solving the STN and computing the corresponding makespan.

To respect precedence constraints, each robot sends at the end of an auction iteration the earliest finish times f of all the tasks in its schedule. The content of f serves two purposes:

- first, it is used to introduce new constraint in the STN, so that the task once scheduled may not be delayed;
- second, it is returned to the auctioneer which uses this information when updating the precedence graph. When a task is moved from T_L to T_F in order to be auctioned, its associated possible earliest start time value is updated to the latest finish time of its parent tasks.

This ensures that a task may not start before its predecessors has finished. What's more, it should be underlined that the exchange of information through the earliest start time and finish time vectors enables to respect the precedence constraints even if a task and its parents/children are not assigned to the same agent.

4.3 Algorithm pseudo-code

As a conclusion to this chapter, we provide the pseudo-codes for the different parts of the algorithm, as well as flowcharts describing the auction process for the auctioneer (fig. 4.5) and the bidders (fig. 4.6).

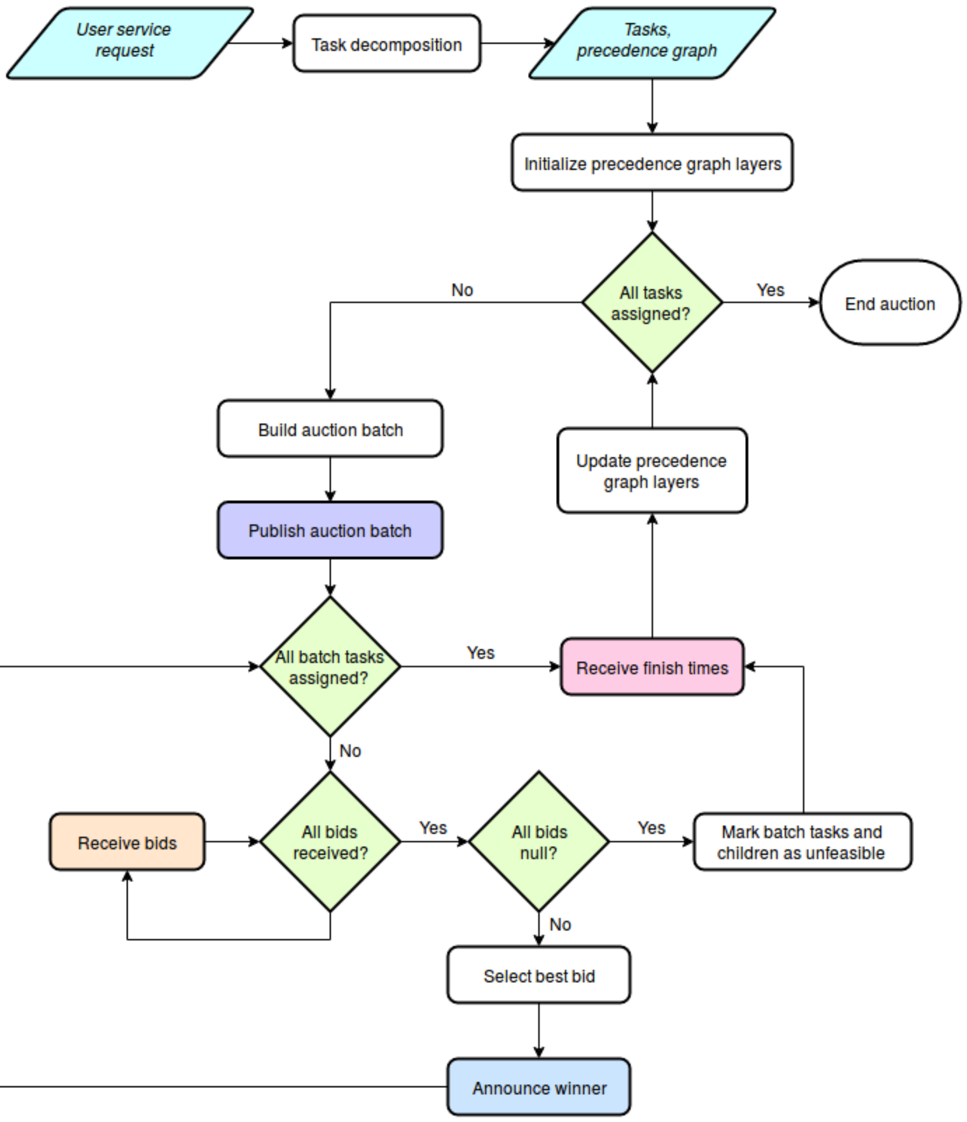


Figure 4.5: Auction process flowchart - auctioneer

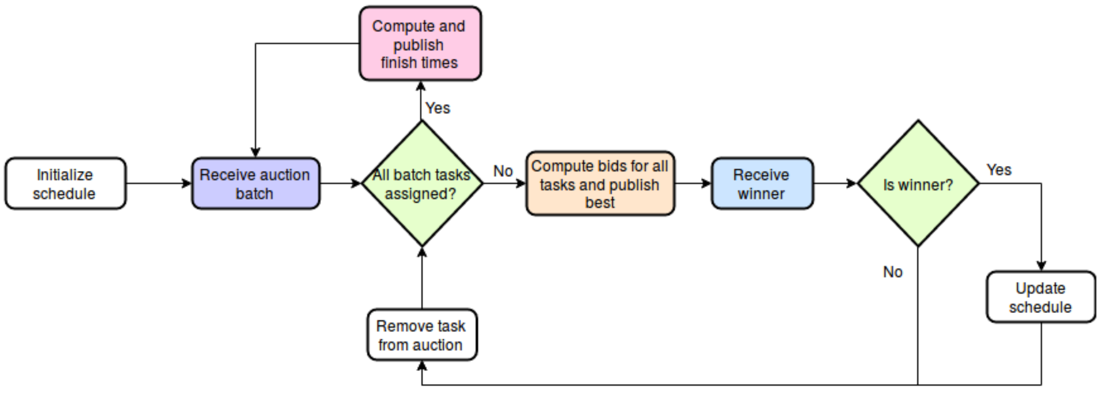


Figure 4.6: Auction process flowchart - bidder

Algorithm 1 Auctioneer's algorithm

```
1: procedure AUCTIONEER(Precedence graph $G = (T, C)$, bidders $R$)
2:     $T_S, T_U \leftarrow \emptyset$, $T_F \leftarrow free(T)$, $T_L \leftarrow free(T \setminus T_F)$, $T_H \leftarrow T \setminus (T_F \cup T_L)$
3:     $F \leftarrow 0$, $PC \leftarrow 0$
4:     while $|T_S| < |T|$ do
5:         $c \leftarrow max(priorities(T_L))$
6:         $T_{auct} \leftarrow t \in T_F : prio(t) \geq c$
7:         Send $T_{auct}, PC$
8:         $count \leftarrow 0$
9:         while $count < |T_{auct}|$ do
10:            Receive bids
11:            if all bids == null then
12:                for all unassigned task t $\in T_{auct}$ do
13:                    Move $t$, successors$(t) \longrightarrow T_U$
14:                    $T_{auct} \leftarrow T_{auct} \setminus t$
15:                end for
16:            else
17:                Select lowest bid
18:                Send winner
19:                $count \leftarrow count + 1$
20:            end if
21:        end while
22:        $F \leftarrow$ Receive finish times
23:        for all $t \in T_{auct}$ do
24:            Update precedence graph(t)
25:        end for
26:    end while
27: end procedure
```

Algorithm 2 Update precedence graph

```
1: procedure UPDATE PRECEDENCE GRAPH(task $t$)
2:     Move $t$: $T_F \longrightarrow T_S$
3:     for all $t' \in T_L \cap$ children($t$) do
4:         if parents($t'$) $\subset T_S$ then
5:             Move $t'$: $T_L \longrightarrow T_F$
6:             $PC[t'] \leftarrow \max_{t_p \in \text{parents}(t')}(F[t_p])$
7:             for all $t'' \in T_H \cap$ children($t'$) do
8:                 if parents($t''$) $\subset T_S \cup T_F$ then
9:                     Move $t''$: $T_H \longrightarrow T_L$
10:                end if
11:            end for
12:        end if
13:    end for
14: end procedure
```

Algorithm 3 Bidder r's algorithm for each iteration

```
1: procedure BIDDER($T_{auct}$, $PC$)
2:     while $|T_{auct}| > 0$ do
3:         $m \leftarrow \infty$, $t_{bid} \leftarrow \varnothing$
4:         for all $t \in T_{auct}$ do
5:             if $is\_feasible(t)$ then
6:                 for all position $p$ in schedule $S$ do
7:                     $S* \leftarrow$ try insert $t$ at $p$ in $S$
8:                     if $is\_consistent(S')$ and $makespan(S') < m$ then
9:                         $m \leftarrow makespan(S')$, $t_{bid} \leftarrow t$
10:                    end if
11:                end for
12:            end if
13:        end for
14:        Send $m$, $t_{bid}$
15:        $r_{win}, t_{win} \leftarrow$ Receive winner
16:        if $r_{win} == r$ then
17:            Insert $t_{win}$ in $S$
18:        else if $r_{win} == none$ then
19:            $T_{auct} \leftarrow \varnothing$
20:        end if
21:        $T_{auct} \leftarrow T_{auct} \setminus t_{win}$
22:    end while
23:    $F \leftarrow$ Tighten schedule
24:    Send $F$
25: end procedure
```

Algorithm 4 Tighten schedule

```
1: procedure TIGHTEN SCHEDULE(finishing times $F$, latest finish times $LF$, schedule $S$)
2:     for all $t \in S$ do
3:         $eft \leftarrow$ compute earliest finish time for $t$
4:         $F[t] \leftarrow eft$
5:         $LF[t] \leftarrow eft$                    ▷ Add constraint to avoid delays
6:     end for
7: end procedure
```

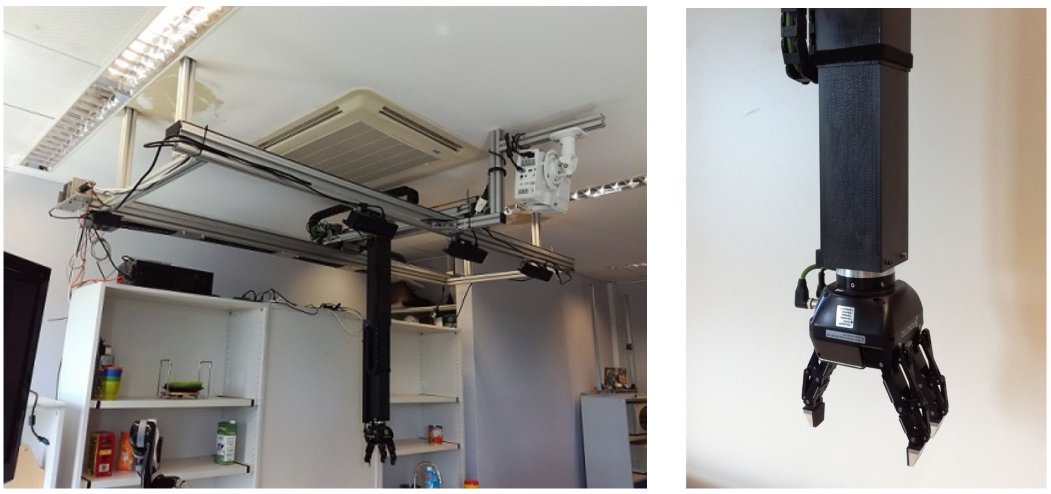
Chapter 5

Solution implementation

The present chapter is dedicated to the implementation of the solution designed in the previous chapter. We will first describe in more details the AURORA setup, starting with the different hardware and software tools that are available. Next, some specific details of the AURORA implementation will be presented. The remaining of the chapter will focus on the specific implementation of the algorithm introduced previously.

5.1 Robots

5.1.1 Capdi



(a) Structure frame (b) Arm with Robotiq adaptive gripper

Figure 5.1: The CAPDI robotic arm

Architecture The Capdi robot is a custom robot that has been designed, built and rebuilt within the laboratories of the UPC, first for the INHANDS project. It is a Cartesian three-DOF robot. It is composed of a vertical telescopic arm that allows moves along the z-axis mounted on a structure built of two frames that allow moves in x- and y-direction. The structure is attached to the ceiling, as it can be seen in figure 5.1a. Its simple design enables to adapt it easily to the user's kitchen.

Hardware The robot movements are controlled using three MERCURY motion controller boards from IngeniaCAT. The communication between the central computer and the hardware is performed over a CAN-BUS connection, using the PCAN Driver for Linux developed by the company Peak System (version 8.5.1).

End-effector The architecture of the Capdi makes it well suited to move objects from one side of the kitchen to the other or to supply the required products to the user. To this purpose, Capdi is equipped with Robotiq's 3-Finger Adaptive Robot Gripper (figure 5.1b), which comes along dedicated ROS packages. The hand is a versatile tool that provides four grip modes ("Pinch", "Wide", "Scissor" and "Basic") and is able to manipulate objects from various shapes and masses (see table 5.1). It is worth noting that the gripper is fixed to the wrist of the Capdi as it is and that its orientation is therefore constant - meaning that objects can only been grasped in one manner (the orientation of the gripper is constant).

Specification	Value
Gripper opening (mm)	0-155
Gripper weight (kg)	2.3
Object diameter for encompassing (mm)	20-155
Max. recommended payload encompassing grip (kg)	10
Max. recommended payload, fingertip grip (kg)	2.5
Grip force, fingertip grip (N)	30-70

Table 5.1: Specifications of the Robotiq's 3-Finger Adaptive Gripper

5.1.2 Kinova's Mico

The kitchen is also equipped with a MICO robotic arm from the Canadian company Kinova, which can be seen in figure 5.2. It is part of their JACO robotic arm range which has been specifically designed for the Assistive Robotics and Service Robotics fields [3].

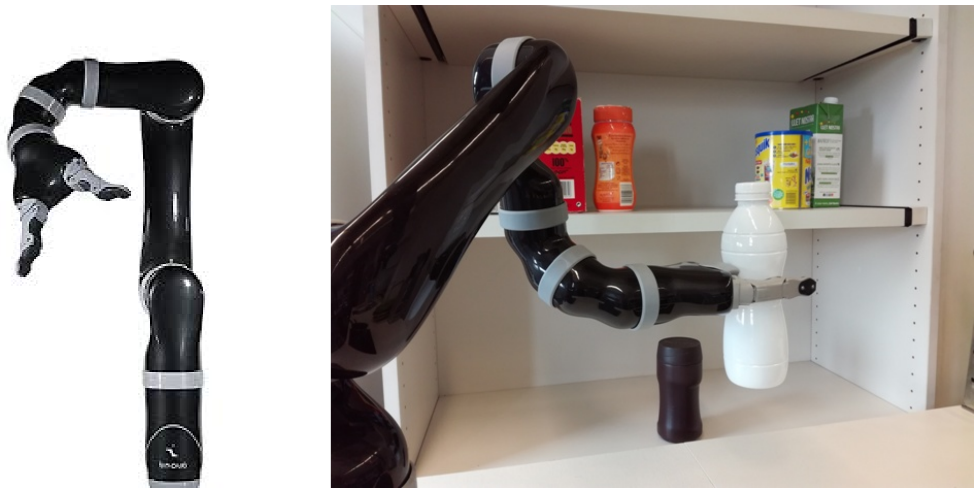


Figure 5.2: Kinova's MICO robotic arm

Architecture The Mico robot is a six-DOF robotic arm that is able to mimic the movement of the human shoulder, elbow and wrist. The MICO version is slightly smaller than the JACO arm and is made of reinforced plastic instead of carbon fiber. Still, it is a lightweight robot that can easily be integrated to various setups. Among other

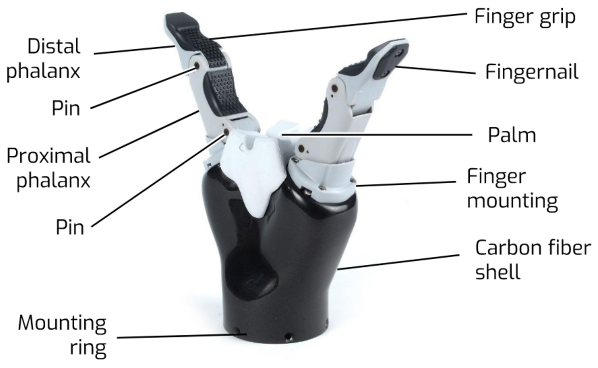


Figure 5.3: Kinova's KG-2 gripper

applications is for instance the arm being mounted on a wheelchair to provide additional assistance to disabled users. Its specifications are presented in table 5.2.

Control Various interfaces can be used to control the arm: a Software Development Kit that includes both high- and low-level APIs, a joystick, and ROS packages, which is the option of interest in our case. The provided tools offer various modes to control the robot: force/torque control or trajectory control, which can be position, velocity or admittance (i.e. force-reactive) control. For each mode it is possible to choose between Cartesian or angular control, i.e. to specify the desired translation and orientation or the position of each joint.

Hardware The Mico is connected to the central unit through USB 3.0. The controller is integrated in the base of the robot, which makes this latter quite compact. The arm embeds various sensors:

- in the base: supply voltage, temperature and accelerometer.
- in each actuator: optical position encoder, absolute position, torque, current, temperature and accelerometer.
- in each fingers: current, temperature, optical position encoder.

End-effector The MICO at the GRINS laboratory is equipped with Kinova's KG-2 gripper, which is a two-finger pliers-like gripper (figure 5.3). The fingers are flexible and under-actuated, which enables the robot to adapt to the shape of the object to be manipulated. The specifications of the gripper are presented in table 5.3.

Specification	**Value**
Payload (kg)	2.1
Reach (mm)	700
Max. linear speed (mm/s)	200
Robot weight (kg)	4.6

Table 5.2: Specifications of the Kinova's MICO^2 robotic arm

Specification	**Value**
Weight (kg)	0.556
Gripping force (N)	25
Object diameter range for cylindrical grip (mm)	55-100
Max. opening, at fingertip (mm)	175
Min. object diameter for object-on-the-ground pinch (mm)	8

Table 5.3: Specifications of the Kinova's KG-2 gripper

5.2 Software

The whole project has been developed using the ROS framework, with C++ and Python as main programming/scripting languages, and runs on Ubuntu 16.04 (Xenial). The following sections are more specifically directed to readers unfamiliar with this technology, and aim at briefly presenting ROS as well as the most relevant packages (MoveIt!) that have been worked with.

5.2.1 The Robot Operating System (ROS) framework

The acronym ROS stands for Robot Operating System [25]. This open-source "operating system" provides a set of frameworks for robot software development, including standard services such as hardware abstraction, low-level device control and message exchanges between processes[1]. It is a handy way to manage the various modules (packages) that will be part of the robot software. Many tools and libraries are also available to help developers write, build and run code across several machines. The framework is intended to be flexible in order to encourage code reuse and to simplify the development of complex yet robust robots.

ROS was first developed in 2007 by the Standford Artificial Intelligence Laboratory as a mere framework intended to be used in the STAIR project. The framework was then developed at Willow Garage, and from 2013, by the Open Source Robotics Foundation [2]. As of now, a total of eleven ROS distributions have been released. The AURORA setup relies on the Kinetic Kame release of May, 2016. The list of distributions as well as there end-of-life date may be found at[3]. Currently, only Unix-based platforms may be used.

The following paragraph is intended as a quick and thus highly non exhaustive introduction to ROS concepts. Detailed information may be found in abundance on the ROS Wiki website [4]. Some tutorials are also available at [5].

ROS is all about **Nodes** and **Topics**.

- **Nodes** are instances of executables. A node may be seen as the receptacle of one subprogram. Nodes are independent from one another and can run on different computers. Nodes may provide **Services** or **Actions**, which can be seen as functions at the system level. As an example, one can imagine a node A in charge of the global behaviour of a robot, which sends a request to a node B that provides an action to move the arm of the robot.

- **Topics** ensure communication. A node may publish **messages** on a topic, to broadcast information, and subscribe to other topics in order to receive information. Topics are a way to carry information and data in a "many-to-many" architecture (a topic may have several publishers and/or several subscribers). The **messages** are data structures defined in .msg files, that may contain various data types (booleans, floats, ints... or custom types).

At the beginning, ROS core process, the **master**, is launched. The master knows the list of the nodes and put them into contact. The nodes are then able to communicate directly through the different topics. ROS is based on a client-server distributed architecture, also meaning that the nodes may run on different machines, with a multi-master synchronization. The global parameters of the robot are defined within the **parameters server**, which consists in a memory shared by all nodes (to ensure parameters consistency).

[1] http://wiki.ros.org/ROS/Introduction
[2] http://www.ros.org/history/
[3] http://wiki.ros.org/Distributions
[4] http://wiki.ros.org/
[5] http://wiki.ros.org/ROS/Tutorials

All those elements (nodes, messages, services, as well as config and/or launch files) are gathered within **packages**, which enable modularity.

ROS comes with many tools. In particular, we will use RViz in order to simulate and visualize the system - which greatly eases the development and testing process. The kitchen environment can be recreated so that the simulation matches the reality, see figure 5.4.

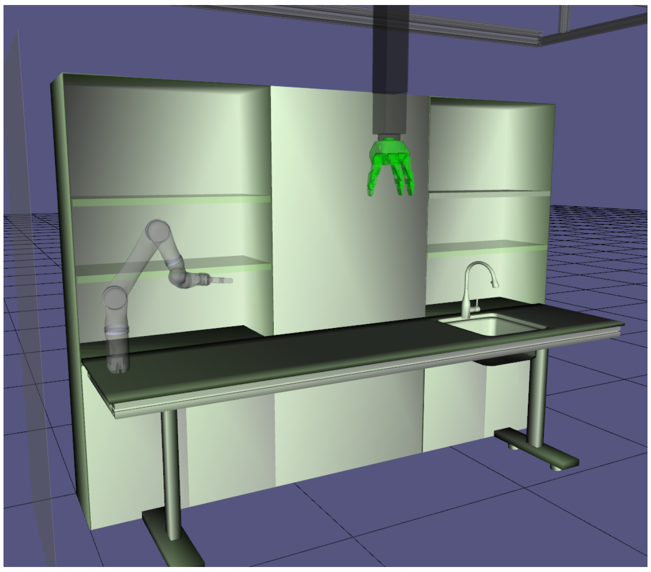


Figure 5.4: Recreation of the real environment in RViz

5.2.2 MoveIt!

5.2.2.1 Overview

The MoveIt! Motion Planning framework is a set of packages for ROS that was designed to provide tools for motion planning and collision management [5], targeting manipulation applications in industrial, commercial and research environments [6]. It was originally developed in 2011 by S. Chitta, I. Sucan, G. E. Jones, A. Pooley, S. Gedikli, and D. Hershberger at Willow Garage. The growth of the project is now supervised by a multi-background team of maintainers and core contributors, under the leadership of PickNik Consulting.[6] MoveIt! is entirely open-source and all the code is available on the dedicated github repository[7].

MoveIt! is written in C++ and includes some Python bindings. By default, it relies on ROS build and messaging systems. In order to provide high flexibility and ease code reuse, most of the functionalities are implemented as plugins, also called *capabilities*. The basic plugins that comes along MoveIt! are listed in table 5.4.

One of the challenge in developing robotic software is the heterogeneity of the platforms - with their specific architecture, morphology, sensors, actuators, communication channels etc. One of the advantage of MoveIt! is the fact it is robot-agnostic [6]. It can therefore be used with any robot once it is provided with the relevant configuration files. These configuration files are used to define the architecture of the robot and its control. They can be generated quite easily using a convenient GUI tool called the MoveIt! Setup Assistant.

[6]`https://moveit.ros.org/about/`
[7]`https://github.com/ros-planning/moveit`

Plugin	**Functionality**
MoveGroupCapability	base class to build plugins upon
KinematicsBase	implementing forward and IK solver
PlannerManager	base class for planners
PlanningRequestAdapter	adjusting motion planning request/result
MoveItControllerManager	interface to robot controllers
ControllerHandleAllocator	handlers for action based controllers
MoveItSensorManager	integrating sensor data to the planning process
CollisionPlugin	collision detection

Table 5.4: MoveIt! default capabilities (plugins)

As proof of its usefulness, official MoveIt! packages have been created for more than a hundred robots[8], including ones from well-known companies such as Universal Robots, Kinova, Kuka, Fanuc, ABB etc. In the particular case of the CAPDI, since it is a custom robot, the MoveIt! files were generated in a previous project using the Setup Assistant.

5.2.2.2 Architecture

Figure 5.5 represents the general, high-level architecture of the system. In the center in pale yellow is the `move_group` node, which constitutes the core of the framework. It is indeed employed to wrap up all the functionalities offered by MoveIt! through the use of ROS Services and Actions (depicted as red and blue arrows) or topic communication (green arrows). The scheme shows well how MoveIt! can be integrated in the general ROS environment:

- Blue boxes illustrate communication with the **robot**. The `move_group` node is able to manage movement execution with the robot by sending action requests to the controllers; by subscribing to topics, it has access to the sensor data.
- The pale green box represents the **ROS Parameter Server**, from which all the information about the architecture and morphology of the robot as well as other configuration parameters may be retrieved.
- Finally, the grey box encapsulates all the interfaces available for the **user**, which can be APIs or GUIs.

5.2.2.3 Motion planning

Motion planning in MoveIt! is performed using external motion planners that are interfaced through a plugin, which therefore allows the use of various tools from different libraries. As primary source, MoveIt! uses the planners from the open-source Open Motion Planning Library (OMPL). The OMPL (developed between others by Ioan Sucan, also author of MoveIt!) provides several state-of-the-art sampling-based motion planning algorithms [27], such as PRM, most RRT variants, KPIECE...

Through the plugin, the user can send a *motion plan request* specifying the desired location of the end-effector or joint position of a group. The user also has the possibility to declare constraints on position, orientation, joint value... that should be observed. If achievable, the `move_group` returns a trajectory (i.e. path, velocities and accelerations) to reach the target.

The whole motion planning pipeline combines the motion planner itself with *planning request adapters* that are used to pre-process the request or post-process the result. By default, MoveIt! includes several adapters that are summarized in table 5.5.

[8]`https://moveit.ros.org/robots/`

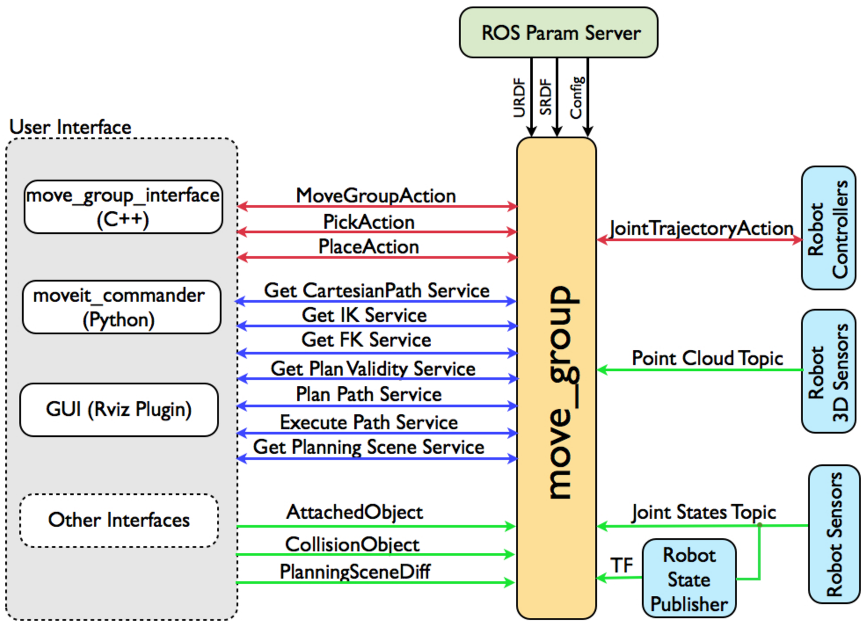


Figure 5.5: Architecture of the MoveIt! framework: Overview

Planning adapter	Functionality
FixStartStateBounds	fixes the joint start state to be within the limits specified in the URDF
FixWorkspaceBounds	specifies a default workspace for planning
FixStartStateCollision	samples a collision-free configuration near start state if needed
FixStartStatePathConstraints	uses a constraint-satisfying configuration near start state if needed
AddTimeParameterization	applies velocity and acceleration constraints to generated motion plans

Table 5.5: MoveIt! default planning adapters

5.2.2.4 Planning scene and collision checking

In order to perform accurate motion planning, it is crucial to have good knowledge and representation of the surrounding world as well as the state of the robot. In MoveIt!, all this information is stored in the *planning scene* which is maintained inside the `move_group` node by the *planning scene monitor*. The latter listens to the various topics on which are published the world geometry, the joint states and the sensor data if any (figure 5.6).

The world geometry is built as an occupancy map using user input and/or sensor information. The occupancy map is maintained through an octomap that can be passed to the Fast-Collision Library that MoveIt! uses to compute collision checking operations. MoveIt! can also work with collision objects built from meshes or primitive shapes (boxes, cylinders, etc.).

To reduce the computational burden that collision checking represents, it is possible to ignore some collision checks using the *Allowed Collision Matrix* that stores a binary value for pairs of bodies (objects or robot links). A value of 1 means that it is not necessary to check for the collision between the two corresponding bodies.

5.3 The AURORA setup

Chapter 3.1 presented some of the most relevant insights about MRS. With this in mind, let's focus on the setup used in this project. The first part of the work was indeed to build up the MRS from what was available at the start of the thesis. Subsection 5.3.1 presents the cornerstones of our MRS, in relation with the features and concepts presented in the previous section of this chapter. The following subsections linger over some specific details.

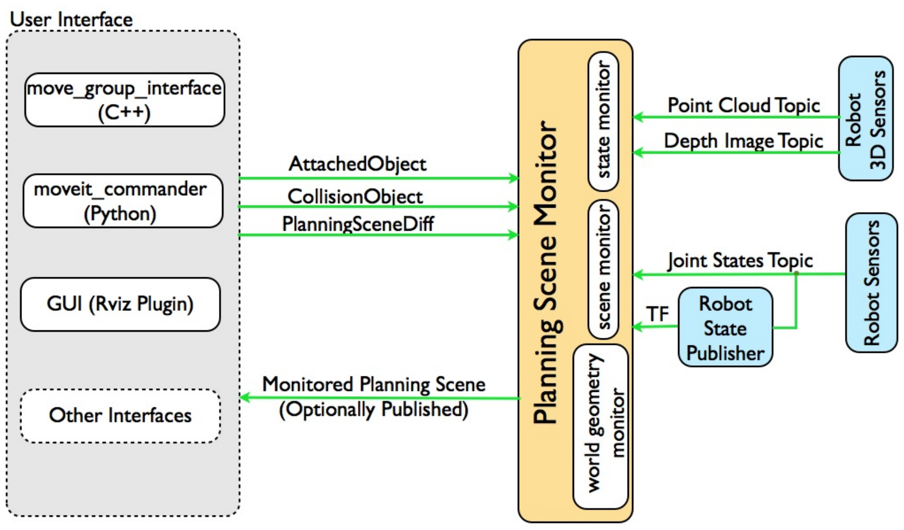


Figure 5.6: Architecture of the MoveIt! framework: the Planning Scene Monitor

5.3.1 General functioning

As presented in section 5.2, the system is developed using the ROS framework. The source code of the project is distributed in several packages:

- The `capdi_metapack` is a metapackage that gathers all the packages containing the description files of the CAPDI as well as the code for the hardware and for the control interface.
- The `mico_metapack` is a metapackage containing the packages developed by Kinova to control the MICO robotic manipulator arm.
- The `robotiq_metapack` is the metapackage containing all the code to use the Robotiq Gripper attached to the CAPDI.
- The `cobot_metapack` is the core metapackage that implements all the code related to the MRS. The packages in this metapackage depends (among others) on the previously mentioned packages.

An essential requirement of the system is to be able to plan trajectories that are collision-free, in order to preserve the integrity of the system itself, but also avoid damaging the objects or hurting the user. That is why we use the MoveIt! plugin, presented in the previous section. To be able to work with MoveIt! in a satisfactory way, each of the available robot is defined as a move group in the general description file of the robot.

Because we are using ROS, all the control schemes are implemented within different packages, mainly using action servers which will use MoveIt! features to plan and execute paths. Taking advantage of the abstraction layer provided by ROS, the work is limited to computing and sending the adequate trajectories; we shall not go deeper into the lower levels of control.

As explained in section 5.2, the communication between the different elements of the MRS happens through topics, on which are sent messages, service calls/responses and action requests/results. In the current state of the setup at the laboratory, both the MICO and the CAPDI are connected to the same computer. Therefore, it is only necessary to run one instance of ROS, and the whole system is managed by the same ROS Master. Note that it is possible to work with different machines, in a multi-master configuration (which of course complicates the setting up, in particular due to synchronization issues).

5.3.2 Environment management (Localization and mapping)

The states of the robots are maintained by ROS thanks to the data coming from the various sensors/encoders and the architecture information. From the joint state values that are retrieved, the end-effector position and orientation can indeed be easily computed.

The robots that are used are blind, meaning they do not have any sensors that allow them to get information about their environment. Instead, the state of the real world is maintained through the MoveIt! planning scene. The planning scene gathers information from three sources:

- the initial model (i.e. the kitchen, see section 1.2),
- the joint values of the robots that are published on the `joint_state` topic,
- the object recognition process.

Let's briefly discuss the last point. To perform object recognition, the kitchen is equipped with three RGB-D cameras, two Microsoft Kinects and one Asus Xtion. These two types of cameras are similar. They are cheap consumer-level sensors which provide both RGB images and a depth map of the scene. The data is then processed by the object recognition algorithm that performs several actions:

- plane segmentation, to identify the table and shelves,
- proper object recognition using a trained support vector machine (SVM),
- table partitioning as a grid to generate an occupancy map

The recognized objects (see figure 5.7 for some examples) are then published as collision objects in the planning scene. To this purpose, one node subscribes to the topics on which are published the recognition results and updates the planning scene accordingly.

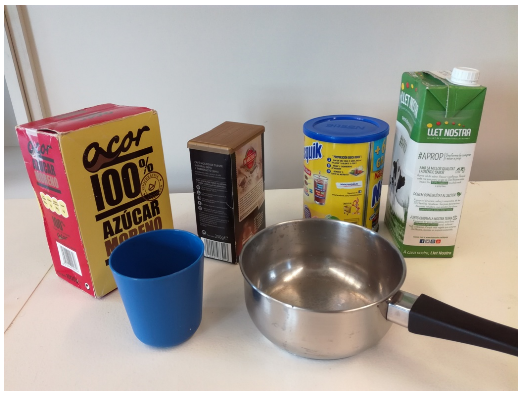


Figure 5.7: Examples of objects in the kitchen setup

5.3.3 Grasping

Grasping may appear as quite simple at first glance, since it is an action that most of us human beings perform naturally everyday. Without thinking much or thinking at all, we are able to seize all kinds of objects that exhibit great diversity in shape, texture, size and weight. In the considered MRS, the way the CAPDI robot grasps an object is not an issue, since its architecture inherently limits the number of available configurations (there is only vertical accessibility). The MICO robotic arm, however, is supposed to be able to mimic a human arm and is able to reach objects with a great variety of configurations.

Grasping has been studied for many years by the robotics community. When it comes to implementing what seems natural to us in a robot, it indeed turns out to be a much more complex problem. Until recently, robots were more known for their clumsiness than their manual dexterity. The progress made in the past few years in Artificial Intelligence and Machine Learning have changed the paradigm, and many applications have gained from using data-driven approaches such as Reinforcement Learning [18][20]. Since it is not the subject of this thesis, it was decided to only implement a quite simple grasping feature for the MICO robot. The approach is based on the fact that the robot is mainly manipulating box-shaped or cylindrical objects. The grasp pose always sets the gripper perpendicular to one of the lateral or the upper face of the object. Therefore, twelve grasp approaches are considered. Figure 5.8 shows schemes of the different approaches. The nomenclature is explained in table 5.6.

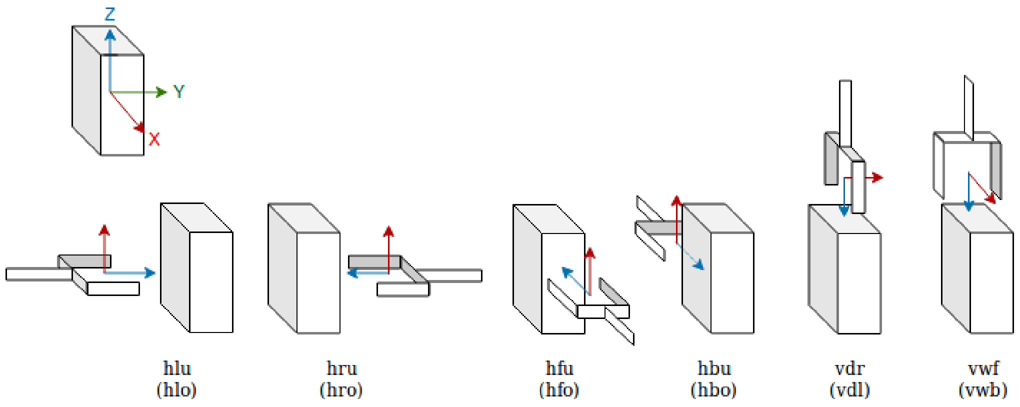


Figure 5.8: Grasp approaches for the MICO robot

The figure shows six of the grasp approaches defined for the MICO robot with their names. The blue and red arrows represent respectively the z- and x-axis of the reference frame attached to the end-effector. The other six configurations (names between parentheses) are obtained by a rotation of π radians around the z-axis.

#1	Gripper w.r.t surface	#2	Object grasped	#3	Gripper x-axis pointing
h	parallel to support surface	b	from **b**ack	o/u	**d**own/**u**p
		f	from **f**ace		
		l	from **l**eft		
		r	from **r**ight		
v	orthogonal to support surface	d	**d**epthwise	l	**l**eft
				r	**r**ight
		w	**w**idthwise	b	**b**ack
				f	**f**ace

Table 5.6: Grasp nomenclature

Each grasp configuration is identified by a three-letter name; each letter encodes information about the pose.

The grasping feature was implemented as a C++ library that is used by the action servers for the MICO. The main function returns a vector containing three poses that constitutes a complete pick (resp. place) approach: the pre-grasp, grasp and post-grasp (resp. pre-place, place and post-place) poses. These poses depends on the type of grasp. Apart from the twelve configurations mentioned earlier, the grasp can also be configured by passing two arguments:

- The `grasp_height` enables to change the height at which the object will be grasped. It takes values between -0.5 and 0.5, 0 being the default grasp at half the object.
- The `grasp_depth` specifies how deep is the grasping, i.e. for instance if the object is grasped using the complete fingers or only with the tip of the fingers.

5.3.4 Kinematics plugins

In order to perform motion planning, it is necessary to be able to compute the inverse kinematics (IK), i.e. for a given pose in the 3D space, find if possible adequate joint values for the robot.

Having a robust and reliable kinematics plugin is important for two reasons. The first one is that the application considered here implies that we mainly work by specifying pose and not joint targets. The second one is the fact that when specifying path constraints (whether orientation or position ones), MoveIt! reasons in the space of possible positions of the end-effector. Therefore, IK computations are performed for every sampled state.

The default Kinematics plugin integrated in MoveIt! uses the Kinematics and Dynamics Library (KDL) by Orocos[9]. The Inverse Kinematics solver provided by the KDL uses the common inverse Jacobian method with convergence algorithms based on Newton's method. The main issue is that in presence of joint limits, as it is the case for the MICO robotic arm, the KDL implementation turns out to have poor performances. As an alternative, it was therefore decided to use the **TRAC-IK plugin**[1], which performs much better (see table 5.7 for some results).

Chain	DOFs	Orocos' KDL		TRAC-IK	
		Solve rate	**Avg Time**	**Solve rate**	**Avg Time**
Baxter	7	60.98%	2.22 ms	99.44%	0.50 ms
Jaco2	6	26.22%	3.79 ms	99.61%	0.50 ms

Table 5.7: Comparison between Orocos' KDL and TRAC-IK solvers [10]

This solver runs two IK computations in parallel. The first one is an extension of the KDL's Newton-based convergence algorithm that takes into account joint limits. More precisely, it escapes local minima due to joint limits by performing random jumps. The second thread runs a sequential Quadradic Programming nonlinear optimization process which better manage joint limits. The plugin offers four modes to know which solution to return:

- **Speed** mode returns the first solution that is found (either by the first or the second method).
- **Distance** mode returns the solution that minimizes the sum of squared error from the seed.
- **Manipulation1** mode returns the solution that maximizes $\sqrt{det(J * J^T)}$, where J is the Jacobian matrix.
- **Manipulation2** mode returns the solution that minimizes $cond(J)$.

Apart from Speed (which is the default mode), all modes run for the full timeout, which is left at its default value, i.e. 5ms. Because the first solution that is returned is sometimes unsuitable for object manipulation, we discard the use of the Speed mode. Among the remaining options, all have been tried without noticeable difference, so for now, any mode seems suitable.

For the CAPDI robot however, the implemented kinematics plugin is based on KDL. Since the CAPDI kinematic chain is much simpler than the one of the MICO arm, working with KDL yields results that are good enough.

5.3.5 Planner for motion planning

Among the different planners that are available in the OMPL library, it was decided to work with the popular **RRT Connect** algorithm [19]. The RRT Connect is a variant of the RRT method. In the latter, at each iteration, the tree is extended by adding a new vertex that depends on a randomly-selected configuration q. It the configuration is not free, it is rejected. If it is, the algorithm selects the vertex in the tree q_{near} that is nearest to q and:

- if q is closer than an incremental distance ϵ, q is connected to q_{near} (situation: REACHED).

[9]http://www.orocos.org/kdl
[10]https://bitbucket.org/traclabs/trac_ik/src/master/

- if the distance to q is greater than ϵ, the tree is grown in direction of q by adding a vertex that satisfies the distance constraint (situation: ADVANCED).

The process can be seen in figure 5.9. Since the probability that a vertex is selected to perform the extension is proportional to the area of its Voronoi region, the RRT is biased to rapidly explore. The RRT Connect variant introduces two major changes. First, it relies on a new heuristic to perform the tree extension, the *Connect* heuristic. Basically, if the randomly-sampled configuration q is reachable, the tree is repeatedly extended until q is REACHED. Second, two trees are grown, using the initial configuration q_{init} and the goal configuration q_{goal} as seeds. The two trees are extended alternatively, and, after each extension step, the algorithm tries to connect both trees.

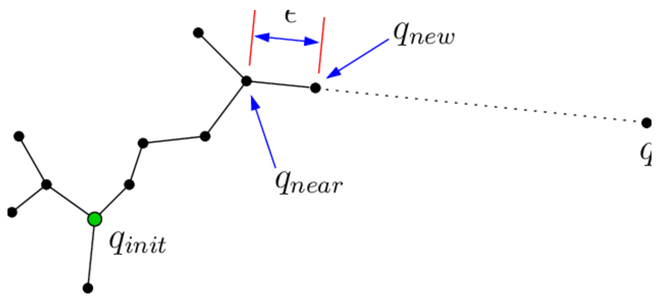


Figure 5.9: The *extend* step in the RRT algorithm [19]

5.4 MRTA solution implementation and integration within AURORA

Now that we have presented the different components and some of the technical choices for the multi-robot system that will be used, let's describe more in depth the actual MRTA system. To integrate it to the AURORA setup, the code was developed within the ROS framework, using as main language C++. This section will present the architecture of the MRTA system. Since this would be of little interest, we shall not cover every line of code but on the contrary, focus on the most interesting and relevant points.

5.4.1 General organization

The system was developed as a ROS metapackage, that gathers two packages:

- the `mrta_pia` package contains the core files for the MRTA system, i.e. the definition and implementation of the different classes as well as the auctioneer and bidder nodes.
- the `mrta_pia_msgs` package contains, as it can be inferred from its name, the ROS messages that will be used during the auction process to exchange information between the different parties.

The source and header files gathered in the `mrta_pia` package are classified within three folders:

- the `mrta_pia` folder contains the files in which are defined the C++ classes for each basic component of the system: the graph, the tasks, the bids, the STN and the robot representation. Theses classes all serve to define data structures with useful attributes and associated methods. This folder also contains the definition of the `auctioneer` and `bidder` classes that will be presented in more details in another subsection.
- the `utils` folder contains useful auxiliary functions. In particular, the file `stn_utils.cpp` implements functions to propagate and evaluate the consistency of a STN, using Floyd-Warshall algorithm.
- the `test` folder contains utility functions that were used when testing the system (see chapter 6 about results).

The content of each folder is compiled as a library, named mrta_pia_lib, mrta_pia_utils and mrta_pia_test respectively, using the command add_library() in the CMakeLists.txt file of the mrta_pia package. This means that none of these files contain a main function. The only files of the package that yield actual executables are the auctioneer.cpp and bidder.cpp, which initialize **ROS nodes** to run instances of the Auctioneer and Bidder classes. It is possible to run as many *bidder* nodes as necessary, and the solution is not limited to the Aurora setup: it is possible to define various bidders using the YAML configuration file (see as example figure 5.10).

```
mico:
    topic : "example_mico_joint_states"
    capabilities_list : [0,1,2,3,4,8]
    workspace_bounds : [0.3,1.4,0,0.7,0.6,1.5]
    location :
        x : 0.85
        y : 0.075
        z : 0.84
    velocity : 0.07
capdi:
    topic : "example_capdi_joint_states"
    capabilities_list : [0,1,2,4,8]
    workspace_bounds : [0.48,1.0,0.2,2.5,0.81,2.0]
    location :
        x : 0.60
        y : 1.32
        z : 1.53
    velocity : 0.12
```

Figure 5.10: Example of configuration file robots.yaml

Involving the user: In particular, it is possible to **include the user into the team** of agents, by treating them like a robot. To introduce the user variable in the system, a new bidder is added into the configuration file described earlier. The same characteristic fields can be filled; however, only the action types that *cannot* be performed by the robots are added to the user's capability list. This ensures that the person is not solicited if the robots are able to carry out the tasks. Of course, introducing a human in the loop implies higher uncertainty, since it is possible for the user to not achieve a task, which compromises the established schedule. This issue should be taken into account when dealing with failures and system dynamics at the execution time; consequently, it is out of the scope of the present work and will not be discussed in this report.

5.4.2 Auction process: auctioneer and bidders

To start running an auction, the auctioneer needs as **inputs** a vector of Task objects specifying the tasks to be assigned, and a precedence Graph which gives the ordering constraints. These two classes are explained in further details in section 5.4.4. In the current architecture, the auctioneer also needs to know the names of the participants. This vector of names is used to create two records (using C++ std::map<> containers) in which the auctioneer can note whether it received the bid and the finish time information from each bidder. The records are updated within the callback functions of the corresponding ros::Subscribers. Auction and batch counters are used to make sure the messages correspond to the current iteration (see next subsection). The bid record is cleared before each new round, and the finish time record before each new batch.

The auctioneer is in charge of managing the precedence graph, which is used to build the auction batches at each step. The auction is published and the auctioneer waits for all the offers. This choice of architecture is suitable for the Aurora setup in which the team of bidders is highly heterogeneous; therefore, if a bidder becomes suddenly unavailable, it is likely that the system has broken down and that the task allocation/execution will fail anyway.

Another possibility, though, would be for the auctioneer to ignore the number and names of bidders, and to wait during a certain time and only consider the bids received before the time out.

The bids are stored in a vector (C++ `std::vector` STL container). When all have been received, the auctioneer `sorts` the bids by ascending value. The first bid (i.e. the one that offers the smallest makespan) is declared winning bid, provided its value is not infinite. In the latter case, it means that no bidder can perform a task in the current auction batch, so the iteration is stopped and the task are categorized as unallocated. Once the batch has been entirely auctioned, the auctioneer receives and stores the finish times in a `vector`. This information is used to update the precedence graph.

All the code is encapsulated in the class `mrta_pia::Auctioneer`, split into various private member functions. The process is easily run by calling in the auctioneer node the public member function `run_auction()`. The definition of the class can be seen in listing A.1.

The counterpart of the `Auctioneer` class is the `Bidder` class. Among the main class attributes, the `robot_` is used to store the characteristics of the bidder (for instance its velocity and capabilities), and the `schedule_` to store the STN instance maintaining the agent's schedule. These two attributes are instances of the classes `mrta_pia::Robot` and `mrta_pia::STN` respectively (see section 5.4.4). As for the auctioneer, private member functions are used to implement the different steps of the bidding algorithm. They are all called in the body of the public function `run_auction()`. The contents of the header file defining the `Bidder` class can be seen in listing A.2.

When the process finishes, each bidder solves its STN, and publishes as **output** its sequence of assigned tasks, along with the associated start and finish times.

5.4.3 Communication

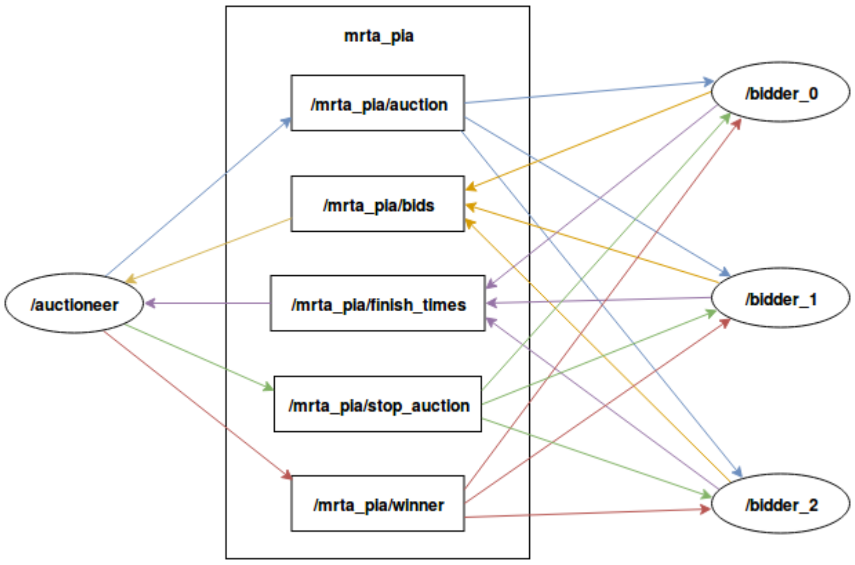


Figure 5.11: Communication topics between the auctioneer and bidder nodes

Since the system is developed using ROS framework, all the communication is done via topics, which can be seen in figure 5.11. Some custom messages have been created. Following a common way of organizing the ROS packages, they are grouped in the `mrta_pia_msgs` folder. Their definitions are presented in listings 5.1 to 5.4. They are rather self-explanatory. It might only be needed to explain the `auction`, `batch` and `round` fields of some of the messages: they are used to identify to which iteration of the process the message refers, so that old messages can be discarded.

Listing 5.1: Auction.msg

```
int32 id
int32 batch
int32 round
mrta_pia_msgs/Task[] tasks
float32[] pc_values
```

Listing 5.2: Bid.msg

```
int32 auction
int32 batch
int32 round
string bidder
int32 task_id
float64 value
```

Listing 5.3: FinishTimes.msg

```
int32 auction
int32 batch
string bidder
int32[] task_indices
float32[] finish_times
```

Listing 5.4: Task.msg

```
int32 TYPE_DUMMY = -1
int32 TYPE_PICKPLACE = 0
int32 TYPE_PICK = 1
int32 TYPE_PLACE = 2
int32 TYPE_POUR = 3
int32 TYPE_HOLD = 4
int32 TYPE_MIX = 5
int32 TYPE_OPEN = 6
int32 TYPE_CLOSE = 7
int32 TYPE_REMOVE = 8

int32 id
int32 type
float64 priority

geometry_msgs/Point location

string object
string surface

float64 duration
float64 earliest_start_time
float64 latest_start_time
```

5.4.4 Data structures

In this subsection will be presented the different data structures implemented to manage different aspects of the MRTA process. The corresponding header files are reported in appendix B.

5.4.4.1 Graph

The precedence graph is a data structure G = (V,E), where V is the set of vertices (i.e. tasks) and E the set of edges (i.e. constraints) between the vertices. The two most common structures to computationally represent a graph are *adjacency lists* and *adjacency matrices* [7]. An adjacency list is an array of |V| lists; the i^{th} list contains the vertices adjacent (i.e. connected) to vertex i in G. The second representation uses a |V|-square matrix M, with element (i,j) equal to 1 (or to the edge weight) is there exists an edge from vertex i to vertex j.

In our case, the graph is implemented as a C++ class, which is used to store the actual graph structure (i.e. the precedence constraints) as well as the vectors that represent the layers. A simple, functional structure is used to represent the nodes, which are identified by their id and two lists of incoming and outgoing edges. These edges are the actual representation of the ordering constraints.

The current implementation of the layer vectors `Ts, Tl`, etc. is a binary implementation in which the i^{th} element is equal to 1 if the i^{th} node is in the corresponding layer and equal to 0 otherwise. The class implements the necessary methods to manage these attributes, in particular the graph update, following the algorithm given in section 4.3.

5.4.4.2 Robot

As mentioned earlier when describing the `Bidder` class, a specific class was implemented to store the **characteristics** of the robot. These characteristics are retrieved from the configuration file (see figure 5.10) and are used when computing the bids. In particular, the velocity is used to calculate estimates of the travel time between two tasks. The workspace bounds and capability list are used within the member function `can_reach` and `can_perform`. This functions are used by the `Bidder` instance to check whether a task is *a priori* feasible. If it is not, the task is not further considered; in particular, the agent does not try to insert it into its schedule. This enables to save computation time but, most of all, to **take into account the team heterogeneity**.

5.4.4.3 STN

One of the most important data structures is the `STN` class, which is used to represent and manage the robots' schedules. Its main attributes are a `vector` of `Task` objects to store the scheduled tasks and two `vectors` of `vectors` of `floats` to represent the distance graph and its working copy. We use `vectors` of `vectors` to build a matrix-like structure, which is the one expected by the Floyd-Warshall algorithm used to check the consistency (see section 5.4.5).

Several member functions are defined and used to insert a task. For clarity, the process is split into several functions. One is used to expand the matrix with two new rows and two new columns, corresponding to the start time point and finish time point of the new task. When adding new columns/rows, the elements are set to infinity (except the diagonal ones which are zero). The values are updated calling two other functions; one to add the constraints related to the task duration, and one to add the travel time constraints between successors. Another function is used to deal with the particular case of the first task to be inserted.

As it can be seen in listing B.3, all these functions exist in two versions: one for the definitive insertion, and one for the temporary insertion done when evaluating the possibility to add a task to the schedule. The main difference is that in the definitive case, the distance graph is expanded using the STL method `std::insert`, to insert elements *at the specified position*. This function is however computationally expensive, since it requires many copies of the matrix. This is why, in the temporary insertion which is called much more often, the new task is systematically added *at the end* of the working copy of the distance graph, so that we can work with the `push_back` method which is more economical. Of course, this last method could be adopted for the final insertion as well, but it seemed clearer to respect the scheduled task order in the distance graph.

As in the other classes, most of the methods are `private`, the sequence of calls to these functions being encapsulated into one `public` class method. Among the other important functions, the `STN` class also implements the `tighten_schedule` function.

5.4.4.4 Task

The `Task` structure definition (see listing B.4) is similar to the ROS `Task` message, since they are used to store the same information: the task id, its type, its location, its temporal information as well as its optional priority and object/surface information. The definition can be found in listing B.4. The task type belongs to a given list that can be modified in function of the application; the only restriction is the dummy type that is mandatory.

5.4.5 Solving the STN

As seen previously, the schedule is consistent if the corresponding STN has no negative-weight cycle. A way to detect negative-weight cycles in a graph is to compute the shortest path between all pair of nodes: if the shortest distance from a node to itself becomes negative, then the graph contains a negative cycle.

By computing the shortest path from the node corresponding to the initial time instant to the node representing the start/finish time of a task, it is also possible to obtain the possible values for these time instants, i.e. one can solve the STN. Therefore, in order to check the consistency of the schedule as well as retrieving the solution values, we have to solve what is commonly known as the **All-Pair-Shortest-Path** (APSP) problem [8]: *Given a weighted directed graph $G = (V, E)$, find for every pair of vertices $i, j \in V$, the least-weight (or shortest) path from i to j, where the weight of a path is the sum of the weights of its constituent edges.*

There exists several methods to solve the APSP problem. Following Dechter's work in [10], we will use **Floyd-Warshall algorithm** to propagate and solve the STN. The Floyd-Warshall algorithm is a dynamic programming method. Its main idea is to compute the shortest path from i to j considering a growing set of intermediate vertices by which the path may pass. For instance, at step k, the path from i to j may only pass through vertices in the set {1,2,...,k}. The dynamic programming equation is:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & if k = 0 \\ min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & if k \geq 1 \end{cases} \tag{5.1}$$

where w_{ij} is the weight of the edge from i to j (set to ∞ if the edge does not exist).

The shortest path is given by the final value $d_{ij}^{(n)}$ where n is the number of vertices. The algorithm runs in $\mathcal{O}(V^3)$ time. It is given in Algorithm 5 along with the pseudo-code to check the consistency of the schedule in Algorithm 6.

Algorithm 5 Floyd-Warshall algorithm

```
1: for i = 1 to n do
2:     $d_{ii} \leftarrow 0$
3: end for
4: for i,j = 1 to n do
5:     $d_{ij} \leftarrow w_{ij}$
6: end for
7: for k: = 1 to n do
8:     for i,j = 1 to n do
9:         $d_{ij} \leftarrow min\{d_{ij}, d_{ik} + d_{kj}\}$
10:    end for
11: end for
```

ETSEIB

Algorithm 6 Check schedule consistency

```
function IS_CONSISTENT(distance graph $d$)
    $r \leftarrow floyd\_warshall(d)$
    $n \leftarrow size(r)$
    for i=1 to n do
        if $r_{ii} < 0$ then                    ▷ Look for negative cycles
            return $false, -1$
        end if
    end for
    $m \leftarrow |r_{n1}|$                     ▷ Makespan is latest finish time
    return $true, m$
end function
```

5.4.6 Computing the bid

At every round, each robot loops on the tasks in the current batch. For each task, it first uses the functions `can_perform()` and `can_reach()` presented earlier in subsection 5.4.4.2. If the task is *a priori* feasible by the agent, it tries to insert it in its current schedule at the different possible positions. Then, using the Floyd-Warshall algorithm presented in the precedent subsection, it is possible to determine if the new schedule is consistent and to retrieve the corresponding makespan value if it is the case. If this makespan value is the smallest encountered so far, the robot updates its best bid data. It also keeps in memory the corresponding insertion position, to avoid repeating the calculations if it ends up winning the task.

5.4.7 Tightening the schedule

Solving the STN is also used to compute the earliest finish times for each task: by propagating the distance graph using Floyd-Warshall algorithm, it is possible to retrieve the smallest possible value for the finish time points of the different tasks. These values are returned to the auctioneer and are also used to update the distance graph components.

Summary

The present chapter has presented some elements of the AURORA setup related to the implementation. In particular, the available robots are a Kinova MICO (a 6-DOF robotic arm) and a custom 3-axis cartesian robot, the CAPDI; the main software tools are ROS with the MoveIt! package.

It has also explained the organisation of the specific MRTA algorithm implementation within the ROS framework. The auctioneer and bidder processes are defined in ROS nodes that instantiate objects of the C++ classes `Auctioneer` and `Bidder`. For clarity, various class methods are used to implement the different steps. Theses two classes and their functions use custom C++ classes or data structures that have been implemented to specifically represent tasks, precedence graphs, schedules or robots.

Chapter 6

Evaluation and results

Now that the design of the solution as well as its implementation have been detailed, this chapter will present the results obtained from different tests that were set up to evaluate the MRTA system. All experiments were run on a octa-core machine equipped with an Intel Core i7-4770K (3.50GHz) processor and 16GB of RAM running Ubuntu 16.04 LTS (Xenial).

6.1 Validation using Solomon's VRP database

The research done for this project has not permitted to find an evaluation database that would be specific for MRTA problems with manipulators; instead, most of the available databases are instances of the vehicle-routing problem. Still, it was decided as a first test to evaluate the implemented solution using the popular Solomon's database [26] for VRP.

The reasoning that supports this choice is the following: since the MRTA solution that was adopted is supposed to be quite general, the ROS implementation should not be limited to our problem within the AURORA setup. On the contrary, using the database is a good way to validate the different features of the system presented in chapters 4 and 5, as well as its general functioning, scalability and portability. In particular, the evaluation aims at ensuring that, besides mere task assignment, the system can **properly take into account time constraints (within the form of time windows or precedence constraints) and heterogeneity**.

The following subsections will therefore present this VRP database and a sample of results obtained with it.

6.1.1 Solomon's database

The Solomon's database is constituted of a set of several instances of a vehicle-routing problem, in which a fleet of vehicles has to serve a given number of customers. The files associated to the database specify the number of vehicles and their capacity (all have the same), as well as the list of customers to be served. For each customer, it is indicated:

- the customer id
- the x and y coordinates
- the demand
- the ready time and the due date
- the service duration

In our evaluation, we shall not consider the capacity of the vehicles, since the system is intended for single-robot tasks, nor the demand, since in this first part task prioritization will not be considered. The customer information is used to build the tasks, in the following way:

- the customer id gives the task id
- the x and y coordinates give the task location
- the ready time and due date are used to define the task earliest and latest start time (time window constraints)
- the service duration specifies the task duration

The several instances of the Solomon's database[1] are classified within six general problems: R1, R2, C1, C2, RC1 and RC2. R means the task locations are randomly distributed, C that they are clustered, and RC is a mixed situation. Problems identified with a 1 have short scheduling time horizon, while problems identified with a 2 have large scheduling time horizon.

Within one type of problem (R, C or RC) the task coordinates are the same; the several instances differ in the density and width of the time windows.

6.1.2 First test: simple task assignment

In this project, we shall not perform an exhaustive analysis of the implemented solution over the whole Solomon's dataset. The focus is set on the R type instances. The first test aims at testing the basic functioning of the system, and consists in a simple task assignment in which there is absolutely no constraint:

- no capability constraints (all robots can do all the tasks),
- no time-window constraints: all tasks are available since the beginning and can be performed at anytime,
- no precedence constraints: tasks can be performed in any order.

Figure 6.1 shows the resulting route for one instance of the R problems with sixteen tasks (which coordinates are given in table 6.1) and four identical robots. All the robots are initially located at the same point, the base, represented by a red asterisk. One can see that, in absence of all constraints, the robots naturally go to the nearest available task, which is the logical way to reduce the makespan. It is also possible to notice that the task distribution between the different robots is quite fair, despite the robots having no knowledge of the other schedules. This is a characteristic of the implemented auction-based solution with scheduled management: if a robot already has a high number of tasks to perform, its bid is likely to be higher than an other robot that would be idle.

The resulting schedule for the same problem with only one robot can be seen in figure 6.2. When working with only one agent, the total time necessary to accomplish all the tasks is of 302.3 time units (t.u.), i.e. three times more than when using four robots.

Additional remark: The chosen architecture is **flexible** and allows to work with a varied number of bidders: it is only necessary to specify the characteristics of the robots within the `.yaml` configuration file and to give the auctioneer the list of bidders that will actually take part in the auction. The system could therefore be tested with fifty tasks and ten bidders. In a no-constraint scenario, the auction is successfully completed within a few seconds. Working with many bidders requires, however, to explicitly **slow down** the process of message publication by inserting a delay of around 0.1 second. If not slowed down, the ROS message publication gets slightly chaotic and some messages are lost, despite the communication being *latched*. This is one limitation of the system.

[1]they can be found at `http://web.cba.neu.edu/~msolomon/problems.htm`

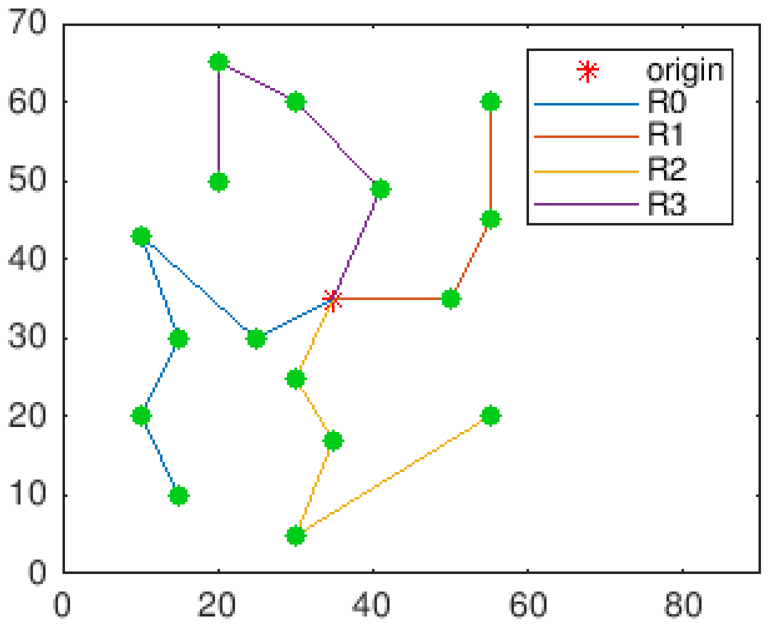


Figure 6.1: Solution to one R instance with 16 tasks, 4 robots, unconstrained (makespan: 94.9 t.u.)

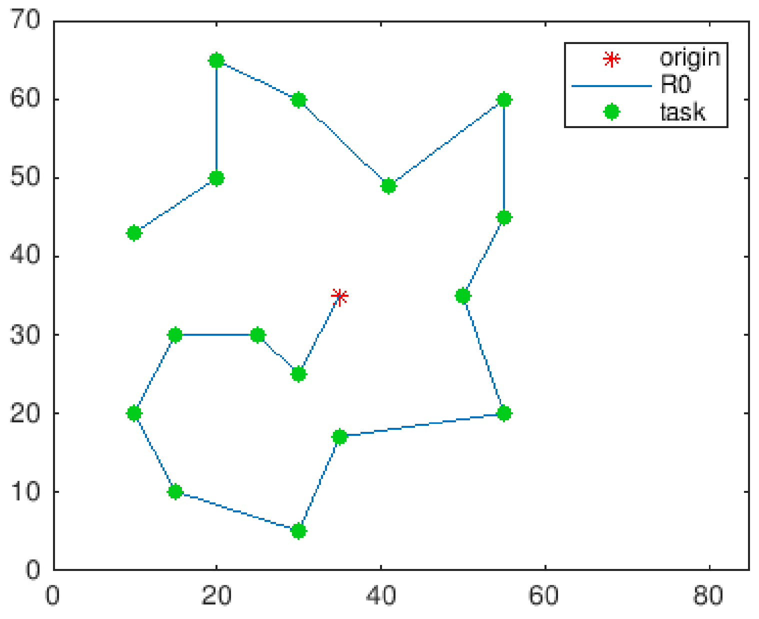


Figure 6.2: Solution to one R instance with 16 tasks, 1 robot, unconstrained (makespan: 302.3 t.u.)

6.1.3 Second test: task assignment with time window constraints

As a second test, the MRTA system was launched with the same instance, but this time taking into account time-window constraints that restrict the start time of a task. This should modify the obtained routes, since some tasks can only be started within a given time period. Not all tasks are constrained, and the time windows vary between the tasks.

Figure 6.3 shows the solution to the problem described above, with the same tasks and robots than in figure 6.1. When comparing the two, the difference between the routes is striking, and, in the second experiments, the

Task id	Coordinates (x,y)	Task id	Coordinates (x,y)
0	(41,49)	8	(55,60)
1	(35,17)	9	(30,60)
2	(55,45)	10	(20,65)
3	(55,20)	11	(50,35)
4	(15,30)	12	(30,25)
5	(25,30)	13	(15,10)
6	(20,50)	14	(30,5)
7	(10,43)	15	(10,20)

Table 6.1: Task coordinates for the 16-task R instance

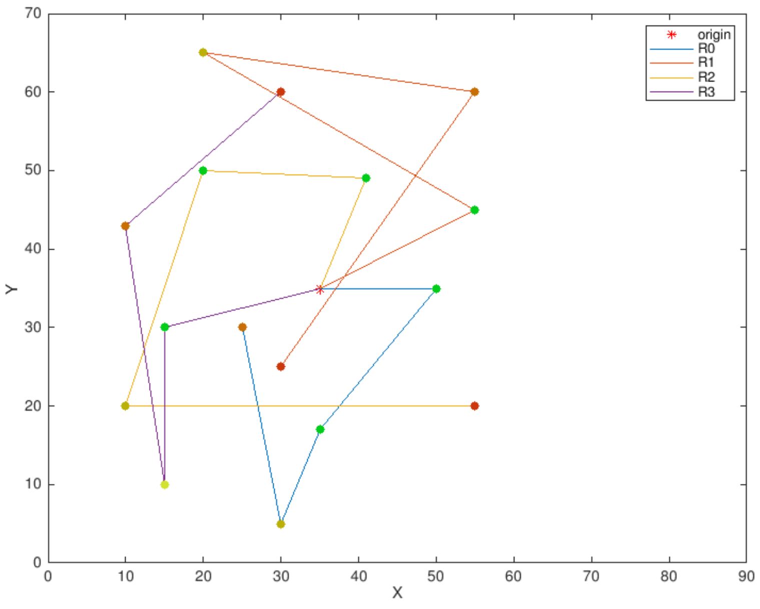


Figure 6.3: Solution to one R instance with 16 tasks, 4 robots, constrained start time

results seem much more disorganised. To understand better the results, one has to take into account the fact that the restrictions implicitly define a loose ordering between the tasks.

It is not easy to schematically represent the time windows information. In figure 6.3, each task is labeled with a color to translate the time-window constraint. The green tasks are tasks that are not constrained, meaning they are available since the beginning. The rest of colors represent intervals in which are included the time-window constraints. The earliest time windows are depicted in yellow, the latest in dark red.

One can therefore see that every robot first performs tasks that are labeled in green, which are available at start. The tasks colored in yellow, orange and red are performed after, since they may not be start earlier than some time instant. That is why the task at coordinates (30, 25), labeled in red, is one of the last task to be complete, despite being close to the origin. In comparison, in the first experiment, this task was one of the firsts to be completed.

To apprehend better the task assignment process, the routes are represented again in the figure 6.4 which shows in series of subfigures the progression as new tasks become available.

As an **additional remark**, it can be highlighted that the results of course depend on the robots characteristics, for instance the speed: if the robots are faster, they can easily reach farther tasks. On the contrary, if they are slower, they will be less likely to cover great distances since the bids for farther tasks will be much higher. This leads to a second remark: here, the optimization objective is to minimize the makespan, i.e. the total time used

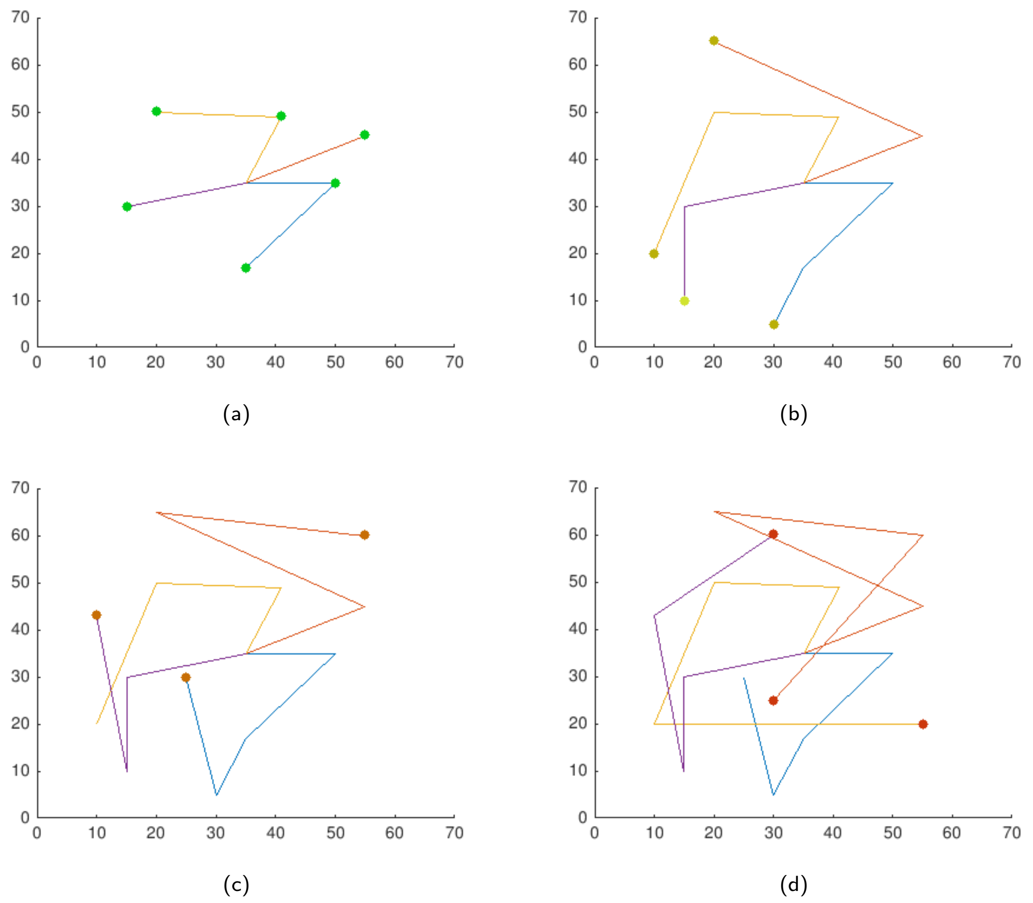


Figure 6.4: Task assignment progression with time window constraints

to complete the tasks. In Solomon's dataset, the optimal results that are given are obtained when minimizing the covered distance. The latter objective makes more sense in VRP, but, as mentioned earlier, in our application, the aim is to serve the user as fast as possible. If we had chosen to minimize the distance, the obtained routes would likely be more clustered.

6.1.4 Third test: task assignment with precedence constraints

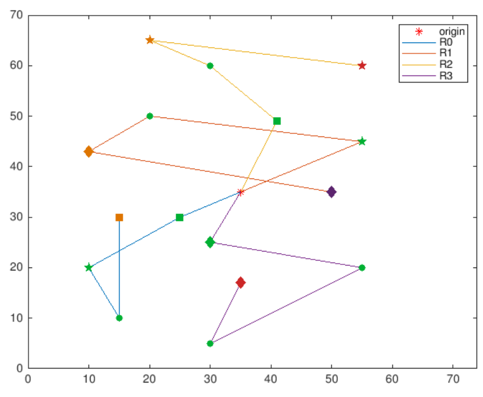
For the third test, the same problem instance was used, this time considering explicit precedence constraints between the tasks.

For n tasks, $n/2$ precedence constraints are defined, so that the obtained graph is sparse. To build a constraint (edge), two tasks (nodes) are randomly selected. To be added to the graph however, the constraint should verify the following conditions:

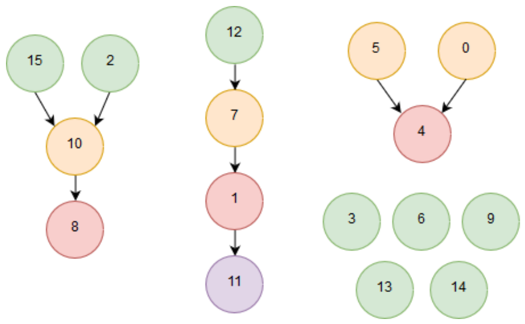
1. the edge between the selected nodes should not exist already,
2. the edge should not create a cycle in the graph, which must be *directed acyclic*,
3. the constraint should be "doable" with respect to the time windows, i.e. for task i to precede task j one should have:

$$est_i + dur_i <= lst_j$$

where est (resp; lst) is the task earliest (resp. latest) start time and dur denotes its duration.

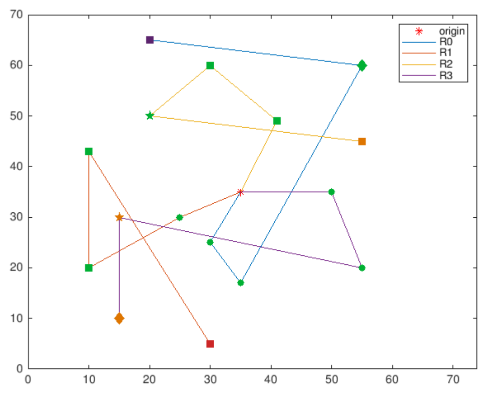


(a) Solution

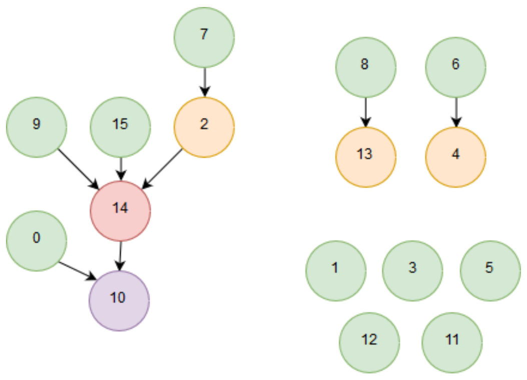


(b) Precedence graph

Figure 6.5: R instance with 16 tasks, 4 robots, precedence constraints - Case 1 (makespan: 117.2 t.u.)



(a) Solution



(b) Precedence graph

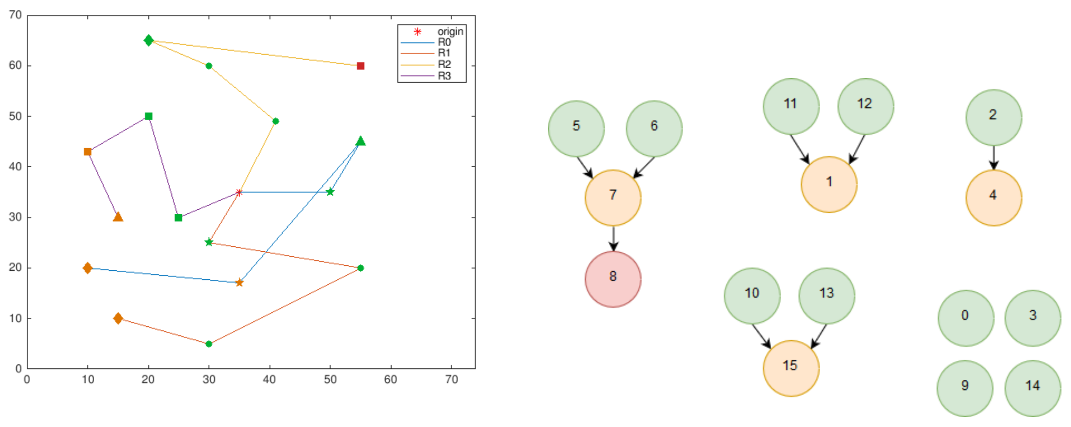
Figure 6.6: R instance with 16 tasks, 4 robots, precedence constraints - Case 2 (makespan: 131.8 t.u.)

Figures 6.5 to 6.8 show the routes obtained for some sample instances of the problem. To represent the precedence constraints, we use different symbols as well as different colors. Each symbol denotes a precedence tree, and the color gradation green⟶orange⟶red⟶purple indicates the task level within the tree. For instance:

- in figure 6.5, the tasks at (41,49) and (25,30), represented by green squares, are predecessors of the task at (15,30), represented by an orange square.
- in figure 6.7, the task at (20,65), represented by a green diamond, is a predecessor of the tasks at (10,20) and (15,10), represented by orange diamonds.

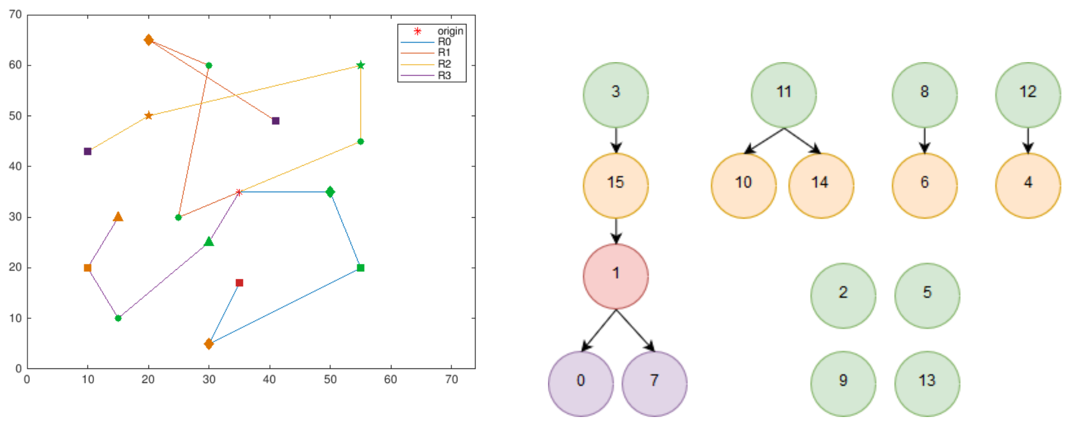
As it can be seen, the results depend a lot on the task ordering, which was randomly generated for each trial. From the graph, it is not possible to check exactly that the precedence constraints are satisfied, but one can see that in general, the tasks from the first level (tasks that are *free*) are executed before the tasks from the subsequent layers. Also, the more complex the precedence graph, the more complex the schedule, and the greater the makespan.

As an **additional remark**, one can see that the system allows to respect the precedence constraints, although the robots have no knowledge about these restrictions or about the other agents' schedules. About the process performances: the mean execution time over thirty trials for the auction process to complete is of **520 ms** (with



(a) Solution (b) Precedence graph

Figure 6.7: R instance with 16 tasks, 4 robots, precedence constraints - Case 3 (makespan: 106.5 t.u.)



(a) Solution (b) Precedence graph

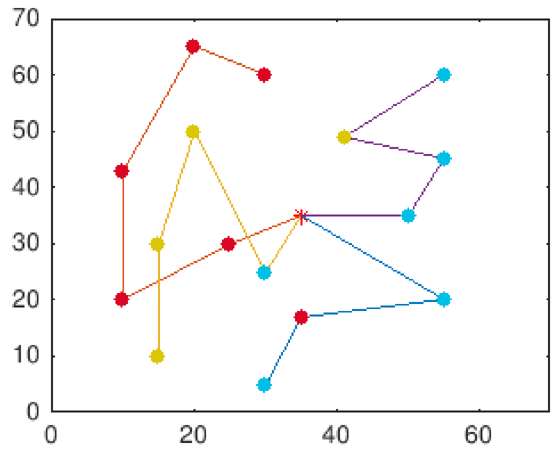
Figure 6.8: R instance with 16 tasks, 4 robots, precedence constraints - Case 4 (makespan: 108.2 t.u.)

a standard deviation of 70 ms). In average, it could successfully assign 15 tasks over the 16 to execute (with a minimum of 13). This last point can be accounted for by the fact that the precedence constraints are randomly picked up; the feasibility is checked for one particular constraint but not for the whole problem, hence the result.

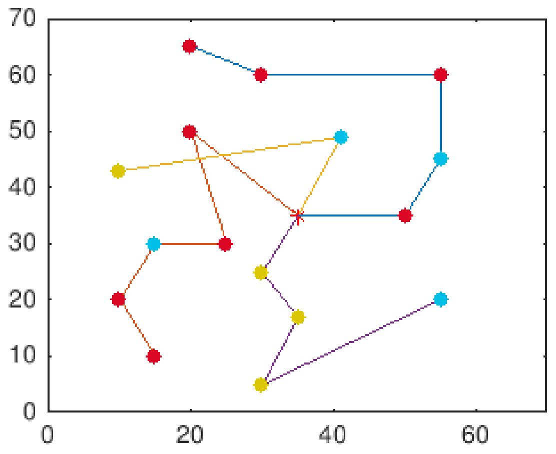
6.1.5 Fourth test: task assignment with task type

The last experiment of that first part of the evaluation was done with the same set of randomly distributed tasks. Its goal is to assess the correct management of heterogeneity during the task allocation process. In this test, each task is characterized by a type which is denoted by one of the following values: 0, 1 or 2. The types are randomly assigned for each instance of the problem; the probability of assigning one type or another is uniform and equal to one third. Using this setup enables to test the MRTA process with an heterogeneous robot team, in which some members may perform only certain types of tasks. In our test, robots R0 and R1 can perform tasks of type 0 and 1, and robot R2 and R3 can perform tasks of type 0 and 2.

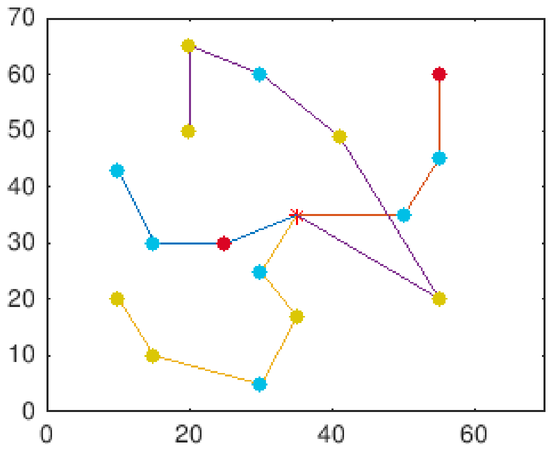
Figure 6.9 shows the resulting task assignment for three trials of the problem with task types. Each task is represented by a dot which color indicates the type. As one can see, the routes for R0 and R1 only contain blue and red dots (type 0 and type 1 tasks), while the routes for R2 and R3 only contain blue and yellow dots (type 0 and type 1 tasks). For these three trials, the makespan is quite low, and the resulting scheme is quite harmonious.



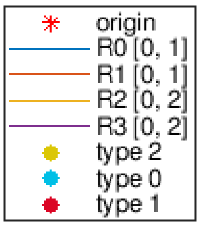
(a) Case 1 (makespan: 108.4 t.u.)



(b) Case 2 (makespan: 101.6 t.u.)



(c) Case 3 (makespan: 116.0 t.u.)



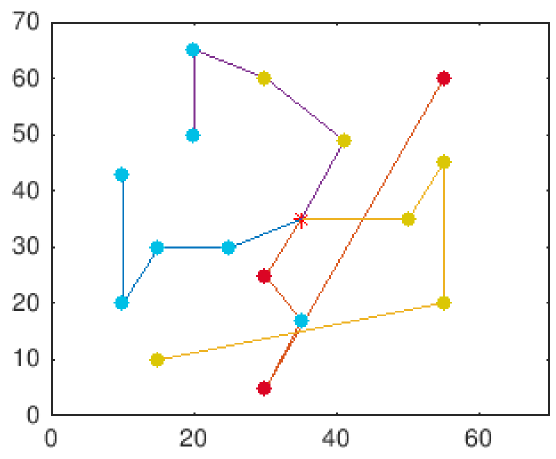
(d) Legend

Figure 6.9: Solution to three R instances with 16 tasks, 4 robots, heterogeneous tasks and team

Figure 6.10 shows two other plots of the same problem that illustrate one weakness of the algorithm, which is the counterpart of the rigidity which is necessary to ensure ordering constraints are and remain satisfied. Here, although the tasks are correctly assigned regarding their types and the capabilities of the robots, the solutions are sub-optimal. For instance, in Case 4 (figure 6.10a), it would have been better to schedule task 1 (30,5) before task 14 (33,17) in robot R1's route. This would (slightly) reduce the makespan to 101.6 t.u. (though the covered distance would be increased). In Case 5 (figure 6.10b), the optimal solution requires to assign task 14 (15,10) to R3 instead of R0. This way, the makespan would be reduced by 10.5%, from 136.6 t.u. to 122.2 t.u..

This sub-optimal behaviour is the result of two facts:

- first, in the current implementation, when the auctioneer receives two bids that are equal, the winner is the one which offer has been received first;
- second, robots can only modify their schedule by inserting new tasks at appropriate positions, which is mandatory to preserve precedence; the order of assigned tasks may not be changed, neither can tasks be reassigned to another robot.


(a) Case 4 (makespan: 102.7 t.u.)

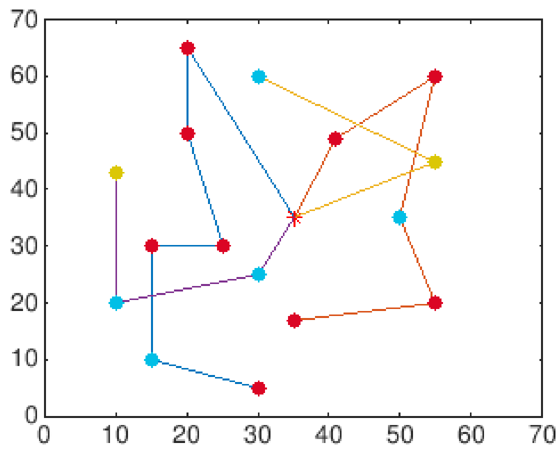

(b) Case 5 (makespan: 136.6 t.u.)

Figure 6.10: Solution to two R instances with 16 tasks, 4 robots, heterogeneous tasks and team

Because of this, in case 4 example, R1 bids and obtains tasks 12, 1 and 14 which are scheduled in this order. At the end, its bid for task 9 is the best, since the other agent able to perform that kind of task is R0 which is already occupied. R1 therefore wins task 9 and insert it at the end of its schedule, which is the best position; it cannot however reschedule task 14 before task 1. In case 5, R0 wins first tasks 5 and 4, while R3 wins tasks 12 and 15. Because it is ready first, R0 can win task 14. It wins tasks 10 and 6 at the end, but they are inserted at the beginning of the schedule which is the best position. This delays the execution of task 14 which could have been executed earlier by R3; but it is not possible to reassign the task to another bidder.

Over thirty repetitions of the same problem, only one case was not entirely feasible, in which the MRTA system could only assign 15 of the 16 tasks. In all the other trials, a solution to perform the whole set of tasks has been encountered. In average, the total time necessary to perform the sixteen tasks is of 111.1 time units, with a standard deviation of 9.3 t.u., the minimum being 99.6 t.u. and the maximum 136.6 t.u.. The auction process needed in average **268 ms** to be completed (with a standard deviation of 29 ms).

6.2 Evaluation on Aurora scenarios

Using the Solomon's VRP database enabled to evaluate the functioning of the MRTA system and gave an insight of its performances. The objective of this second and last series of experiments is to come back to the main motivation behind this project and to test the system on plausible Aurora scenarios that involve precedence constraints. The bidders are robotic arms like the CAPDI or the MICO presented in chapter 5, and the general setting is similar to the kitchen presented in figures 1.1 and 5.4.

Two scenarios involving different teams and different tasks to complete are proposed and solved. In particular, the first scenario involves only robots, while in the second one the team includes the user. They are presented, as well as the corresponding output schedules, in the remaining of this section. As mentioned when presenting the adopted MRTA solution, it is possible to prioritize the tasks. The tests were run with various prioritization policies, which are explained in the next subsection.

6.2.1 Task prioritization

One can think about several methods to determine the priority of a task. Here, three policies are considered:

- **no priority**: no priority is applied which is equivalent to the tasks having the same priority. In practice, all the tasks have priority 0, but one could use any other scalar.
- **c-policy**: the *c* in the name of this policy stands for "children". In this policy, the priority is set to the number of children a task has in the precedence graph. All layers are taken into account and not only the direct children. Tasks with a lot of successors therefore have a high priority. The idea is to bias the batch selection so that these tasks are performed earlier, in order to free the set of tasks that depends on them.
- **α-policy**: the last policy, here called α-policy, is defined according to McIntire et al.'s proposal in [22]. They suggest the following heuristic:

$$p_\alpha(t) = (1-\alpha)\ L(t) + \alpha\ U(t), \quad 0 \leq \alpha \leq 1 \tag{6.1}$$

$$\text{with} \begin{cases} L(t) &= \mathrm{dur}(t) + \max\limits_{t' \in \text{children}(t)} \left(L\left(t'\right)\right) \\ U(t) &= \mathrm{dur}(t) + \max\limits_{t' \in \text{children}(t)} \left(d\left(t,t'\right) + U\left(t'\right)\right) \end{cases} \tag{6.2}$$

  where $\mathrm{dur}(t)$ denotes the duration of task t and $d\left(t,t'\right)$ the travel time from task t to task t'. $L(t)$ is the minimal time necessary to complete task t and the whole sequence of its successors. $U(t)$ is the total time a single robot would need to execute the sequence. The two terms are pondered by α, which can be coarsely interpreted as the weight of travel time in the priority computation. As the $c-policy$, this rule biases the batch selection in favor of the tasks that have a lot of successors; however, it also takes into account the time dimension. As a result, a task with one single child that takes time to complete may have a higher priority than a task with several low-duration successors. In our scenarios, the travel time between most tasks is negligible in comparison with the task duration. Consequently, $U(t) \simeq L(t)$ for almost all t, and the choice of α does not impact the prioritization. The resulting schedules presented afterwards were calculated with $\alpha = 0.5$, but would be the same for any value of α within the interval [0,1].

6.2.2 First scenario: robots only

6.2.2.1 Scope

The first scenario conditions are actually slightly different from the real Aurora setup. In particular, we consider the case where the kitchen is equipped with one CAPDI robot and three, instead of one, MICO arms. This enables to test more demanding scenarios, as well as to evaluate the possibility of parallel task execution, which is rather limited in the real Aurora setup. In the rest of this subsection, the three MICOs will be identified as "M0", "M1" and "M2". M0 is the arm that corresponds to the current position of the MICO in the real setup, at the left edge of the table. M1 is the arm that has been placed closer to the middle, and M2 is the robot located on the right edge of the table. This arrangement, which can be seen in figure 6.11, enables to reach both cupboards to store objects, and to create parallelism.

As mentioned in introduction, the problem is solved for different task prioritization policies. In order to evaluate the interest of having two MICOs on the left side of the table, the MRTA is carried out on the same scenario but without M1.

6.2.2.2 Tasks

The following situation is considered: a box containing various food items has to be unpacked, and each item be stored to its place (one can imagine the user has just gone shopping or had their purchase delivered at home). In the

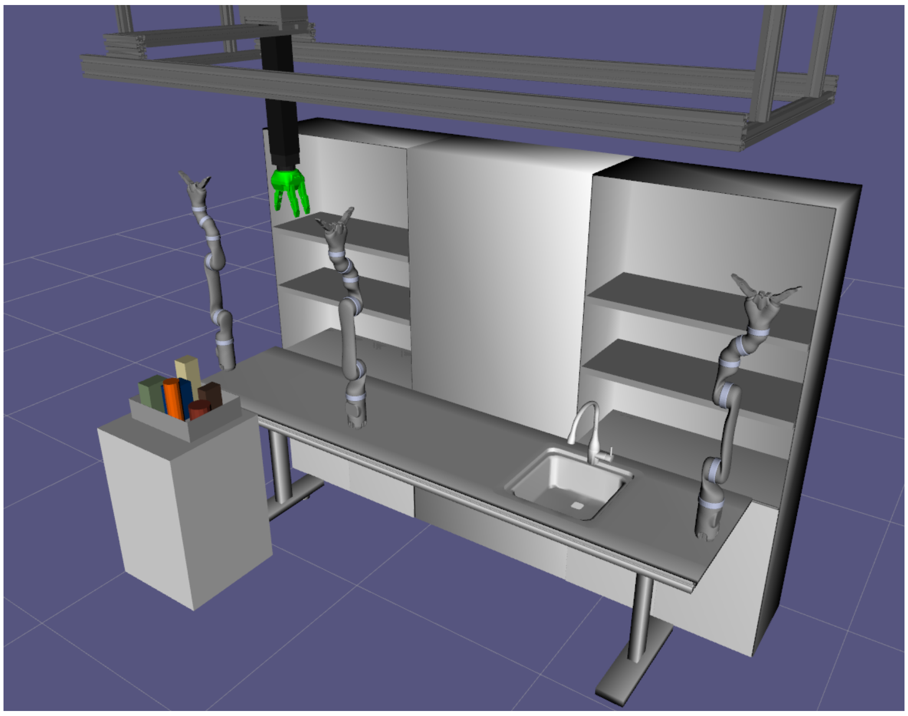


Figure 6.11: Evaluation scenario 2 - environment

studied scenario, a small piece of furniture is added to the Aurora kitchen environment to place the box. The robots have to store six items:

- sugar (green box in fig. 6.11) and coffee (brown box), to be stored on the shelves of the left cupboard;
- a tomato sauce can (in red) and pasta (in blue), to be placed on the shelves of the right cupboard;
- a carton of milk (in pale yellow) and a bottle of juice (in orange), that have to be stored in the fridge (imagining for instance a fridge with an automatic door). They should be placed in a position reachable by the user, on the middle of the table, between M1 and the sink.

Some of the tasks can be performed by several agents, while others can only be achieved by one of the team member, depending on the agent's capabilities and reachable workspace. The precedence graph is represented in figure 6.12. Each task is represented by a circle divided in two: in the upper half is given the action type, and in the lower part the associated object. As one can see, the global precedence graph is composed of six disconnected branches which can be performed in parallel. The bold number at the left of each circle indicates the task index. Table 6.2 gives more details about each task, in particular the locations associated to each task. One can observe that all the tasks of type `REMOVE` (task type 8) have the same location of arrival: this is because it is only an indication of where the object should be left; the actual locations can be computed at execution by searching for a free point around this location. In this example, only precedence constraints are considered, so the time window is the same for all the tasks ($[0,\infty]$).

6.2.2.3 Output schedule

The output schedules obtained for the different variants of the first scenario are presented in tables 6.3 and 6.4. The first column of each table gives the task ids, classified in ascending order with respect to their start times. For each task, its start and finish times are specified, along with the agent to which it has been assigned. The makespan is indicated in bold; for all cases, it corresponds to the finish time of the last task to be started. For this example, the no-priority and the c-policy yielded the exact same output; that is why only the latter is reported.

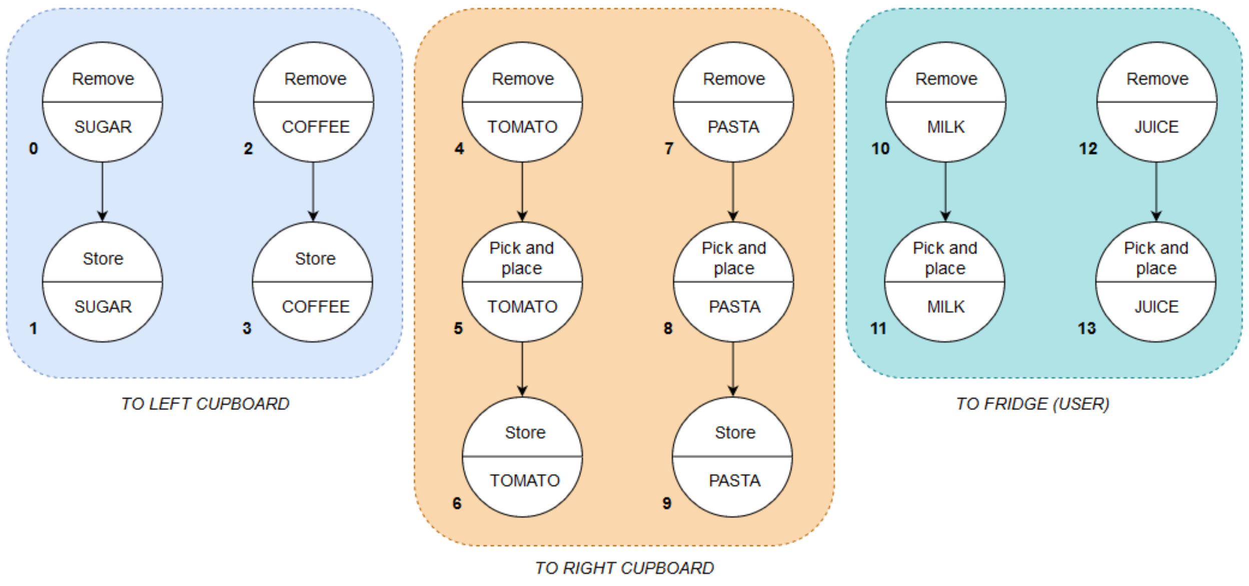


Figure 6.12: Evaluation scenario 1 - task precedence graph

Id	Type	Duration(s)	Position 1	Position 2	Object	Surface	c-priority	α-priority
0	8	30	(1.32 0.12 0.89)	(0.85 0.45 0.89)	sugar	table	1	60
1	0	30	(0.85 0.45 0.89)	(0.31 0.32 0.85)	sugar	lower left shelf	0	30
2	8	30	(1.18 0.33 0.87)	(0.85 0.45 0.87)	coffee	table	1	60
3	0	30	(0.85 0.45 0.87)	(0.31 0.32 1.20)	coffee	middle left shelf	0	30
4	8	30	(1.29 0.39 0.86)	(0.85 0.45 0.86)	tomato	table	2	105
5	0	45	(0.85 0.45 0.86)	(0.65 2.37 0.86)	tomato	table	1	75
6	0	30	(0.65 2.37 0.86)	(0.31 2.48 1.20)	tomato	middle left shelf	0	30
7	8	30	(1.28 0.23 0.89)	(0.85 0.45 0.89)	pasta	table	2	105
8	0	45	(0.85 0.45 0.89)	(0.85 2.28 0.90)	pasta	table	1	75
9	0	30	(0.85 2.28 0.90)	(0.31 2.41 1.20)	pasta	middle left shelf	0	30
10	8	30	(1.17 0.18 0.92)	(0.85 0.45 0.92)	milk	table	1	60
11	0	30	(0.85 0.45 0.92)	(0.55 1.20 0.92)	milk	table	0	30
12	8	30	(1.36 0.31 0.92)	(0.85 0.45 0.92)	juice	table	1	60
13	0	30	(0.85 0.45 0.92)	(0.55 1.10 0.92)	juice	table	0	30

Table 6.2: Evaluation scenario 1 - Task list

As it can be seen, the prioritization does influence the outcome. For instance, in the c-priority case, the first batch to be auctioned is composed of the tasks 0, 2, 4, 7, 10, 12, while with the α-policy, the first batch is the set 4, 7. This results in a **different task sequence**; in the latter case, task 7 is scheduled second instead of seventh, which enables M2 to complete the succeeding tasks earlier. One can see that the **makespan** obtained with α-prioritization is **13% smaller** than the one obtained with the c-policy. The **task distribution** is also different. In the first case, the Capdi has two tasks to complete, the ones to bring the pasta and tomato sauce to the other side of the table. In the second case, it is also in charge of giving the milk and juice to the user, tasks that are completed by M1 otherwise.

In the variant without M1, the same observation can be made about prioritization, the task sequence and the makespan. The task distribution is the same however; and the saving in time is of just 1%. This can be accounted by the fact that, since each task can actually be performed by only one agent, there is less flexibility in the solution.

Comparing the two variants, results are as expected: the makespan is higher with one robot less; removing M1 led to **60% and 41% increases** in the total task completion time for the two subcases. In the first variant, one can indeed highlight the visible fact in the task distribution that M0 and M1 works in parallel most of the time.

Task	Start time(s)	Finish time(s)	Agent
4	5.7	35.7	M1
7	6.7	36.6	M0
8	36.6	81.6	Capdi
2	40.8	70.8	M1
0	44.8	74.8	M0
12	78.3	108.3	M1
10	80.8	110.8	M0
9	81.6	111.6	M2
5	96.8	141.8	Capdi
13	108.3	138.3	M1
3	111.8	141.8	M0
6	141.8	171.8	M2
11	148.6	178.6	M1
3	151.1	**181.1**	M0

(a) α-priority

Task	Start time(s)	Finish time(s)	Agent
4	5.7	35.7	M1
0	6.8	36.8	M0
5	35.7	80.7	Capdi
2	40.8	70.8	M1
10	42.8	72.8	M0
12	78.3	108.3	M1
7	79.7	109.7	M0
6	80.7	110.7	M2
11	96.8	126.8	Capdi
13	108.3	138.3	M1
1	110.3	140.3	M0
8	133.6	178.6	Capdi
3	149.6	179.6	M0
9	178.6	**208.6**	M2

(b) c-priority

Table 6.3: Scenario 1 output schedules - with four robots

Task	Start time(s)	Finish time(s)	Agent
7	6.6	36.6	M0
8	36.6	81.6	Capdi
4	42.9	72.9	M0
2	77.9	107.9	M0
9	81.6	111.6	M2
5	96.8	141.8	Capdi
0	116.2	146.2	M0
6	141.8	171.8	M2
12	153.7	183.7	M0
13	183.7	213.7	Capdi
10	189.7	219.7	M0
11	219.7	249.7	Capdi
1	220.7	250.7	M0
3	260.1	**290.1**	M0

(a) α-priority

Task	Start time(s)	Finish time(s)	Agent
0	6.8	36.8	M0
10	42.8	72.8	M0
11	72.8	102.8	Capdi
12	80.3	110.3	M0
13	110.3	140.3	Capdi
7	117.2	147.2	M0
8	147.2	192.2	Capdi
2	152.3	182.3	M0
4	188.6	218.6	M0
9	192.2	222.2	M2
5	218.6	263.6	Capdi
1	218.8	248.8	M0
3	258.1	288.1	M0
6	263.6	**293.6**	M2

(b) c-priority

Table 6.4: Scenario 1 output schedules - with three robots

6.2.3 Second scenario: involving the user

6.2.3.1 Scope

In the second scenario, the environment (fig. 6.13) matches the current Aurora setup, with only one MICO to work with the CAPDI. One can therefore test the MRTA solution on a potential real case. This experiment is done with a number of tasks similar to the previous one, but with longer precedence chains and less parallelism. It also involves the user: some of the tasks that cannot be performed by the robots will be assigned to the user. Having in mind the general framework of the Aurora project, it is imagined that the user is sitting in a wheelchair next to the table between the hobs and the sink, and has limited mobility.

The user is easily included to the team following the explications given in section 5.4.1, i.e. using the configuration file to declare an additional bidder with the same characteristics as the others.

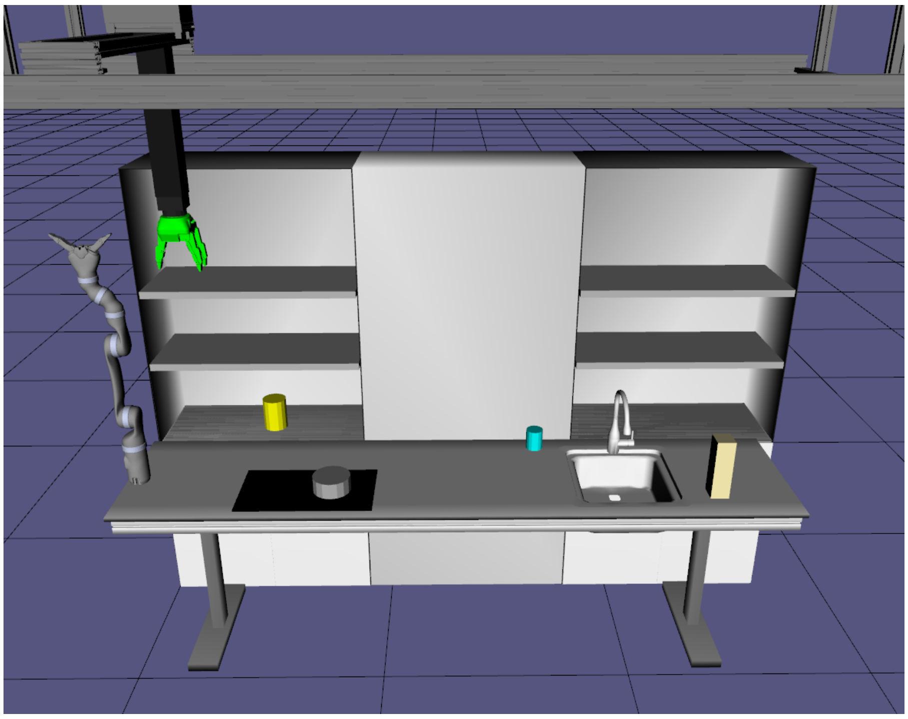


Figure 6.13: Evaluation scenario 2 - environment

6.2.3.2 Tasks

The story line of the test is the following one: let's imagine the user wants to prepare at hot chocolate. This general goal requires to complete various subgoals. In particular, one should heat up milk in a pan, and add chocolate powder in a glass. These two subgoals can then be decomposed into various atomic tasks, as shown in the precedence graph presented in figure 6.14. In comparison with the previous case, this graph has less disconnected components but longer branches. As previously, the figure gives for each task its type, the associated object as well as its id. Tasks 8 and 9 are the ones that should be performed by the user; they correspond respectively to the actions of opening the chocolate powder box, and putting powder in the glass (closing the box afterwards).

The different objects required for this scenario can be seen in figure 6.13 at their respective initial positions: the Nesquik box (represented in yellow) in the left cupboard, the pan (in grey) on the hob, the glass (in blue) next to the sink and the milk (in pale yellow) on the right side of the table. The task list can be found in table 6.5: the robots should pour milk in the pan, then place the pan on the hob and replace the milk at its initial position; in parallel, they should also bring to the user the glass and the Nesquik box, which must be put back in place after the user has finished using it.

6.2.3.3 Output schedule

The output schedules are reported in table 6.6. As for the previous experiment, they present the task sequence in ascending start time order, giving also the finish time and the agent.

As a first remark, one can see that the user is well integrated to the schedule. Since it was declared to the system as any other bidder, this could be expected.

Again, it is possible to see that the task prioritization does have an impact of the task sequence and the resulting makespan. The schedule obtained using the c-policy is quite close to the one obtained with the α-policy: except for the position of task 1, the task sequences are identical. As a consequence, the makespans are almost equal, with a slight advantage for the c-policy case (1.5% better).

Id	Type	Duration(s)	Position 1	Position 2	Object	Surface	c-priority	α-priority
0	0	45	(0.90 2.30 0.92)	(0.96 0.35 0.92)	milk	table	3	120
1	0	30	(0.86 0.82 0.83)	(0.86 0.35 0.83)	pan	table	3	105.7
2	3	30	(0.96 0.35 0.92)	(0.96 0.35 0.92)	milk	table	2	75
3	8	45	(0.96 0.35 0.92)	(0.90 2.30 0.92)	milk	table	0	45
4	0	30	(0.86 0.35 0.83)	(0.86 0.82 0.83)	pan	table	0	30
5	0	45	(0.31 0.50 0.97)	(0.72 0.35 0.88)	nesquik	table	5	240
6	0	30	(0.72 0.35 0.88)	(1.01 1.10 0.88)	nesquik	table	4	195
7	0	30	(0.52 1.63 0.85)	(1.01 1.17 0.85)	glass	table	3	165.4
8	6	30	(1.01 1.10 0.88)	(1.01 1.10 0.88)	nesquik	table	3	165
9	9	45	(1.01 1.10 0.88)	(1.01 1.10 0.88)	nesquik	table	2	135
10	0	45	(1.01 1.10 0.88)	(0.72 0.35 0.88)	nesquik	table	1	90
11	8	45	(0.72 0.35 0.88)	(0.31 0.50 0.97)	nesquik	lower left shelf	0	45

Table 6.5: Evaluation scenario 2 - Task list

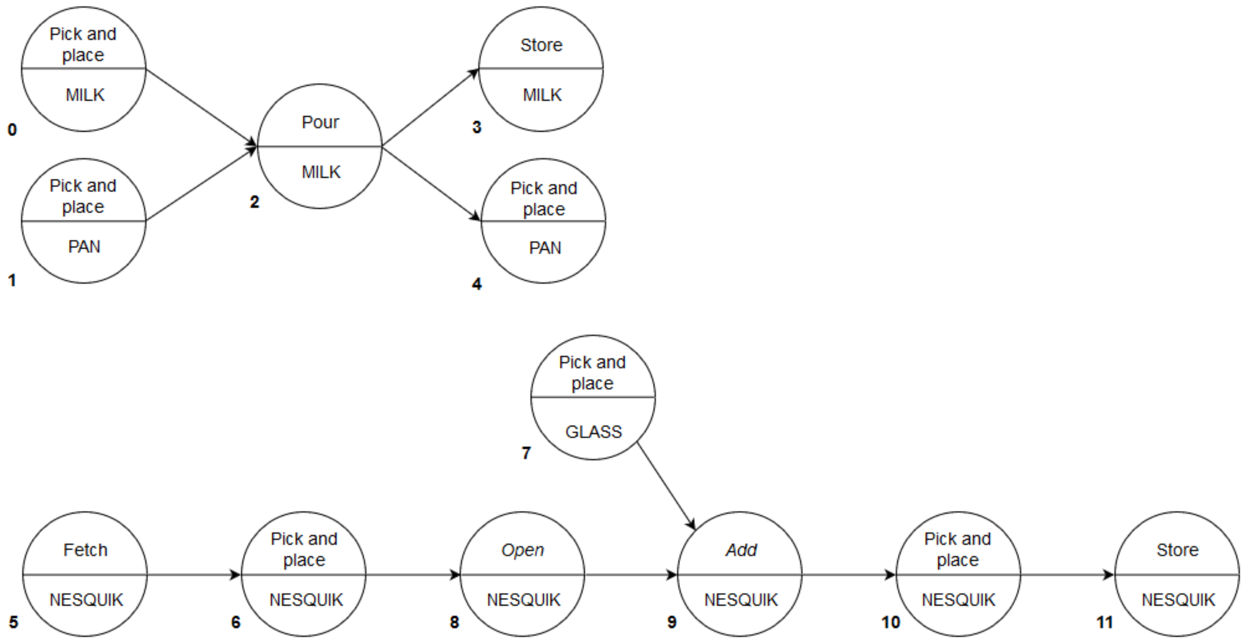


Figure 6.14: Evaluation scenario 2 - task precedence graph

The worse results are obtained for the unprioritized case, with a makespan that is almost 14% greater than the best case value. This can be accounted for by the fact that in this version, the tasks 6 and 8 are achieved later, which postpones the execution of the rest of the longer chain. In the other two sub-cases, the task that are completed last are the tasks 3 and 4, the lasts of the small chain in the precedence graph.

As a last remark, one could argue that here, it might be more important to put the pan on the hob before serving the Nesquik, since the milk needs time to heat up. Once more, this is related to task prioritization. Here the different policies only take into account the constraint data from the precedence graph, and not the higher-level information. One can possibly imagine a way of computing the priorities that also depends on the nature of the tasks.

Task	Start time(s)	Finish time(s)	Agent
7	6.3	36.3	Capdi
5	9.8	54.8	M0
6	54.8	84.8	Capdi
8	84.8	114.8	User
0	94.9	139.9	Capdi
9	114.8	159.8	User
1	143.9	173.9	Capdi
2	173.9	203.9	M0
10	180.3	225.3	Capdi
11	225.3	270.3	M0
4	226.5	256.5	Capdi
3	260.5	**305.5**	Capdi

(a) α-priority

Task	Start time(s)	Finish time(s)	Agent
7	6.3	36.3	Capdi
5	9.8	54.8	M0
1	39.5	69.5	Capdi
6	70.7	100.7	Capdi
8	100.7	130.7	User
0	110.7	155.7	Capdi
9	130.7	175.7	User
2	155.7	185.7	M0
10	175.7	220.7	Capdi
11	220.7	265.7	M0
4	221.9	251.9	Capdi
3	255.9	**300.9**	Capdi

(b) c-priority

Task	Start time(s)	Finish time(s)	Agent
7	6.3	36.3	Capdi
5	9.8	54.8	M0
0	45.7	90.7	Capdi
1	94.8	124.8	Capdi
2	124.8	154.8	M0
6	126.0	156.0	Capdi
8	156.0	186.0	User
4	162.4	192.4	Capdi
9	186.0	231.0	User
3	196.4	241.4	Capdi
10	251.6	296.6	Capdi
11	296.6	**341.6**	M0

(c) No priority

Table 6.6: Scenario 2 output schedules

6.2.4 Performances

To evaluate the computational performances of the system, each one of the four problem instances for scenario 1 and of the three instances for scenario 2 (considering the different priority policies) was run thirty times. The execution times for the auction process are reported in the box plots of figure 6.15. The bottom and top edges of each box indicate the 25th and 75th percentiles while the red line marks the median, and the whiskers extend to the most extreme values.

As it can be noticed, for scenario 1, the execution time tends to be higher when using the c-priority policy. This can be accounted for by the fact that the priorities influence the batch selection, and therefore the number of iterations. There are no differences however between the two subcases: using three or four bidders does not impact the results. It therefore appears that it is the auctioneer process that has the most impact on the execution time.

In scenario 2, one can only study the influence of the different priority policies. The c-policy yields almost the same output than in the no-priority case, and the variation is rather insignificant. The difference with the α-priority case is more marked, the median execution time being smaller in the latter case. A possible explanation to these results is that in the auctioneer process, the batch selection (which has an influence on the number of iterations) is more similar in the former cases than in the latter one.

Comparing the results obtained with the two scenarios, it is not possible however to identify a policy that would be significantly better regarding the execution time: it depends on the tasks and the precedence constraint layout. As a last comment, however, it can be highlighted that in most cases, the execution time is between 140 ms and 190 ms. This is suitable for Aurora-like applications, and means the implemented solution can easily be used online.

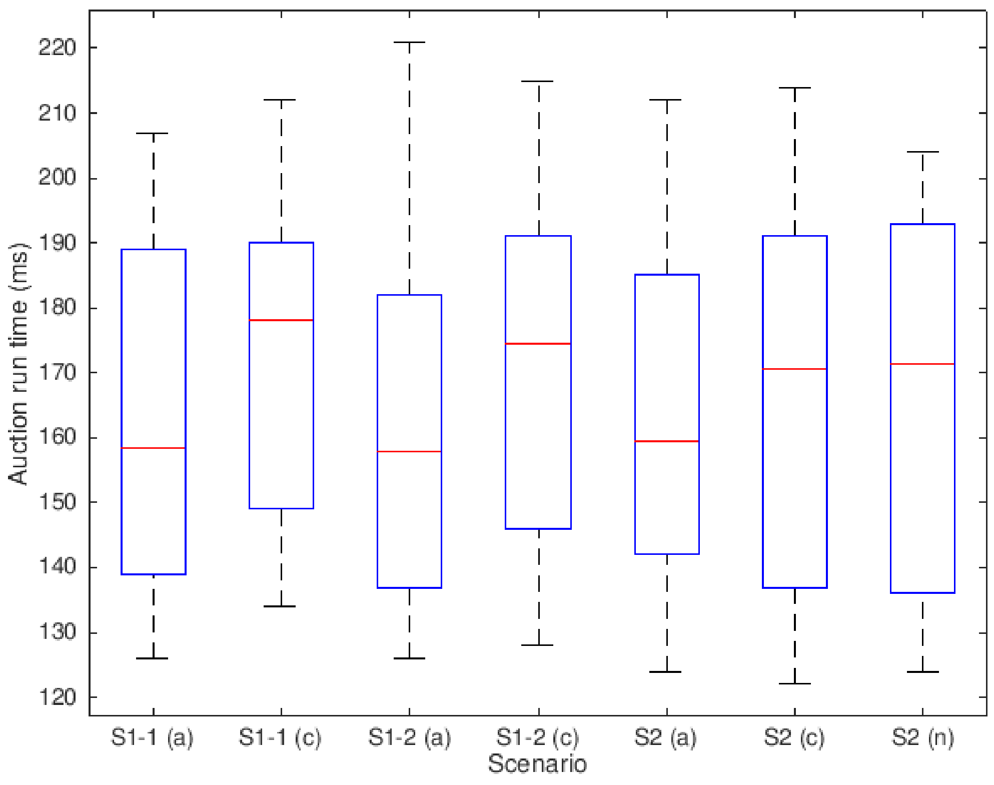


Figure 6.15: Auction process execution time for each scenario

Execution time computed over thirty trials for each problem instance. S1-1: first scenario with four robots; S1-2: first scenario with three robots; S2: second scenario. The letter between parentheses indicates the priority policy: a for α-policy, c for c-policy and n for no priority.

Chapter 7

Project impact

To complete this report, this chapter provides a brief analysis of the economical and environmental impact associated to the project.

7.1 Economical impact and budget

The work developed in this thesis is limited to research and is not currently designed to provide a marketable product. Therefore, it does not have any direct economic benefits. That is why we will only evaluate the cost of the project, considering equipment, workforce and energy expenses. Note that this estimation only takes into account the direct costs, i.e. the expenses that are directly imputable to the project. In particular, we shall not take into account the general expenses for the building - whether it is maintenance, cleaning or ordinary electricity/water consumption.

To reflect the reality of the **equipment expenses**, one have to take into account the amortization and not the cost of purchase. Table 7.1 presents the list of equipment used for the project as well as the associated cost of purchase and amortization time used to compute the actual corresponding expense. Note that this is an estimation only, since not all invoices were available. The depreciation periods were chosen according to classical values[1].

Item	**Cost of purchase** (€)	**Useful life** (years)	**Cost** (€/month)
Kinova robot arm (with Gripper) (*)	22 300	10	185.83
CAPDI structure and arm (*)	15 000	10	125.00
CAPDI Robotiq gripper (*)	15 000	10	125.00
Kitchen setup: cupboards set (*)	300	10	2.50
Kitchen setup: hobs	411.80	10	3.43
Kitchen setup: sink	69.60	10	0.58
Kitchen setup: table	57.72	10	0.48
Laboratory computer 1 (*)	600	3	16.67
Laboratory computer 2 (*)	600	3	16.67
Total			**476.16**

Table 7.1: Equipment expenses estimation (using linear depreciation; (*) = estimated)

The main **workforce** is formed by one graduated engineer (Group 2 within the UPC classification) working full-time during a period of five months and a supervisor dedicating 10% of his/her time to the project. The corresponding gross salary according to the UPC's official remuneration tables of the administration and labor service staff is reported in Table 7.2.

[1]See for instance `http://132.248.9.34/hevila/Revistahaciendamunicipal/2013/no120/3.pdf`

Employee	**Gross salary full-time** (€/m)	**Rate**	**Associated cost** (€/m)
Main engineer	1 979.29	100%	1979.29
Project director	4 282.77	10%	428.28
Total			2407.57

Table 7.2: Workforce expenses estimation

Finally, one can estimate the **energy expenses** associated to the electricity consumption. These values are gathered in Table 7.3.

Item	**E** (W)	**Use** (h/day)	**E** (kWh/month)	Cost (€/month)
MICO	25	1	0.58	0.06
CAPDI	300	1	6.90	0.75
Robotiq Adaptive Gripper	10	1	0.23	0.03
Computer 1 (Dell T1500)	350	8	64.40	7.08
Monitor Computer 1 (Dell E2211H)	20	8	3.68	0.40
Monitor Computer 1 (Dell P190Sb)	25	8	4.60	0.51
Computer 2 (Dell T7500)	350	8	64.40	7.08
Computer 2 (Dell T7500), idle	20	16	7.36	0.81
Monitor Computer 2 (TV Panasonic)	425	1	9.78	1.08
Office lighting (neon tube x4)	600	4	55.20	6.07
Total				23.88

Table 7.3: Energy expenses estimation (with electricity cost: 0.11€/kWh, 23 days per month)

The total cost of the project is obtained by adding the elements previously detailed. The results are shown in Table 7.4.

Item	**Cost per month** (€)	**Cost for project** (€)
Equipment	476.16	2 380.80
Energy expenses	23.88	119.42
Workforce	2407.57	12 037.85
Total	2 907.61	**14 538.07**

Table 7.4: Estimated total cost for the project

7.2 Environmental impact

Besides the economical aspects, it is important to evaluate the social and environmental impact of the project. The evaluation procedure is framed at national level by the Environmental Impact Assessment Law (*Ley 21/2013, de 9 de diciembre, de evaluación ambiental*[2]), following a directive from the European Union. This law establishes the bases that should govern the environmental impact evaluation of projects, in order to promote sustainable development and protect the environment.

The work presented in this Master's Thesis did not have any significant impact on the environment other than the one associated to the equipment that was used. The main impact of the project is linked to the energy consumption - to which one could add the hidden impact derived from the resource consumption that were necessary to manufacture the different elements (computers, robots, furniture...). The estimated energy consumption is presented in Table 7.5.

[2]`https://www.boe.es/buscar/pdf/2013/BOE-A-2013-12913-consolidado.pdf`

Item	**Power** (W)	**Use** (h/day)	**E** (kWh/month)	**Total** (kWh)
MICO	25	1	0.58	2.88
CAPDI	300	1	6.90	34.50
Robotiq Adaptive Gripper	10	1	0.23	1.15
Computer 1 (Dell T1500)	350	8	64.40	322.00
Monitor Computer 1 (Dell E2211H)	20	8	3.68	18.40
Monitor Computer 1 (Dell P190Sb)	25	8	4.60	23.00
Computer 2 (Dell T7500)	350	8	64.40	322.00
Computer 2 (Dell T7500), idle	20	16	7.36	36.80
Monitor Computer 2 (TV Panasonic)	425	1	9.78	48.88
Office lighting (neon tube x4)	600	4	55.20	276.00
Total project			217.12	**1 085.60**

Table 7.5: Energy consumption estimation (5 months, 23 days per month)

Using the 2017 report on the Spanish electricity system[3] by Red Eléctrica de España, we can estimate the carbon footprint of the electricity used to be of around 0.28 kgCO_2/kWh, which gives for our project an equivalent emission of around **304 kg of CO_2**.

7.3 Social impact

As such, this project is not intended to design an actual product to be commercialized. Still, research done within the frame of of projects such as the AURORA project have the potential to impact people's lives when they are subsequently used to build marketable items.

One of the main societal issue would probably be the price of such devices, since the robots are far from being affordable for every individual. This particular point comes within bigger political and societal considerations to be solved at national or even international level, in which we should question our vision of assistance and healthcare.

Besides this complex question, the general impact however remains positive, since such systems are a way to assist disabled or elderly users in their daily life, so that they can be more independent at home. By helping these persons being more autonomous, that kind of robotized system also helps relieving the burden of caregivers.

[3]`https://www.ree.es/sites/default/files/11_PUBLICACIONES/Documentos/InformesSistemaElectrico/2017/spanish-electricity-system-2017.pdf` (see p.45)

Chapter 8

Conclusion

The work presented aimed at analyzing possible solutions and developing a multi-robot task allocation system, with in mind as end application the AURORA project. The identified main requirements for the solution were the following:

- the system should be able to handle the heterogeneity of the robot team members that have different capabilities,
- the system should be able to handle precedence constraints that partially impose the order of the task execution sequence,
- the system should be able to run online, so that it can be used in practice.

The first part of the work consisted in reviewing the existing literature on multi-robot systems and multi-robot task allocation to better apprehend the stakes of the problem and to select the relevant methods that could be applied to our project. This review of the state of the art has been presented in the report. In particular, several useful MRTA taxonomies were presented and discussed. Analyzing our objective through the prism of this classification helped to select one of the possible MRTA methods that have been mentioned and explained as well.

More specifically, the key application considered here falls into the ST-SR-TA:SP category within the MRTA/TOC taxonomy. What is more, because of the heterogeneity requirements, auction-based assignment methods are the best suited. Taking into account these two aspects, it was decided to implement the MRTA solution following the method developed by McIntire et al. who presented an *iterated multi-robot [auction algorithm] for precedence-constrained task scheduling*.

The model presented is quite general and flexible. It has been adapted to take into account capabilities, in order to deal with heterogeneous teams; it was also extended with the possibility to provide only partial schedules. The corresponding code has been developed in C++ using the ROS framework. The implementation, also presented in this report, is therefore structured to be compatible with ROS.

The solution has been tested on various scenarios. A first set of evaluation tests using the Solomon's VRP database proved the basic good functioning of the solution, while showing its flexibility and performance. A second set of tests demonstrated the possible use of the implemented algorithm applied to real AURORA scenarios. Since the algorithm gives the possibility to add priorities, several prioritization policies have been tested. From the experiments, prioritization appears to improve the results, though for now it is not possible to conclude about which policy is the best. Some results also illustrated a weakness of the algorithm, which yields in some cases sub-optimal outputs; this is the counterpart of the necessary restrictions that enable management of precedence constraint and online real-time execution. The measured execution times and communication processes ensure the practical aspect of the solution, that can even take into account human users.

Besides the exposed main requirements, one can also say that, with respect to the stated objectives:

- the solution is indeed **flexible** and **scalable**; it can be used for various problems with different configurations and number of bidders.
- the solution allows for **partial scheduling** if some tasks are unfeasible, which ensures **robustness**.
- the solution offers the possibility to take into account **human-in-the-loop** scenarios.

Future work

Of course, there are still many ways to improve and complete the proposed solution. This section provides a few points that could be further explored:

- **extending the robot team:** currently, the actual scenarios that can be studied in "real life" only include the Capdi robot in collaboration with a Kinova Mico robotic arm. The laboratory where the AURORA setup is installed also has a Rethink Robotics Baxter robot and a KUKA robot. Baxter is a two-armed robot that can be used for various manipulation tasks. It is between 175 cm and 190 cm tall, and is equipped with an animated face and some camera sensors. The KUKA robot is a more classical robotic arm. Both could be added to the system to extend its global capabilities. Also, the abilities of the CAPDI could be enhanced by taking advantage of the dexterity offered by the Robotiq Adaptive gripper.
- **task decomposition management:** the task decomposition management was out of the scope of this thesis. It is however a crucial aspect of task allocation. For the application considered here, it is also necessary to be able to generate from higher-level goals the relevant list of atomic tasks, with the associated constraints. This decomposition is essential for the system to work, and the chosen methodology has a significant influence on the outcome. A possible future work proposal could therefore focus on this particular aspect.
- **execution dynamics management:** along with the task decomposition, further work is required in order to implement the essential features to manage failures and exceptions, a part that was out of the scope of the present project. This part would greatly enhance the system by making it reliable and fully usable. In link with this point, it is possible to work towards a **better integration of the user**, focusing on communication as well as uncertainty management because of the user behaviour.

These points are just some of the leads that can be explored. Multi-robot task allocation is a complex and broad research field, and no doubt that there is still a lot to study, discover and learn.

Appendix A

Auctioneer and Bidder class definition

Listing A.1: Definition of the class Auctioneer

```
#ifndef AUCTIONEER_H
#define AUCTIONEER_H

#include <ros/ros.h>

#include <mrta_pia/bid.h>
#include <mrta_pia/graph.h>
#include <mrta_pia/task.h>

#include <mrta_pia_msgs/Auction.h>
#include <mrta_pia_msgs/FinishTimes.h>
#include <std_msgs/Bool.h>

#include <utils/utils.h>

/** @class Auctioneer class for the iterated-auction MRTA process
*/

class Auctioneer
{

public:

    Auctioneer(ros::NodeHandle n);
    ~Auctioneer();

    void get_f(std::vector<float> &f);
    void get_pc(std::vector<float> &pc);
    int get_number_of_scheduled_tasks();

    // Public functions to run auction
    void init_auction(mrta_pia::Graph* graphPtr,
                          std::vector<mrta_pia::Task> tasks,
                          std::vector<std::string> bidders);
    void run_auction();
    void stop_auction();
    void reset();

private:
    ros::NodeHandle nh_;

    mrta_pia::Graph* graph_;             // Precedence graph
    std::vector<mrta_pia::Task> tasks_; // Tasks to assign

    std::vector<float> f_;

    int nb_ts_;            // Number of tasks to schedule
    int ts_count_;         // Counter of scheduled tasks

    std::vector<std::string> participants_;  // Robots actually participating
    int nb_bidders_auct_;

    int auct_count_;          // Counter - general auction process
    int auct_batch_;          // Counter - iteration (= 1 batch to auction)
    int auct_round_;          // Counter - round within iteration
    std::vector<int> tauct_; // Tasks to be auctioned
    std::vector<int> tauct_assigned_; // Tasks from batch assigned
    int maxprio_;             // Maximum priority of second layer tasks

    std::vector<mrta_pia::Bid> bids_; // Collected bids
    mrta_pia::Bid* best_bid_;
    float best_bid_value_;
    const float inf = 99999;  // A large value to serve as infinite

    // Received bids and finish time records
    std::map<std::string,bool> receivedBidsRecord_;
    std::map<std::string,bool> receivedFRecord_;

    bool received_all_bids_();
    bool received_all_f_();

    // Publishers and subscribers for communication
    ros::Publisher stop_pub;
    ros::Publisher auction_pub;
    ros::Publisher winner_pub;
    ros::Subscriber bids_sub;
    ros::Subscriber f_sub;

    // Auctioneer core methods
    void build_tauct_();

    void send_auction_();
    void select_winner_();
    void announce_winner_();

    void receive_bids_(const mrta_pia_msgs::Bid &msg);
    void receive_f_(const mrta_pia_msgs::FinishTimes &msg);

    void mark_unassigned_();

};

#endif // AUCTIONEER_H
```

Listing A.2: Definition of the class Bidder

```
#ifndef BIDDER_H
#define BIDDER_H

#include <ros/ros.h>

#include <mrta_pia_msgs/Auction.h>
#include <mrta_pia_msgs/Bid.h>
#include <mrta_pia_msgs/FinishTimes.h>
#include <mrta_pia_msgs/Task.h>
#include <std_msgs/Bool.h>

#include <mrta_pia/bid.h>
#include <mrta_pia/robot.h>
#include <mrta_pia/stn.h>

#include <vector>

namespace mrta_pia
{

class Bidder
{

public:

    Bidder(ros::NodeHandle n, mrta_pia::Robot* robot);
    ~Bidder();

    void run_auction();

private:

    ros::NodeHandle nh_;
    mrta_pia::Robot* robot_;                     // Robot features

    std::vector<mrta_pia_msgs::Task> tauct_; // Auctioned batch of tasks
    std::vector<float> pc_;                      // Received PC values

    mrta_pia::STN* schedule_;                    // Schedule
    std::vector<int> assigned_tasks_;            // Assigned tasks
    std::vector<float> f_;                       // Corresponding finish times

    mrta_pia::Bid* best_bid_;
    mrta_pia::Bid* winning_bid_;
    int best_bid_position_;
    int best_bid_index_;

    int auction_id_;     // Auction counter
    int batch_id_;       // Batch counter
    int round_id_;       // Round counter

    // Publishers and subscribers for communication
    ros::Publisher bid_pub_;
    ros::Publisher f_pub_;
    ros::Subscriber auction_sub_;
    ros::Subscriber winner_sub_;
    ros::Subscriber stop_auction_sub_;
    ros::Publisher schedule_pub_;

    bool continue_;
    bool received_winner_;
    bool received_auction_;

    // Bidder core methods
    void receive_auction_(const mrta_pia_msgs::Auction &msg);

    void compute_bid_();
    void send_bid_();

    void receive_winner_(const mrta_pia_msgs::Bid &msg);
    void update_tauct_(int sold_task_id);

    void tighten_schedule_();
    void send_f_();

    void stop_auction_(const std_msgs::Bool &msg);

};

}

#endif // BIDDER_H
```

Appendix B

Data structures definition

This appendix gathers the header files that define the different data structure and C++ classes presented in section 5.4.4. For sake of brevity, and because the structure is certainly the most important, the source code files are not presented.

Listing B.1: Graph class definition

```
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <iterator>
#include <algorithm>
#include <vector>
#include <string>

#include <mrta_pia/task.h>
#include <utils/utils.h>

namespace mrta_pia {

class Graph {

public:

    struct node {
        typedef std::pair <int,node*> edge;
        std::vector<edge> in, out;
        int id;
        node(int i): id(i){}
    };


    Graph(int nb_nodes);
    ~Graph();

    void init();
    void print();
    void add_node(node n);

    void get_parents(const node &t, std::vector<int> &parents);
    void get_children(const node &t, std::vector<int> &children);
    bool is_in_tf(int task);
    bool is_in_tl(int task);

    void get_PC(std::vector<float> &pc);
    void get_F(std::vector<float> &f);

    float get_pc_el(int index);

    void set_F(std::vector<float> f);

    void update(const int task_index);
    void mark_unfeasible(int task_index);
    void get_unfeasible_tasks(std::vector<int> &unfeasible);

private:

    std::vector<node> nodes_;

    std::vector<int> ts_; // Scheduled tasks
    std::vector<int> tu_; // Unfeasible tasks
    std::vector<int> tf_; // Free tasks
    std::vector<int> tl_; // Second-layer tasks
    std::vector<int> th_; // Hidden tasks

    std::vector<float> pc_; // Earliest start time
    std::vector<float> f_; // Finishing times

    // Methods to build layers at graph initialization
    void build_tf_();
    void build_tl_();
    void build_th_();

    // Move task with id from v1 to v2
    void move_(const int id, std::vector<int> &v1,
                                            std::vector<int> &v2);
    // Update PC value
    void update_pc_(node t);

    // Utilitary functions
    bool parents_are_all_scheduled_(int node_id);
    bool parents_are_scheduled_or_free_(int node_id);
    void node_index_from_id(const int id, int &i);

};
}
#endif // GRAPH_H
```

Listing B.2: Robot class definition

```
#ifndef ROBOT_H
#define ROBOT_H

#include <iostream>
#include <string>
#include <vector>
#include <geometry_msgs/Point.h>
```

```
namespace mrta_pia
{

class Robot
{

public:

    Robot();
    Robot(std::string name, std::string topic,
                 std::vector<int> capabilities,
                 std::vector<float> workspace,
                 geometry_msgs::Point location,
                 float velocity);
    ~Robot();

    std::string get_name();
    std::string get_state_topic();
    std::vector<int> get_capabilities();
    std::vector<float> get_workspace();
    geometry_msgs::Point get_location();
    float get_velocity();

    void update_location();

    bool can_perform(int task_type);
    bool can_reach(geometry_msgs::Point p);

private:

    // Private attributes with default values
    std::string name_ = "";
    std::string state_topic_ = "";
    std::vector<int> capabilities_ = {};
    std::vector<float> workspace_ = {-0.5,0.5,-0.5,0.5,-0.5,0.5};
    geometry_msgs::Point location_;
    float velocity_ = 0;

};

}

#endif
```

Listing B.3: STN class definition

```
#ifndef STN_H
#define STN_H

#include <vector>
#include <cmath>

#include <mrta_pia/task.h>
#include <geometry_msgs/Point.h>
```

```cpp
#include <utils/stn_utils.h>

namespace mrta_pia {

class STN
{
    static const int inf = 999999; // Large nb to serve as infinity

public:

    STN(geometry_msgs::Point robot_location, float robot_velocity);
    ~STN();

    std::vector<mrta_pia::Task> get_scheduled_tasks();

    // Temporary insertion in working copy
    bool try_insert(const mrta_pia::Task &task, float pc_value,
                        float &r_makespan, int &r_position);
    // Final insertion at given position
    void insert(const mrta_pia::Task task, int position,
                        float pc_value);

    std::vector<float> tighten_schedule();

    void print();

private:

    std::vector<mrta_pia::Task> scheduled_tasks_;

    std::vector<std::vector<float>> dgraph_; // Distance graph
    std::vector<std::vector<float>> w_dgraph_; // Working copy

    float makespan_;
    float travel_speed_;

    // Methods for final insertion
    bool insert_first_task_(const mrta_pia::Task &task);
    void insert_task_in_graph_(const mrta_pia::Task &task, int p);
    void insert_task_constraints_(const mrta_pia::Task &task,
                                            int p, float pc);
    void insert_travel_constraints_(const mrta_pia::Task &task,
                                                int p);

    // Methods for temporary insertion
    bool tmp_insert_first_task_(const mrta_pia::Task &task);
    void tmp_insert_task_in_graph_();
    void tmp_insert_task_constraints_(const mrta_pia::Task &task,
                                                float pc);
    void tmp_insert_travel_constraints_(const mrta_pia::Task &task,
                                                int p);

    // Compute travel time from t1 to t2
    float compute_travel_time_(const mrta_pia::Task &t1,
                                    const mrta_pia::Task &t2);
    // Restrict latest finish time
    void add_lft_constraint_(const mrta_pia::Task &task, float lft);

};
}

#endif // STN_H
```

Listing B.4: Task structure definition

```
#ifndef TASK_H
#define TASK_H

#include <geometry_msgs/Point.h>
#include <mrta_pia_msgs/Task.h>

namespace mrta_pia {

struct Task {

    // Task type list
    static const int TYPE_DUMMY = -1;
    static const int TYPE_PICKPLACE = 0;
    static const int TYPE_PICK = 1;
    static const int TYPE_PLACE = 2;
    static const int TYPE_POUR = 3;
    static const int TYPE_HOLD = 4;
    static const int TYPE_MIX = 5;
    static const int TYPE_OPEN = 6;
    static const int TYPE_CLOSE = 7;
    static const int TYPE_REMOVE = 8;

    int id;
    int type;
    float priority;
    geometry_msgs::Point location_1, location_2;
    std::string object, surface;
    float duration, est, lst;

    // Constructor and copy constructors
    Task(int t_id, int t_type, float t_dur, float t_est, float lst,
         geometry_msgs::Point p1, geometry_msgs::Point p2,
         std::string obj, std::string surf, float prio);
    Task(const Task &task);
    Task(const mrta_pia_msgs::Task &msg);

    // Conversion to ROS message
    void toMsg(mrta_pia_msgs::Task &msg);

    bool operator==(const Task& task) const;
    bool operator!=(const Task& task) const;
};
}
#endif
```

List of Figures

ETSEIB

List of Tables

List of Algorithms

ETSEIB

Listings

ETSEIB

Bibliography

[1] P. Beeson and B. Ames. TRAC-IK: An open-source library for improved solving of generic inverse kinematics. In *Proceedings of the IEEE RAS Humanoids Conference*, pages 928–935, Nov. 2015.

[2] K. E. C. Booth, G. Nejat, and J. C. Beck. A constraint programming approach to multi-robot task allocation and scheduling in retirement homes. In M. Rueher, editor, *Principles and Practice of Constraint Programming*, pages 539–555, Cham, 2016. Springer International Publishing.

[3] A. Campeau-Lecours et al. Kinova modular robot arms for service robotics applications. *International Journal of Robotics Applications and Technologies*, 5:49–71, July 2017.

[4] A. Casals et al. Plataforma para un entorno asistencial inteligente heterogéneo. *Jornadas Nacionales de Robótica 2018*, June 2018.

[5] S. Chitta, I. Sucan, and S. Cousins. Moveit! [ROS topics]. *IEEE Robotics and Automation Magazine*, 19:18–19, Mar. 2012.

[6] D. Coleman et al. Reducing the barrier to entry of complex robotic software: a moveit! case study. *Journal of Software Engineering for Robotics*, 5(1):3–16, May 2014.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Section 22.1: Representation of graphs. In *Introduction to Algorithms (2nd ed.)*, pages 527–530. MIT Press and McGraw-Hill, Oct. 2006.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Section 25: All-pairs shortest paths. In *Introduction to Algorithms (2nd ed.)*, pages 620–642. MIT Press and McGraw-Hill, Oct. 2006.

[9] R. N. Darmanin and M. K. Bugeja. A review on multi-robot systems categorised by application domain. In *2017 25th Mediterranean Conference on Control and Automation (MED)*, pages 701–706, July 2017.

[10] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1):61–95, May 1991.

[11] M. D'Emidio and I. Khan. Multi-robot task allocation problem: Current trends and new ideas. In *Joint Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic co-located with the 2017 IEEE International Workshop on Measurements and Networking (2017 IEEE M&N)*, pages 99–103, Sept. 2017.

[12] A. Farinelli, L. Iocchi, and D. Nardi. Multirobot systems: A classification focused on coordination. *IEEE Transactions on Systems, Man and Cybernetics (Part B)*, 34(5):2015–2028, Oct. 2004.

[13] A. Gautam and S. Mohan. A review of research in multi-robot systems. In *Proceedings of the 2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS)*, pages 1–5, Aug. 2012.

[14] B. P. Gerkey and M. J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9):939–954, Sept. 2004.

[15] M. Gini. Multi-robot allocation of tasks with temporal and ordering constraints. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pages 4863–4869. AAAI Press, 2017.

[16] A. Khamis, A. Hussein, and A. Elmogy. *Multi-robot Task Allocation: A Review of the State-of-the-Art*, pages 31–51. Springer International Publishing, Cham, 2015.

[17] G. A. Korsah, A. Stentz, and M. B. Dias. A comprehensive taxonomy for multi-robot task allocation. *Internation Journal of Robotics Research*, 32(12):1495–1512, Oct. 2013.

[18] A. Kothari et al. Grasping objects big and small: Human heuristics relating grasp-type and object size. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–6, May 2018.

[19] J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation.*, volume 2, pages 995–1001, Apr. 2000.

[20] H. Lipson. Robots on the run. *Nature (online)*, Mar. 2019.

[21] L. Luo, N. Chakraborty, and K. P. Sycara. Multi-robot assignment algorithm for tasks with set precedence constraints. *2011 IEEE International Conference on Robotics and Automation*, pages 2526–2533, May 2011.

[22] M. McIntire, E. Nunes, and M. Gini. Iterated multi-robot auctions for precedence-constrained task scheduling. In *AAMAS 2016 - Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*, pages 1078–1086. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), Jan. 2016.

[23] E. Nunes et al. A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, 90(C):55–70, Apr. 2017.

[24] E. Nunes and M. Gini. Multi-robot auctions for allocation of tasks with temporal constraints. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2110–2216. AAAI Press, 2015.

[25] M. Quigley et al. ROS: an open-source robot operating system. volume 3, Jan. 2009.

[26] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, Mar. 1987.

[27] I. A. Sucan, M. Moll, and L. E. Kavraki. The open motion planning library. *IEEE Robotics Automation Magazine*, 19(4):72–82, Dec. 2012.

[28] R. M. Zlot. *An Auction-Based Approach to Complex Task Allocation for Multirobot Teams*. PhD thesis, Robotics Institute, Carnegie Mellon University, Oct. 2006.