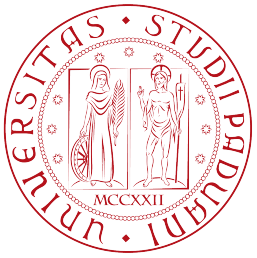


UNIVERSITÀ DI PADOVA FACOLTÀ DI INGEGNERIA

Role Assignment for Multi-Robot System in Dynamic and Uncertain Environments

Laureando DE BATTISTI RICCARDO

Relatore PAGELLO ENRICO
Correlatore FARINELLI ALESSANDRO

Corso di laurea Magistrale in INGEGNERIA INFORMATICA

Anno Accademico 2011-2012

A Lino, Mara e Francesca

Introduction

Coordination in a multi-robot system

The growing interest in developing colonies of robots engaged in complex tasks has caused an increasing interest in coordination approaches which can provide flexible and reliable solutions for different problems, for instance:

- Search and rescue;
- Monitoring environmental phenomena;
- Surveillance in security applications.

In fact, the coordination of the robotic activities can increase both the efficiency of the global task execution and the robustness of the system to individual robot failures. However, devising flexible and effective coordination methods for multi-robot systems is a very complex and challenging task, in fact:

- Coordination in these domains is particularly difficult because it requires the solution to be distributed among the robots to enhance robustness and avoid the existence of a central point of failure;
- The environment where robots act is highly dynamic and unpredictable;
- The coordination method should be able to react to unexpected changes and provide good-quality solutions minimizing the reaction time.

Even if agent-based coordination techniques are widely used to achieve cooperative behavior in distributed settings, after analyzing some of state-of-the-art approaches, we decide to focus on coalition formation. In detail in a *coalition formation* problem [1]:

1. A set of robots must cooperate to accomplish a set of tasks (or roles);
2. Each robot can execute one task at a time;
3. Robots can form coalitions to cooperate on specific tasks.

Indeed coalitions can perform tasks better, e.g. faster, than single robots and the quality of the execution of a specific task depends both on the individual capabilities that each robot has for that task, and on how the capabilities can be combined together. It is knows that *coalition formation* is an NP-hard problem and several approaches, which are also able to compute the optimal solution, have been studied to address such problems, e.g. [1, 2], however they do not consider that robots are situated in the environment hence collisions can occur when they act and move, because robotics *collision avoidance* and *task allocation* are kinds of coordination generally treated as separate entities.

Handled problems

The following problems are those that have been handled in this thesis:

- The development of a coordination system based on the distributed *Max-sum* algorithm whose messages are exchanged over a *factor graph* representation of the system, where robots, even with low resources, have to collaborate to offer services to tasks, which can require more than a robot to be accomplished;
- The explanation of a lower level coordination based on a *kinodynamic collision avoidance* approach which, in turn, is distributively optimized through the *Max-sum algorithm* over another *factor graph* system representation;
- The introduction of an hybrid approach which merges the *collision avoidance* system with the *task allocation* system into a unique architecture, in order to get some benefit from the usage of the same algorithm and the similar system representations, and trade off between greedy and optimal solutions approaches.

Experiments

The created system for solving a *coalition formation* problem through the *Max-sum algorithm* and the *factor graph* framework has been simulated and tested on the *Simulator Gazebo* over the ROS middleware. In detail we implemented:

1. Two utility functions which re able to:
   (a) Estimate robots' capacities to carry out all the system tasks;
   (b) Merge such capacities hence getting coalition utilities for each task.
2. Two *factor graph* versions, the former resulted in a bipartite *complete* graph, in order to solve the *task allocation* problem, the latter, needed for the *collision avoidance* system, which can permit a graph to be *incomplete*;

3. All the tools used for execute the *Max-sum algorithm*, e.g. the elaboration, maximization and normalization of exchanged messages;

4. The S*imulator Gazebo*'s environments necessary to test and make experiments.

5. The procedures needed to solve both the *task allocation* and the *collision avoidance* problem within the coordinated collision avoidance framework.

Moreover the same experiments have been done applying a greedy algorithm, called *agent satisfaction,* we took from the state of the art and implement in order to get comparisons with the proposed *task allocation* approach.

Results

The experiments has shown that such distributed *task allocation* approach works well managing to guarantee optimality with a few number of exchanged messages in all the tested instances, in particular those where a greedy approach has failed. However such tests have also stressed a *collision avoidance* system is fundamental in that kind of multi-robot environments, hence confirming an hybrid approach is necessary to get significant and important results. In fact the proposed hybrid structure has permitted to trade off between greedy approaches and optimal solutions algorithms, making robots able to avoid collisions, rapidly choose tasks and optimizing those choices at the same time. Nevertheless all carried out experiments are only simulated, hence future works could concern the test and analysis of the proposed systems on real Pioneer 3AT robots and real-life environments.

Outline

The outline of this thesis is as follows:

- In Chapter 1 we introduce the problem of coordination between robots, giving an example of the utility function called, *Q-function* and a d*ynamic role assignment* developed for a *RoboCup* scenario;

- In Chapter 2 we treat the *Optimal assignment problem* and how it is correlated with robot *role allocation*, proposing both centralized and distributed approaches which can lead to optimal or suboptimal solutions;

- In Chapter 3 the *Implicit coordination* is considered presenting some state-of-the-art approaches and a practical example of how can be used in a *Middle Size League RoboCup* team;

- In Chapter 4 we deal with some architectures needed to completely coordinate multi-robot systems in all their complex structure;

- In Chapter 5 we give an introduction to the utility function, the theoretical framework and the distributed algorithm used to solve the proposed task allocation problem and propose some toy problem examples;
- In Chapter 6 a detailed explanation of how the proposed system for the problem solution is given, in particular how the *factor graph* and the *Max-sum algorithm* really works over ROS middleware;
- In Chapter 7 after giving instances on how the proposed system works and making some interesting comparisons with a greedy approach to the problem, we show a distributed kinodynamic collision avoidance [3] and introduce the hybrid system we called *coordinated collision avoidance.* approach.

Contents

Part I

State of the art

Chapter 1

The importance of coordination

Communication among a group of robots should in principle improve the overall performance of the team of robots, as robots may share their world views and may negotiate task assignments. However, in practice, effectively handling in real-time multi-robot merge of information and coordination is a challenging task which usually requires complex and hybrid architectures (see Chapter 4). However it is important to distinguish a *Multi Agent Systems* (MAS) from *Multi Robot Systems* (MRS): the former are developed with regards to the simulated, software teams of robots, while refer to the real robot teams [4], but in both case the difficulties arise from the development of distributed approaches to coordination, because the advantages of coordination can not only be measured in terms of reached goals, but also in the achievements of subgoals. Obviously, when real robotic players are considered, issues such as uncertainty on the physical sensors and actuators emerge, so that coordinating a multi robot system, where there are many possible sources of errors, appears to be a rather challenging problem.

The concept of MAS is emerged as an important model of designing intelligent and complex software application, because it exploits diverse and distributed online information resources, and builds sophisticated and distributed systems that work effectively in a group setting.

In general, to perform specific tasks by cooperation of a team of multi agents, three distinguishing features are needed:

1. The environment where the agents act is unpredictable and the tasks are complex;
2. It should achieve effective coordination;
3. The behaviors in MAS are hierarchical in nature, including both low level emergent and high level cooperative behaviors.

These behavior-based MAS are suitable to a variety of tasks where either more than one agent is unable to do the job, or a team of agents cam achieve an

optimal solution in terms of time, energy efficiency and quality. Another basic feature of MAS is the system architecture, whose simplest version can be pure reactive, where the agents:

1. Perceive environment changes (inputs);
2. Interpret them;
3. React to them according to predefined procedures.

However, the really essential requirement for a real world agent acting in a dynamic and uncertain environment is to be able to show reactive as well as deliberative behaviors. This need of embodying both kinds of behaviors has lead to the so-called hybrid architectures (see Chapter 4), where because of the hierarchical task structure, two types of control flow can be identified within this layered architectures:

1. The horizontally layered, whose layers are each directly connected to the sensory input and to the action output (each layer acts like an agent);
2. The vertically layered, where sensory input and action output have dedicated layers, while intermediate layers perform the I/O processing.

On the other end in a MRS, every robot in order to successfully coordinate its action to achieve a common complex goal, must be aware, even if at a minimal level, of the subtask carried out by the other robots in the system. This problem is known as the problem of action recognition, that is the ability of a robot to observe and interpret the behaviors of another robot.

The research in cooperative robotics has widely investigated on this issue, classifying MRS in two major categories: MRS based on *implicit* and *explicit communication*. In the former communication is achieved throughout sensory feedback from the operating environment (see Chapter 3), while in the latter (see Chapter 1.2) the explicit exchange of information between robots, typically based on messages, is exploited. *Implicit communication* has the advantage of not requiring any type of common communication device and negotiation among the robots, leading to emergent, non intentional, coordination. On the other side, *explicit communication* has been more extensively exploited and has proven to be more efficient in different domains, because MRS can be designed and implemented such a way to guarantee important features, e.g. *robustness*, *adaptivity* and *fault tolerance*.

1.1 Robocup

RoboCup[1] is an international robotics competition founded in 1997 whose aim is to develop autonomous soccer robots with the intention of promoting research and education in the field of artificial intelligence. The contest is a very interesting domain in which experimenting coordination tasks, currently has four

[1] http://www.robocup.org

major competition domains, each with a number of leagues and sub-leagues and covers the following themes:

1. *RoboCup Soccer*, creating teams of fully autonomous, cooperative robots that exhibit advanced competitive behaviors and strategies, whose sub-league are:
   (a) *Standard Platform League*;
   (b) *Small Size League*;
   (c) *Middle Size League*;
   (d) *Simulation League*;
   (e) *Humanoid League*;
2. *RoboCup Rescue*, assisting emergency responders to save people and perform hazardous tasks with highly mobile, dexterous and semi-autonomous robots capable of mapping and negotiating complex environments;
3. *RoboCup@Home*, helping people in their daily lives at home and in public with autonomous and naturally interactive assistant robots;
4. *RoboCupJunior*, motivating young people to learn skills and knowledge necessary in science, technology, engineering, and mathematics as well as to foster their soft skills through participating in the creative process of building and programming autonomous robots.

The problem of coordinating the behaviors of a team of autonomous agents playing soccer is one of the scientific challenges that have been put forward for the *RoboCup Soccer* competitions. Even if in this part the focus in on these competitions with great care of the *Middle Size League*, let briefly describe all the leagues of the soccer competition domain.

In *Standard Platform League* all teams use identical (i.e. standard) robots, hence teams concentrate on software development only. Moreover omnidirectional vision is not allowed, forcing decision-making to trade vision resources for self-localization and ball localization. The *Small Size League* or *F180* league, as it is otherwise known, is one of the oldest *RoboCup* Soccer leagues. It focuses on the problem of intelligent multi-robot/agent cooperation and control in a highly dynamic environment with a hybrid centralized/distributed system. In *Middle Size League*, robots are no more than 50 cm diameter and play soccer in teams of up to 6 robots with regular size FIFA soccer ball on a field similar to a scaled human soccer field. Because all sensors are on-board and robots can use wireless networking to communicate, the research focus is on full autonomy and cooperation at plan and perception levels. The *Simulation League* is another old league in *RoboCup Soccer*, where independently moving software players which play on a virtual field inside a computer, has led to the development of artificial intelligence and team strategies. In the *Humanoid League*, autonomous robots, with a human-like body plan and human-like senses, play soccer against each

other. Dynamic walking, running, and kicking the ball while maintaining balance, visual perception of the ball, self-localization, and team play are among the many research issues investigated in the league.

An interesting case for coordination in MRS is the Middle Size League, where coordination usually relies on explicit communication, but due to the frequent communication failures, the robots must not depend completely on communication, nor on information provided by other robots. In such a setting and since the heterogeneity of robot players, coordination needs to be achieve without laying down drastic prerequisites on the knowledge of the single players, hence leading to distributed approaches.

1.1.1 Middle Size League

An interesting case for coordination in MRS is the Middle Size League, where coordination usually relies on explicit communication, but due to the frequent communication failures, the robots must not depend completely on communication, nor on information provided by other robots. In such a setting and since the heterogeneity of robot players, coordination needs to be achieve without laying down drastic prerequisites on the knowledge of the single players, hence leading to distributed approaches.

In the *Middle Size League* (MSL) the rules are generally the same as the laws of football except for some modifications:

1. The standard field dimensions are 12 m by 18 m;
2. The official tournament ball used in matches is any orange FIFA standard size 5 football;
3. Only 5 robots per team may play on the field, including the goalkeeper.
4. The robots may only explicitly communicate by means of a WLAN satisfying the IEEE 802.11 specification.

Moreover there are the following robot design restrictions:

1. The robot must fit inside a 52 cm $\times$ 52 cm $\times$ 80 cm box;
2. The keeper can also temporarily increase its dimensions to 60 cm $\times$ 60 cm $\times$ 80 cm or 52 cm $\times$ 52 cm $\times$ 90 cm, but only for 1 second, if the goal is endangered by an approaching ball (after complete reducing its size, he has to wait 4 seconds before he is allowed to expand again);
3. The maximum weight of the robot is 40 kg;
4. The base color of a robot's body must be black;
5. The paint or used material must be matte in order to minimize reflectivity.

1.2 Dynamic role assignment

Here we present a distributed and *explicit* coordination of a multi-robot system [5] based on a *dynamic role assignment*, which relies on the broadcast communication of utility functions. First of all, it is worth highlighting the hypotheses underlying the MRS:

- *Communication-based coordination*: the usage of communication among the robots to improve team performances, allowing the robots to acquire more information and self-organize in a more reliable way;
- *Autonomy* in coordination: the robots are capable to perform their task, possibly in a degraded way, even in case of partial or total lack of communication;
- *Distributed coordination*: the communication capabilities, combined with the autonomy requirement, require that each robot, while interacting with the others, must rely on local control;
- *Heterogeneity*: the robots are heterogeneous both from hardware and software viewpoints, they can usually perform the same tasks but with different performance;
- *Highly dynamic, hostile environment*: the robots must be able to perform the assigned task in the presence of external and dynamic changes in the environment.

The approach is a formation/role system, where a formation decomposes the task space defining a set of roles: each robot has the knowledge and capabilities necessary to play any role, therefore robots can switch their roles on the fly, if needed. The coordination protocol is based on broadcast communication of some data, which are processed by every robot in order to establish the formation that the team will adopt and the roles assigned to robots. The computation is *distributed*, because each robot must process the information coming from the others to identify the team formation and its own role and and the protocol is also *robust* because it relies on a little amount of transmitted data. It is also based on the concept of *utility function*s, which are defined off-line before the actual operation of the MRS in the environment, but they are then evaluated periodically during the robot mission and exchanged among the robots. The protocol includes two steps that are periodically executed on-line during the MRS mission:

1. *Role assignment*;
2. *Formation selection*.

Formally, an utility function $f_j^i(\cdot)$ for a robot R_i and role r_j is a function that, given the information about the status of the robot, returns the value that indicates how well R_i can play role r_j. In other words, it should return higher

values when robot R_i is in a good situation to play role r_j, lower values when the robot is a bad situation to fulfill it.

The definition of the utility functions is an important step in the design and realization of the MRS, and, since they are deeply related to the application domain of the MRS, it is not easy to develop a general methodology for defining them. The designer of the MRS must anyhow take into account two considerations:

1. There are some variables or conditions that characterize the state of the robot that are relevant for the execution of the task associated to a role;

2. Some parameters of the utility functions must be experimentally evaluated, since they are also depend from the characteristics of the individual robots.

Therefore the following steps are performed for defining the utility functions:

1. Identify the variables that are relevant for the execution of the task associated to a role;

2. Define the utility function as a linear combinations of these variables;

3. Perform a set of systematic experiments in order to determine the coefficients of the utility functions.

Even if calibration typically requires a significant experimental work, the experimentation of the proposed coordination protocol has been done in three stages:

1. A simulator;

2. Experimentation without playing;

3. Experimentation during actual games;

4. Analysis of log files of the games.

The first and easier experimental setting is provided by a simulator, which is useful for verifying the correctness of the protocol and for computing a first estimation of the coefficients of the *utility functions*. Then, the experiments with real robots have been done without and with playing: the former is needed to adjust the discrepancies arising from differences in heterogeneous robots' implementation, the latter to single out the failures of the coordination system. Finally an analysis of the *log files* generated during the games is very useful for identifying misbehaviors of the coordination systems, detecting several interesting features and further refining the *utility functions*.

1.2.1 Role assignment

Suppose we have n robots $\{R_1, \ldots, R_n\}$ and m roles $\{r_1, \ldots, r_m\}$, which are ordered with respect to importance in the global task to be performed. Moreover for each role r_j let P_j define the percentage of robots of the team that should be assigned to this role and let $A(i) = j$ denote that r_j is assigned to the robot R_i, hence the method for *dynamic role assignment* requires that each robot R_p computes the following steps (see Algorithm 1.1):

1. For each role r_j, robot R_p computes and broadcasts the values of the utility function $f_j^p(\cdot)$ (lines 1-2);

2. It collects the values of the utility function computed by other robots (lines 3-5);

3. After creating an empty list $\mathcal{L}$ for assigned robots (line 6), it assigns it sets roles to all robots (lines 7-13).

It is easy to see that every role is assigned to at most $P_j \times n$ robots and every robot is assigned to only on role. In fact at every cycle of the algorithm a different assignment $A(i) = j$ is done: j changes after $P_j \times n$ cycles and robots already included in the set $\mathcal{L}$ of assigned robots cannot be chosen for function assignments. In particular, the first role (i.e. the one with the highest priority), will be assigned to those robots that have the best utility values for role r_1, the second role to those among the remaining robots that have the best utility values for the second role, and so on, while in the case of a complete lack of communication all the robots will assume the most important role.

In order to obtain an effective application of the above algorithm, an important issue to be dealt with is the stability of decisions with respect to possible oscillations of the numerical parameters on which they depend upon. The method adopted to stabilize decisions is based on the notion of *hysteresis* (see Figure 1.1), which amounts to smoothing the changes in the parameters values. This technique prevents a numerical parameter's oscillation from causing oscillations in high level decisions. For instance, if at a certain instant robot R_i covers role r_j, its utility function $f_j^i(\cdot)$ for role r_j returns a higher value, or once a robot realizes a sudden difficulty in performing its task, all its utility functions must return low values so that the role can be assigned to other robots.Moreover in the case of a great loss of transmitted data due to interferences, the robots may have slightly inconsistent data. Therefore, there could be roles temporarily assigned to more than one robot or not assigned at all, but if it is assumed that the values of utility functions do not change sharply, the correct use of the hysteresis method guarantees that the roles will be correctly assigned almost always.

Algorithm 1.1 Dynamic role allocation

1. **for each** role r_j **do**
2. compute and broadcast $f_j^p(\cdot)$;
3. **for each** robot R_i $(i \neq p)$ **do**
4. **for each** role r_j **do**
5. collect $f_j^i(\cdot)$;
6. $\mathcal{L} = \emptyset$;
7. **for each** role r_j **do**
8. **for** $c = 1$ **to** $P_j \times n$ **do**
9. **begin**
10. $h = \arg\max_{(i \notin \mathcal{L})} \left\{ f_j^i(\cdot) \right\}$;
11. **if** $h = p$ **then** $A(p) = j$;
12. $\mathcal{L} = \mathcal{L} \cup \{h\}$;
13. **end**

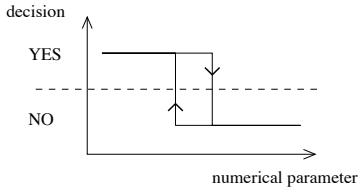


Figure 1.1: Hysteresis

1.2.2 Formation selection

The robots have at their disposal a number of predefined formations and rules to select the formation to adopt, on the basis of the environment configuration, where the instructions in Algorithm 1.2 for the formation selection.

Algorithm 1.2 Formation selection

1. **for each** robot R_i of the team **do**
2. **begin**
3. collect voted_formation[i];
4. votes[voted_formation[i]] = votes[voted_formation[i]] + 1;
5. **end**
6. **if** there is a formation f such that votes[f] $> n/2$ **then**
7. selected_formation = f;

Since each robot status do not necessarily coincide with those of the others, the robots may choose different formations, hence this algorithm is based on a voting scheme that allows for changing the formation only in presence of the absolute majority of votes. This voting scheme also ensure stability of decisions because the formation selection is accomplished at a lower frequency to that of role selection.

1.2.3 Q function

Q function [6], as an example of utility function, is a important measure of quality in a dynamic role assignment. According to its design, at any given time, the value taken by Q, depends on the following quantities:

- Its distance from the ball;
- Its relative position with regard to a right approaching configuration to the ball;
- The last visible position of the ball, if the ball is not currently visible;
- The position of other robots if there are any toward the goal;
- The number of failure while it is trying to move around collision-free;
- Its previous role.

Each robot R_i computes independently Q_i based on its local estimation. Then it sends the calculated value to the teammates (10 times per second) and decides autonomously how to behave comparing its own estimation Q_i with the values of the other robots. In this MRS scenario players may assume three different roles:

- *Master*, when the robot holds the ball, either as defender or attacker;
- *Active supporter*, when the robot cooperates with the *master* avoiding to interfere with it and protecting if from opponents;
- *Passive supporter*, when the robot has located far from the ball, but it is ready to enter the game.

This function guarantees *fault tolerance* and flexibility and prevents ambiguity, indeed to enhance role swapping *robustness*, and avoid the system instability, it was made sensitive to the previous role played by R_i. Thus, if the robot R_i do not play as *master* in the previous move, its actual value of Q_i is penalized in order to make more difficult to move from *active supporter* to *master*. To enforce the effectiveness of the *active supporter*, when a robot is assigned to that role, the following statements are always true:

1. It must never interfere with the *master*;
2. It must quickly try to get the ball if the *master* fails to perform its task;
3. It must keep itself close to the *master* to eventually recover the ball if the *master* looses it;
4. It must avoid any position on the straight line connecting the *master* with the opponent's goal.

For example if the *active supporter* meets the *master* along its moving to the ball, it handles the *master* as an obstacle and does not interfere with the master action, on the other hand, if it does not meet the *master* along its path to the ball, because the master is faced with some unexpected difficulties while performing its task, then the it is able to become the master. Indeed in the situation the *master* meets an opponent, while keeping the ball, it often makes a back step to avoid collision and the *active supporter* succeeds to move to a better approaching position to the ball. Hence the *master* makes room to the *active supporter*, that may take the ball, because it comes to be in a better position to score, and swap its role. The value of Q computed by the *master* becomes lower than the one computed by the *active supporter*, and after swapping their roles they succeeds to exchange the ball, showing an emergent behavior (see Figure 1.2).

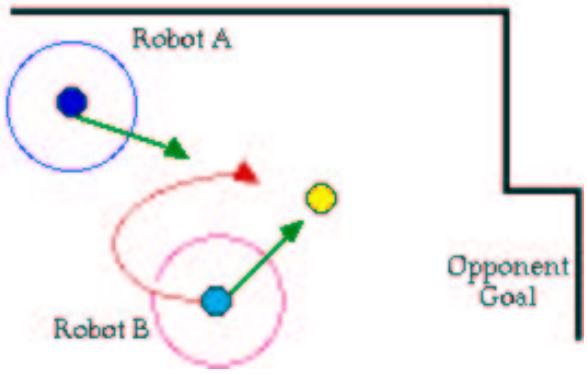


Figure 1.2: Ball exchanging

The coordination method described in Section 1.2 and a generalization of Q function have been implemented within the Azzurra Robot Team (ART team[2]), the italian national team of heterogeneous robotic soccer players participating in the Middle Size League RoboCup competitions.

The approach adopted by the ART team relies on a very flexible coordination protocol: it is based on a set of formations and a set of roles for every formation. The formation that has been mostly used is the *standard formation*, in which 3 roles for the 3 players are defined, i.e. *Attacker*, *Defender* and *Support*, while other formations have been considered to deal with special situations. The utility functions for these roles are determined as a linear combination of:

- The distance from the ball;
- The position of the robot in the field;
- The orientation of the robot in the field;

[2]http://www.dis.uniroma1.it/ART.

- The obstacles in the path towards the ball.

Since there are 3 roles for 3 robots the percentage role covering P_j are set to 1/3, so that every role is assigned to one robot. Moreover, the roles have priorities, so that if one robot is out or does not communicate with the others the first two roles (*Attacker* and *Defender*) are assigned to the remaining two robots, leaving the Support role unassigned.

Chapter 2

Optimal assignment problem

Let here introduce and examine the *optimal assignment* problem (OAP) [12], that is is a well-known problem that was originally studied in game theory and then in operations research, in the context of personnel assignment. In fact given m workers, each looking for one job, each requiring one worker, and for each worker a nonnegative skill rating estimating its performance for each job, the goal of the OAP problem is the assignment of workers to jobs in order to maximize the overall expected performance simultaneously taking into account the priorities of the jobs and the skill ratings of the workers.

The *multi-robot task allocation* problem (MRTA) can be posed similarly: given m robots, each capable of executing one task and possibly weighted tasks, each requiring one robot and given for each robot a nonnegative efficiency rating estimating its performance for each task the goal for the MRTA problem is to assign robots to tasks so as to maximize overall expected performance, taking into account the priorities of the tasks and the efficiency ratings of the robots.

In this chapter, it is shown how in several fields the OAP and consequently the MRTA problem can be casted in and then solved [14, 11], but before that, we first show how robots can calculate the performance estimates, which are also called *utility estimates*, next we formalize the MRTA problem and finally we present an interesting taxonomy [13] of *multi-robot task allocation* problem.

However the problem of role allocation is a dynamic decision problem, the OAP refers to a static environment, hence the static assignment is iteratively resolved over time. Of course, the cost of running the assignment algorithm must be taken into account: at one extreme, a costless algorithm can be executed arbitrarily quickly, ensuring an efficient assignment over time, on the other hand, an expensive algorithm that can only be executed once will produce a static assignment that is only initially efficient and will degrade over time.

In order to create and maintain an efficient allocation, even if it might be very expensive, the assignment algorithm must consider and reassign every role in the system. Indeed some implemented approaches to role allocation use heuristics to determine a subset of roles that will be considered in a particular iteration.

Formally, given a robot I and a role J , if I is capable of executing J and Q_{IJ} and C_{IJ} define the quality and cost, respectively, a combined and nonnegative utility measure for that performance can be derived as

$$U_{IJ} = \begin{cases} Q_{IJ} - C_{IJ} & if\ I\ is\ capable\ of\ executing\ J\ and\ Q_{IJ} > C_{ij} \\ 0 & otherwise \end{cases}.$$

For an instance of OAP, it is given:

- A set of m robots, $I_1,\ I_2,\ , \ldots I_m$;
- A set of n roles, $J_1,\ J_2,\ , \ldots J_n$ with relative weights $w_1,\ w_2,\ , \ldots w_n$;
- U_{ij}, the nonnegative utility of robot I_i for role J_j, $1 \leq i, j \leq n$.

It is also assumed that:

- Each robot I_i is capable of executing at most one role at any given time;
- Each role J_j requires exactly one robot to execute it.

Hence the problem is to find an optimal and feasible[1] allocation of robots to roles, that is a set of robot-role pairs

$$(i_1, j_1) \ldots (i_k, j_k)\,,\ 1 \leq k \leq \min(m, n)\,,$$

that maximize the weighted utility sum

$$U = \sum_{m=1}^{n} U_{i_m j_m} w_{j_m}.$$

The utilities are usually represented in a matrix form, called the *utility matrix*, however, some algorithms are designed to minimize, rather than maximize, the previous sum, so a cost matrix C from a utility matrix U can be obtained by

$$C_{ij} = U_{ij} - \max\{U_{ij}\}\ \forall i, j$$

that is subtracting each element in U from the maximum value in U. On the other hand other algorithms require that the utility (or cost) matrix is symmetric, i.e. $m = n$, hence we can construct a symmetric matrix from an asymmetric one by applying as many zeros padding as necessary without carrying out assignments that involve those columns and rows added.

The proposed taxonomy is done by dividing the space along three axes, hence getting:

- *Single-task robots* (ST) vs. *multi-task robots* (MT):

[1] An allocation is considered feasible, the robots $i_1 \ldots i_k$ and the task $j_1 \ldots j_k$ must be unique.

  - ST means that each robot is capable for executing as most one task at time;
  - MT means that some robots can execute multiple tasks simultaneously;

- *Single-robot tasks* (SR) vs. *multi-robot tasks* (MR):
  - SR means that each task requires exactly one robot to achieve it;
  - MR means that some tasks can require multiple robots;

- *Instantaneous assignment* (IA) vs. *time-extended assignment* (TA):
  - IA means that the available information concerning the robots, the tasks, and the environment permits only an instantaneous allocation of tasks to robots with no planning for future allocations;
  - TA means that more information is available, such as the set of all tasks that will need to be assigned, or a model of how tasks are expected to arrive over time.

Indeed, as described below, various MRTA problems can be positioned in the resulting problem space and some organizational theories are strongly related to those problems.

2.1 Solution approaches

Here we present some centralized but computationally complex approaches: a first and simple method to solve an OAP instance is the so called *brute force* approach, which:

1. Creates all the possible assignments;
2. Computes the relative costs;
3. Chooses the best one.

However, given an $n \times n$ matrix, there are n possibilities for the first assignment, $n-1$ for the second assignment, $n-2$ for the third assignment and so on, to the amounts of $n!$ possible assignments. Hence the complexity of this approach is at least exponential and not suitable to be implemented.

There are some approaches formulated specifically for the OAP that result in less computational complexity, e.g. the *Hungarian method*, which runs in $O\left(n^3\right)$ time and is efficient, even if run in real time. In detail, given a $n \times n$ cost matrix C, this method consists of several steps (see Algorithm 2.1).

Algorithm 2.1 Hungarian standard version

1. For each row:
   (a) Find the smallest element;
   (b) Subtract it to the entire row;
2. Until there is a zero uncovered:
   (a) Mark it with a star;
   (b) Cover its relative column and row;
3. Uncover all the rows and columns;
4. Cover all the columns with a starred zero;
5. If all columns are covered, END;
6. Until there are uncovered zeros:
   (a) Mark an unstarred zero with a bar;
   (b) If there are starred zeros in the same row:
      i. Cover the row and uncover the column with the starred zero;
      ii. Find the smallest value uncovered;
      iii. Add it to the elements of each covered row;
      iv. Subtract it to the elements of each uncovered column;
7. Add the first barred zero found in previous step into a list;
8. Until the list do no end with a barred zero, that has no starred zero in its column:
   (a) Add the starred zero into the list;
   (b) Add the first barred zero that is in the row of the barred zero;
9. For each element in the list
   (a) Remove the star, if it is a starred zero;
   (b) Substitute the bar with a stars, if it is a barred zero;
10. Uncover every line of the matrix;
11. GO to step 5;

Another way to solve this problem is done by achieving its graph representation and then applying a modified version of the Hungarian method (see

Algorithm 2.2), where the OAP becomes: given a bipartite complete graph[2] $G = (V_1 + V_2, E)$, find a minimum weight perfect matching[3].

Algorithm 2.2 Hungarian graph version

1. For each row:
   (a) Find the smallest element;
   (b) Subtract it to the entire row;
2. For each column:
   (a) Find the smallest element;
   (b) Subtract it to the entire column;
3. Get graph $G' = (V_1 + V_2, E')$, where $E' = \{(i, j) \mid C_{ij} = 0, \forall i, j\}$;
4. Find on G' a matching M of maximum cardinality;
5. If $|M| = |V_1| = |V_2|$, END;
6. Label all the rows unmatched in step 4;
7. Until there are rows or columns unlabeled:
   (a) Label the columns that have zeros in correspondence of labeled rows;
   (b) Label the unlabeled rows, matched by the matching algorithm;
8. Bar every row not labeled and every column labeled;
9. Get the minimum element not labeled;
10. Subtract it from all the unbarred elements;
11. Add it to the barred elements;
12. GO to step 3.

2.1.1 Sub optimal approaches

Sub optimal algorithms are those methods that guarantee to get admissible but not optimal solutions. Because of the high complexity of getting an optimal solution for an OAP instance they try to come closer the optimal solution as best as possible for examples by exploiting greedy paradigms.

[2] A complete bipartite graph $G = (V_1 + V_2, E)$ is a bipartite graph such that for any two vertices, $v_1 \in V_1$ and $v_2 \in V_2$, $(v_1, v_2) \in E$.

[3] A perfect matching is a matching which matches all vertices of the graph. That is, every vertex of the graph is incident to exactly one edge of the matching.

For instance, given a $n \times n$ utility matrix U, referred to a set of n robots $R = \{1, \ldots, n\}$ and a set of n tasks $T = \{1, \ldots, n\}$, a first approach is presented in Algorithm 2.3.

Algorithm 2.3 OAP greedy approach

1. For each robot i:
   (a) Find the task best suited for robot i, that is task j such that $u_{ij} = \max_k u_{ik}$;
   (b) Set task j as assigned.

However the simplicity of the approach is in contrast the high running complexity which equals $O\left(n^2\right)$.

The following greedy technique (see Algorithm 2.4) is another approach commonly used on role allocation problems.

Algorithm 2.4 OAP 2-competitive greedy approach

1. Until all roles have been assigned:
   (a) Find the highest utility u_{ij};
   (b) Assign robot i to role j;
   (c) Cross out row i and column j from the utility matrix;

The complexity of this algorithm is even higher than the previous one, because it fundamentally got worse by the research of the best utility value in the matrix, which spends a $O\left(n^2\right)$ time (hence $O\left(n^3\right)$ is the total running time). However it can be proved that the worst-case performance of this algorithm on the OAP is 2-competitive[4], that is this algorithm in the worst case produce a solution with utility that is $\frac{1}{2}$ of that given by an optimal solution.

2.2 Linear programming model

Given an $m \times n$ matrix A, an n-vector b, and an m-vector c, a *maximum problem for a linear program* tries to find a nonnegative m-vector x such that

$$xc \, is \, a \, maximum$$

under the constraints

$$xA \leq b.$$

[4] An algorithm is said to be α-competitive if, for any input, it finds a solution that is no worse than $\frac{1}{\alpha}$ of the optimum.

If there exists a vector x that satisfies the constraints, then the problem is called *feasible* and the vector x is a *feasible solution*, which is called an *optimal solution* if there is no other feasible solution that produces a higher value in the function to be maximized. Often the elements of the solution vector x are required to be integers, so making the problem is call an *integral linear program* (ILP). The MRTA problem can be cast as an ILP if the problem goal becomes finding, always in a centralized way, n^2 nonnegative integers α_{ij} that maximize

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_{ij}U_{ij}w_j \tag{2.1}$$

subject to:

$$\sum_{i=1}^{n}\alpha_{ij} = 1,\ 1 \leq j \leq n$$
$$\sum_{j=1}^{n}\alpha_{ij} = 1,\ 1 \leq i \leq n,$$

where

$$\alpha_{ij} = \begin{cases} 1 & ith\ robot\ executes\ jth\ task \\ 0 & otherwise \end{cases}.$$

Given an optimal solution to this problem, that is a set of integers α_{ij} under the above constraints, an optimal task allocation can be constructed by assigning robot i to task j only when $\alpha_{ij} = 1$.

By creating the *linear program* of Algorithm 2.5, the space of role allocation problems is restricted in the sense the function to be maximized, i.e. Equation (2.1), must be linear, but there is no such restriction on the manner in which the components of that function are derived, in other words, individual utilities can be computed in any arbitrary way, but they must be combined linearly.

Algorithm 2.5 ILP

$\max \sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_{ij}U_{ij}w_j$
$\sum_{i=1}^{n}\alpha_{ij} = 1,\ 1 \leq j \leq n$
$\sum_{j=1}^{n}\alpha_{ij} = 1,\ 1 \leq i \leq n$
$\alpha_{ij} \in \{0,1\},\ 1 \leq i,j \leq n$

Fundamental to the theory of linear programming is the concept of duality, that is to say, given a maximum LP problem, it is possible to construct a related minimum LP problem (and vice versa), where the original problem is called the

primal and the related problem is called the *dual*. In particular If both a problem and its dual are *feasible* then both have optimal solutions and the values of the optimal solutions are the same, while if either is infeasible, then neither has an optimal solution.

The dual of the MRTA problem (see Algorithm 2.6) can be stated as follows: find n integers u_i and n integers v_j that minimize

$$\sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j$$

subject to

$$u_i + v_j \geq U_{ij} \; \forall i, j.$$

Thus, given an instance of the MRTA problem, its dual can be trivially constructed, then whichever form is more computationally convenient can be solved because there exist algorithms for each.

Algorithm 2.6 Dual problem

$\min \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j$
$u_i + v_j \geq U_{ij} \; 1 \leq i, j \leq n$
u_i *integer*, $1 \leq i \leq n$
u_j *integer*, $1 \leq j \leq n$

The Dantzig's *simplex algorithm*, also called the *simplex method*, gives an optimal solution to an ILP problem whose matrix is TUM[5], after at most $\binom{n}{n}$ iterations.

2.3 Economic game

It is known that a strong connection between economics and game theory and game-theoretic techniques are often used to analyze and synthesize rules for economic systems. Hence the MRTA problem is here posed as an *economic game* and solved in a distributed manner, even gaining optimal solution.

Construct a price-based task market, in which tasks are sold by brokers to robots and each task j is for sale by a broker, which places a value c_j on the task. As each robot i places a value h_{ij} on task j, thus the the problem is to establish *feasible*[6] task prices p_j, which will in turn determine the allocation of tasks to robots. If we assume that robots are acting selfishly, each robot i will

[5] $A \in \mathbb{R}^{n \times n}$ is TUM if $det(Q) \in \{-1, 0, 1\}$ for each square sub matrix of order $k \leq n$. If A is TUM and $b \in \mathbb{R}^n$ is made up of integers, then the relative ILP problem has integer solutions.

[6] We say a price p_j is feasible for task j if it is greater than or equal to the broker's valuation c_j, because otherwise the broker would refuse to sell.

elect to buy a task t_i, where

$$t_i \in \arg\max_j \{h_{ij} - p_j\},$$

that is the task for which its profit is maximized. In particular a market is said to be at *equilibrium* when prices are such that no two robots select the same task, and consequently each individual's profit in this market is maximized. The profits made by the robots and the brokers form an optimal solution if the dual of the MRTA problem (see Algorithm 2.6), where

$$\begin{cases} u_i = h_{it_i} - p_{t_i} & \forall i \\ v_j = p_j - c_j & \forall j \end{cases}.$$

Since in the MRTA problem there are not separate valuations as above, but only combined *utility estimates* for robot-task pairs, if we define task valuations for the robots and brokers as

$$\begin{cases} h_{ij} = \alpha_{ij} & \forall i, j \\ c_j = 0 & \forall j \end{cases},$$

we make the solution of the corresponding dual problem become

$$\begin{cases} u_i = \alpha_{it_i} - p_{t_i} & \forall i \\ v_j = p_j & \forall j \end{cases}.$$

Setting $c_j = 0$ implicitly states that the brokers always prefer to sell their tasks, regardless of how much they are paid, i.e. it is always better to execute a task than not execute it, regardless of the expected performance.

For solving this economic form of the MRTA problem, a number of algorithms are available, where prices are usually determined by some kind of auction in which the participants make bids on items of interest. For the purposes of Algorithm 2.7, ϵ is a positive scalar and a robot i is said to be *happy* if and only if it is assigned to a task t_i for which its profit is ϵ-maximized, i.e.

$$\alpha_{it_i} - p_{t_i} \geq \max_j \{\alpha_{ij} - p_j\} - \epsilon.$$

Algorithm 2.7 Bertsekas's Auction

1. Randomly assign tasks to robots and prices to tasks;
2. Randomly select a robot i that is not happy;
3. If no such robot exists END;
4. Find a task t_i that maximizes profit for robot i;
5. Swap tasks between robot i and the robot that is currently assigned task t_i;
6. Increment the price of t_i by $\gamma_i = v_i - w_i + \epsilon$, where $v_i = \max_j \{\alpha_{ij} - p_j\}$ and $w_i = \max_{j \neq t_i} \{\alpha_{ij} - p_j\}$;
7. GO to step 2.

This auction algorithm is guaranteed to terminate and it produces an assignment for which the overall utility is within $n\epsilon$ of the optimal utility, but if the utilities α_{ij} are integral and if $\epsilon < \frac{1}{n}$, then the assignment is also optimal.

2.4 Stable marriage problem

The MRTA problem can also be casted in the *stable marriage problem* (SMP), that is the problem of finding a stable matching between two sets of elements with a given a set of preferences for each element, and solved with a distributed but computationally complex algorithm. A matching is a mapping from the elements of one set to the elements of the other set and it is defined as *stable* whenever it is not the case that both:

1. Some given element A of the first matched set prefers some given element B of the second matched set over the element to which A is already matched;
2. B also prefers A over the element to which B is already matched.

In other words, a matching is *stable* when there does not exist any alternative pairing (A, B), in which both A and B are individually better off than they would be with the element to which they are currently matched.

Hence given n men and n women first each person ranks all members of the opposite sex with a unique number between 1 and n in order of preference, then the algorithm tries to marry the men and women together. These marriages are chosen so that there are no two people of opposite sex who would both rather have each other than their current partners, i.e. the algorithm tries to get *stable* marriages.

An instance of SMP, whose solving pseudocode is shown in Algorithm 2.8, can be cast to an instance of MRTA simply by substituting in the previous

statement the word man with robot and the word woman to task. This algorithm guarantees with a $O\left(n^2+2n+2\right)$ complexity that everyone gets married and that such the marriages are *stable*, but it was empirically proved that it is not good in real situations because of the high number of messages that robots have to exchange each other.

Algorithm 2.8 Stable marriage

```
1. Initialize all m ∈ M and w ∈ W to free
2. while ∃ free man m who still has a woman w to propose to
3.     w = m's highest ranked such woman who he has not proposed to yet;
4.     if w is free (m, w) become engaged;
5.     else // some pair (m', w) already exists
6.         if w prefers m to m'
7.             (m, w) become engaged;
8.             m' becomes free;
9.         else
10.            (m', w) remain engaged;
11.        end if
12.    end if
13. end while
```

2.5 Network flow problem

The fastest but centralized way to solve an OAP problem is casting it in a general *network flow* problem, a weighted graph with maximum capacities for the edges is give: the nodes in the graph that can provide a common resource are called *sources*, while nodes that require the common resource are called *sinks*. In particular, when dealing with a *Minimum cost network flow problem*, the given directed graph $G=(V,E)$ is such that:

1. $|E|=n$;
2. Edges have Nonnegative costs c_e;
3. Edges have capacities k_e;
4. Vertexes have demands b_v so that:
   (a) If $b_v<0$, v is a *sink*;
   (b) If $b_v>0$, v is a *source*.

Let $\delta(v)$ and $\gamma(v)$ respectively denote the set of edges entering and leaving each vertex $v\in V$, the problem consists of finding n flows f_e that minimize

$$\sum_{e\in E} c_e f_e,$$

subject to conditions

$$\sum_{e \in \gamma(v)} f_e - \sum_{e \in \delta(v)} f_e = b_v, \ \forall v \in V$$

$$f_e \leq k_e, \ \forall e \in E$$

that is, find the cheapest way to satisfy demands, satisfying *source* and capacity constraints.

Hence the MRTA problem can be considered as a network flow problem after constructing a bipartite graph $G = (R \cup T, E)$, where:

- $n = |R| = |T|$;
- R is the set of robots;
- T is the set of tasks;
- E is the set of possible robot-task pairs.

The cost c_e associated with an edge is defined to be the cost estimate of the underlying robot-task pair and the capacities k_e for all edges in the graph are set to 1. Moreover G must be modified, getting G_x by adding:

1. A *source* node p with demand $b_p = n$;
2. A *sink* node q with demand $b_q = -n$;
3. For each robot $r_i \in R$, a zero-cost edge linking p and r_i;
4. For each task $t_i \in T$, a zero-cost edge linking t_i and q.

Then a solution can be obtained by finding the minimum cost integral flow in the previous modified graph G_x, for instance by choosing successive shortest paths from the source p to the sink q, as in Algorithm 2.9 where:

1. A flow is sent from p to q along the shortest path respecting arc costs;
2. The residual network is updated;
3. Another shortest path is found and the flow is augmented;
4. If the residual network contains a path from p to q, i.e. the flow is not maximal, GO to step 1.

Since the flow is maximal, it corresponds to a feasible solution of the original minimum cost flow problem (it can be proved it is also optimal). By implementing this search with Dijkstra's algorithm for computing shortest paths on

Fibonacci heaps[7], an $n \times n$ MRTA problem can be solved in time $O\left(n^2 \log n\right)$, which is the fastest known algorithm for the OAP.

Algorithm 2.9 Successive paths

```
1 Initial flow x is zero
2 while G_x contains a path from p to q do
3       Find any shortest path P from p to q
4       Augment current flow x along P
5       Update G_x
6 end while
```

2.6 Scheduling problem

Given a set of machines, a set of jobs to be processed, and a performance criterion, a schedule of jobs for each machine have to be constructed such that performance is maximized. A *scheduling problem* is formally defined by three variables:

1. The machine environment α;
2. The job characteristics β;
3. The optimality criterion γ.

Traditionally, a problem's classification is given by the triple:

$$\alpha \mid \beta \mid \gamma,$$

where a *machine environment* α determines how many machines are available, which jobs each machine can process and how fast each machine can be expected to process a given job. It will generally be:

1. P if indicates an arbitrary number of identical parallel machines that all process each job at the same rate;
2. R if indicates an arbitrary number of unrelated parallel machines that potentially process each job at a different rate.

Moreover the *job characteristics* β determine what relationships and constraints exist between the jobs. It will generally be empty, to indicate that the individual jobs are not related to or dependent upon each other. Finally the *optimality*

[7] A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent. This implies that the minimum key is always at the root of one of the trees. Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree. In particular, degrees of nodes (here degree means the number of children) are kept quite low: every node has degree at most O(log n) and the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the kth Fibonacci number.

criterion γ determines the goal of the scheduling problem and is usually some aspect of the time taken to process the jobs, such as finishing time or lateness. It will generally be:

1. C_{max} if indicates that the aim is to minimize the maximum time taken to complete any job;

2. $\sum_j C_j$ if indicates that the aim is to minimize the sum of processing times over all jobs;

3. $\sum_j w_j C_j$ if indicates that the aim is to minimize the weighted sum of processing times over all jobs.

The MRTA problem is in the class of scheduling problems described by

$$R||\sum_j w_j C_j,$$

i.e. the system is composed of unrelated parallel machines and the overall performance is computed as a weighted sum of the processing times for the individual tasks, where the processing time C_j is defined as K_{ij}, the cost expected from the execution of task j by the robot i to which j is assigned.

Even if such problem is known to be strongly NP-hard, it can be simplified by making two domain-specific observations.

1. Since MRTA is a degenerate *scheduling problem*, whereas in scheduling one must assign tasks to machines over time, only a single time-slot is considered;

2. The task weights can be directly incorporated into the cost estimates if we make the reasonable assumption that the task weights are known to the robots and can be used in cost estimation.

Hence given a cost estimate K_{ij} for robot i and task j and a scalar task weight w_j, a new weighted cost estimate $K_{ij}^{'}$ can be defined, where

$$K_{ij}^{'} = w_j K_{ij},$$

making the problem become

$$R||\sum_j C_j$$

which takes $O\left(n^4\right)$ time to be solved with Bruno, Coffman and Sethi's job scheduling algorithm [26].

2.7 Coalition formation problem

The MRTA problem can take a more complex and composite form which goes beyond the task allocation treated so far, i.e. the *coalition formation problem* [11], we see fit to be described in this chapter. In detail in a *coalition formation problem* a set of robots have to cooperate to accomplish a set of tasks (or roles), with the constraints that each robot can execute one task at time and robots can form coalitions to cooperate on specific tasks. Thus the overall performance of the formation can be better than the single performances of robots and the quality of the execution of a specific task depends both on the individual capabilities that each robot has for that task and on how the capabilities can be combined together.

More precisely, the problem can is composed of two sets:

1. The set of robotic agents $R = \{R_1, \dots R_n\}$;
2. The set of tasks (or roles) $T = \{t_1, \dots t_m\}$.

Each robotic agent R_i has capabilities to perform each task which are represented by a vector $S_i = \langle s_i^1, \dots s_i^m \rangle$, where $s_i^j \in \mathbb{R}$ is the level of performance that R_i an achieve when allocated to role t_j. Moreover each task t_j has a desired achievement level $l_j \in \mathbb{R}$ that needs to be reached by the agents accomplishing the task. The level of achievement of a task represents an objective of the whole system and will therefore be called the *system objective*, while each of the s_i^j can then be interpreted as the level of satisfaction that the agent will obtain if it is allocated to the task or *self evaluation* of the agent (it is an estimation that each agents computes of its capability to perform a task).

A coalition C is a set of agents, in particular C_j is the coalition of agents assigned to task t_j and the set $C = \{C_1, \dots C_m\}$ is a partition of R and represents the set of coalitions assigned to all tasks, hence we define $F(C, t_i)$ as the amount of work that agents in coalition C can perform task t_i when the coalition works on task t_i. Then if we define $V_i(C) = v_i \in \mathbb{R}^+$ as the utility that the system can gain when a coalition successfully accomplishes a task t_i i.e. the aggregation of the individual agents' *self evaluations* which indicates the total level of satisfaction that the agents inside the coalition have for task t_i, called *agent satisfaction*, the objective of this approach is then

$$\arg\max_{C} \sum_{i=1}^{m} V_i(C_i).$$

However coalition formation is usually a one-shot problem where coalitions values are known in advance, and once coalitions are formed and allocated to tasks, robots will simply carry on with their tasks. Indeed in the *Middle Size League* RoboCup domain robots have to deal with a more complex setting, because they have opponents which can change the situation very rapidly: the *self evaluation* that each robotic agent computes for each task will change over time as well as the value of coalitions, the priorities of tasks and the *system objectives* and some hardware problems can occur:

1. Robots may have wrong estimation of their *self evaluation*;

2. Robots might fail unexpectedly;

3. Messages communicated among robots might be lost, leading the team to have temporarily misaligned knowledge about the current situation.

Hereafter Algorithm 2.10 shows the pseudocode which represents a cooperative control method to address the described problem, whose handling follows the basic idea in which robots first evaluate and share the *self evaluation* for the roles that the robots can perform, then they compute the best allocation of coalitions to tasks deciding which roles should be executed. In particular such process is iterated over time in order to react from;

1. The environment dynamism;

2. The changes in task priorities;

3. Robot failures or malfunctioning.

In fact the algorithm is run at a predefined execution rate, which is specified according to the application domain, and at each execution all information required to run the algorithm is acquired by the robots through sensor perception or through communication. The assumptions underlying this method are the following:

1. Robots are able to compute their *self evaluation* for each role depending on their current state and the state of the environment;

2. Each robot can estimate the *self evaluation* for each role and each of their teammates;

3. The *system objective* is known to all robots;

4. Tasks to be allocated have priorities (task t_i has higher priority than t_{i+1}) which are known to the whole team.

Given the above assumptions, the cooperative control method includes three main steps:

1. Each robot computes the Self Evaluation value for each task;

2. Robots broadcast the computed Self Evaluation value, for each task, to all team members;

3. Using a greedy approach each robot computes the coalitions to allocate to each task based on the information received by teammates.

Algorithm 2.10 Agent Satisfaction task assignment

```
1. Input: Tasks, Agents, SystemObjectives, SelfEvaluations
2. Output: TaskToExecute
3. SortedTasks ← Sort Tasks given priority
4. while SortedTasks ≠ ∅ do
5.     Task ← Pop(SortedTasks)
6.     SortedAgents ← Sort Agents given Self Evaluation for Task
7.     AgentSatisfaction ← 0
8.     AssignedAgents(Task) ← ∅
9.     while AgentSatisfaction < SystemObjectives(Task) ∧ SortedAgents ≠ ∅ do
10.        AssignedAgents(Task) ← AssignedAgents(Task) ∪ Pop(SortedAgents)
11.        AgentSatisfaction ← Aggregate(AssignedAgents(Task))
12.    end while
13.    if AgentSatisfaction < SystemObjectives(Task) then
14.        AssignedAgents(Task) ← ∅
15.    else
16.        Agents ← Agents \ AssignedAgents(Task)
17.    end if
18.    if mySelf ∈ AssignedAgents(Task) then
19.        TaskToExecute ← Task
20.        return TaskToExecute
21.    end if
22. end while
23. return NoTask
```

In detail the proposed algorithm taking as inputs the tasks to be executed, the available agents (including the one executing the algorithm) and the desired achievement level for each task, returns as output the task which should be executed by the agent running the algorithm. Basically, it sorts the tasks according to their priority (line 3), and then for each task computes the best agent coalition for that task. The algorithm sorts agents according to their ability to fulfill the task (line 6), and then incrementally builds a set of assigned agents for that task: at each iteration it checks whether the achievement level of the task has been reached (line 9), by computing the current *agent satisfaction*, through the evaluation of the *Aggregate* function which sums the achievement values of the agents and computes the achievement level of the coalition. Hence the inner while loop terminates if one of these cases occurs:

1. The algorithm found a set of agents that satisfies the achievement level of the task;
2. Task is not achievable with the current available agents.

In the latter case the algorithm makes the set of assigned agents be empty (line 14), while in the former case it removes the assigned agents from the set of available agents (line 16). Before proceeding for the second task, each agent checks the set of the agent assigned to the current task:

1. If it belongs to this set, it terminates the algorithm execution and returns the task to execute;

2. Otherwise, it proceeds considering the following task.

However, in the case the agent is never assigned a special value, NoTask is returned (line 23).

Notice that the algorithm always terminates, because at each iteration of the inner while loop one agent is removed from the SortedAgents list, and in the while condition at line 9 the algorithm checks whether the SortedAgents list is empty. Therefore, at most the algorithm repeats the statements inside the while loop until all agents have been included in the coalition (similar reasoning holds for the outer while loop over tasks). Moreover, since allocated agents are removed from the list of available agents (line 16), agents allocated to one task will never be considered for another task, and thus we will never have an agent being part of two different coalitions. Finally, recalling that all agents have the same input data, because the set of tasks and the desired achievement levels are known a priori and agents communicate their Self Evaluation for each task, by executing the same algorithm the allocation to tasks will converge to a common solution.

Chapter 3

Implicit coordination

3.1 Task allocation strategies

It is well-known that the general problem of dynamically allocating tasks in a group of multiple robots satisfying multiple goals is yet unsolved. However, if this problem is viewed as an instance of dynamic task allocation under uncertainty, individualistic strategies [15], which do not involve explicit coordination and negotiation among the robots, can produce cooperative behaviors without explicit coordination, without guaranteeing optimality.

A framework, which is a general formulation for the MRS coordination problem, can be obtained exploiting the following decomposition of the task allocation problem:

1. Each robot bids on a task based on its perceived fitness to perform the task;
2. An auctioning mechanism decides which robot gets the task;
3. The winning robot's controller performs one or more actions to execute the task.

In this formulation, first a *bidding function* determines each robot abilities to perform each task according to robot states, next a *task allocation mechanism* determines which robot should perform a particular task examining their bids. Finally, considering their current task engagements, the robot controllers determine appropriate actions for each robot. This partitioning is shown in Figure 3.1 serves the purposes of reducing:

- The dimensionality of the coordination problem;
- The amount required for inter-robot communication.

In fact instead of mapping

$$S^{|R|} \rightarrow A^{|R|},$$

where S is the state space of a robot, $|R|$ is the number of robots, and A is the set of robot available actions, the function becomes

$$B^{|R||T|} \rightarrow T^{|R|},$$

i.e. a map from robot bids B for tasks T to a task assignment for each robot. Moreover the considered systems are Markovian, that is for a given robot R_i, the task allocation function:

1. Receives as inputs:
   (a) R_i's current task assignment;
   (b) Every other robot's current bid on each task;
2. Produces as output R_i's new assignment.

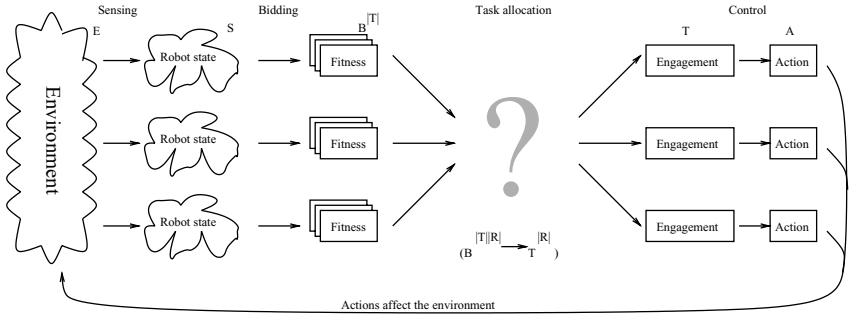


Figure 3.1: Dimensions reducing

However the overall mapping here is treated as a global and centralized process because the focus is on what the task allocation function should be, rather than on how it should be distributed. Indeed we get four task allocation strategies (see Table 3.1) by exploring the effects on performance of two key aspects of distributed control, commitment and coordination in their extreme cases:

1. No commitment and full commitment;
2. No coordination and full coordination.

Strategies	Coordination		
Commitment		Individualistic	Mutual exclusion
	Committed	Strategy 1	Strategy 2
	Opportunistic	Strategy 3	Strategy 4

Table 3.1: Task allocation strategies

Along the commitment axis, in a *fully committed* strategy a robot completes its assigned task before considering any new engagements, while a f*ully opportunistic* strategy allows robots to drop their ongoing engagements, at any time, in favor of new ones. On the other hand, along the coordination axis, the *individualistic strategy* makes a robot perform considering only its local information, while in a mutual exclusion strategy only one robot can be assigned to a task and no redundancies were allowed.

This concept of coordination is very simple and is not intended to represent explicit cooperation and coordination strategies: tested tasks are structured so that one robot is sufficient for completion of an individual task assignment, so that mutual exclusion can be the simplest yet effective form of coordination. Suppose to fill a table that results from listing each robot's current engagement and each robot's current bid on each task, hence Algorithm 3.1 shows a fully committed mutually exclusive strategy, where in case of individualistic (uncoordinated) strategies, the same steps are run on a separate table for each robot, while in the opportunistic (uncommitted) case, step 1 is skipped.

Algorithm 3.1 Committed mutually exclusive strategy

```
1  If a robot is currently engaged in a task and its bid on that task
       is greater than zero:
2    (a) Remove the row and column of the bid from the table;
3    (b) Set the robot's new assignment to its current one;
4  Find the highest bid in the remaining table.
5  Assign the corresponding robot to the corresponding task.
6  Remove the row and column of the bid from the table.
7  Repeat from step 2 until there are no more bids.
```

3.1.1 Alarm handling problem

In order to study and compare the task allocation strategies described above, a task domain can be used in simulations: this is the case of the *emergency handling* problem, in which robots roam around a planar environment and *alarms* occur at unpredictable times and in unpredictable locations. The task of the robot team is to detect alarms and fix problems they indicates, but there are the following costs:

1. A variable time-cost associated with traveling to an alarm, depending on:
   (a) The speed of the robot;
   (b) The distance from the alarm to the robot.
2. A fixed time-cost for fixing the alarm.

In the implementation represented by Figure 3.2, the situation is restricted to the case where any robot can fix any alarm. In this 10×10 grid three new alarms appeared every twelve time-steps at random positions, so that robots bid on them depending on their distance to those alarms, i.e. $20-d$, where d is

the Manhattan distance[1] to the alarm. At each time-step, any robot assigned to a particular alarm moves toward that alarm and when it arrives, because the fixed time-cost is 0, that alarm is instantly put out.

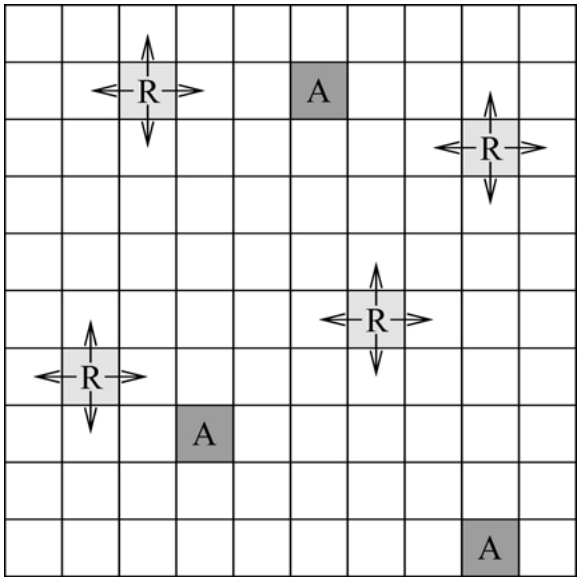


Figure 3.2: World grid representation

3.2 Belief communication

Another approach centered on implicit coordination is here applied to a typical coordination task from heterogenous robotic soccers, that is regaining ball possession [13]. Acquiring ball possession is a goal for the team as a whole, but all robots must agree upon which of the field robots will approach the ball. In order to infer the intentions of others, they first learn utility prediction models from observed experience: for the ball approach task, the utility measure is time, so the robots first locally learn to predict how long all robots will take to approach the ball, then they globally coordinate.

In a computational model for implicit coordination three components are necessary:

1. *Utility prediction* models;

2. Knowledge of the states of others;

[1]The Manhattan distance is a form of geometry in which the usual distance function or metric of Euclidean geometry is replaced by a new metric where the distance between two points is the sum of the absolute differences of their coordinates.

3. *Shared performance* model.

The *utility prediction* model makes all robots able to predict their own ball approach time, as well as that of the others. For instance, navigating to random targets on the field, robots can measure the time they took to approach them, and with these instances a model tree[2] is first trained, then by recursively partition the data, they fit them to linear.

However, predicting utilities for others can only be done if the robots have an estimation of the other robot's state, because, usually due to the vision system, it is often not possible to see all the teammates. Hence the task of the second component is making robots communicate their *belief states*[3] to each other to achieve more coherent and complete beliefs about the world, which are then used to determine their joint actions.

The last component is a locker-room agreement that only the quickest robot approaching the ball can enter, in tother words It is a flexible teamwork structure with inter-agent communication protocols which is accessible only to internal behaviors.

Now consider these three experiments used to evaluate learn prediction models and shared representations:

1. A dynamic environment experiment;
2. A static environment experiment;
3. A simulated experiment.

In the dynamic environment 2 robots continuously navigate to random targets on the field, for about half an hour, but the paths are generated such that interference between the robots was excluded. Each robot, 10 times per second, records:

1. Its own position and orientation;
2. The position an orientation of its teammate;
3. The position of the ball;
4. The predicted approach time for both robots.

Based on the above times, they choose which robot should approach the ball, even if they never actually approach it.

In the previous experiment, it is impossible to measure if the temporal predictions are actually correct, and if potential inaccuracies cause incorrect estimations. Therefore in the second experiment both robots navigate to different random positions and wait there, but the target to approach is fixed and the same for both robots and they are requested to record:

[2]Model trees are functions that map continuos or nominal features to a continuos value.

[3]This might seem contrary to a communication-free paradigm, but there is an important difference between communicating intentions and beliefs.

1. Their own state;

2. The state of their team mate;

3. The predicted approach times;

4. The actual approach duration to the goal position.

The log-files so acquired are almost identical to the ones in the dynamic experiment, with the difference they also contain the actual observed time for the robot. This static environment is less realistic, but allows comparisons between the predicted time with the actually measured time for each robot.

Regarding the simulated experiment, the set-up is identical to the dynamic one and the simulator allows to vary two variables that most strongly influence the success of implicit coordination: the communication quality and the field of view, which respectively vary from 100% (perfect communication) to 0% (no communication) and between 0 (blind) and 360 (omni-directional vision) degrees.The other robot and the ball are only perceived when they are in the field of view. Gaussian noise with a standard deviation of 9, 22 and 25 cm is added to the robot's estimates of the position of itself, the teammate and the ball respectively, which correspond to the errors observed on the real robots.

3.2.1 Experiments results

In the 96% of the dynamic experiments, robots agreed on which robot should approach the ball, while in the static experiment the chosen robot is actually the quickest one to approach the ball the 92% of the time. Moreover experiments, using only distance as a rough estimate of the approach time, although time is certainly strongly correlated with distance, led to significantly more incorrect coordinations: agreement is still very good (95%), but the robot that is really the quickest is chosen only 68% of the time.

The results of the simulation experiment depends on the quality of communication and the field of view (see Figure 3.3), where communication quality is the percentage of packets that arrive, and field of view is in degrees. The z-axis depicts coordination success, which is the percentage that only one robot intended to approach the ball.

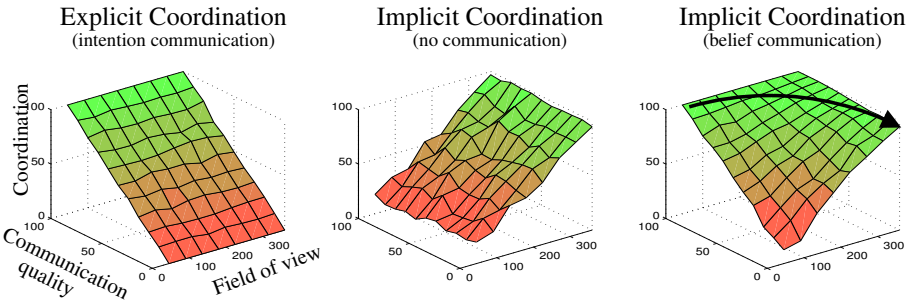


Figure 3.3: Simulation experiments

Since explicit coordination is based completely on communication, it is not surprising that it perfectly correlates with the quality of the communication, but is independent of the size of the field of view.

For implicit coordination without communication there is a correlation with the field of view, indeed if a robot is able to estimate the states of others better, it is able to coordinate better.

The third graph shows implicit coordination with belief state exchange, in which:

1. If the robot has another one in its field of view, it determines the other's state through state estimation;

2. Otherwise it uses communication (if possible) to exchange beliefs.

These states are then used to predict the utilities of others, independent if they were perceived or communicated, where the graph clearly shows that this approach combines the benefits of both.

3.3 Implicit coordination in RoboCup

CAMBADA is the *RoboCup* MSL soccer team of the University of Aveiro, Portugal. After the development of the team, started in 2003, it has participated in several national and international competitions, including RoboCup world championships, achieving excellent results in the mandatory technical challenge of the *RoboCup* MSL, e.g. the 1st place in 2009.

For instance Figure 3.4 shows their 2008 *RoboCup* MSL team whose robots basically follow a biomorphic paradigm. A robot is centered on a main processing unit, e.g. a laptop, which is responsible for the high-level behavior coordination, called the coordination layer. This processing unit:

1. Handles external communication with other robots;

2. Has a vision system directly attached;

3. Receives a low bandwidth sensing information;

4. Sends actuating commands to control the robot attitude by means of a distributed low-level system.

Figure 3.4: CAMBADA robots

Deliberation considerably relies on the concepts of role and behavior. Indeed at each time step the robots can take one the roles of Table 3.2, which actives some of these behaviors:

1. *bMove* that may activate the functions relative to:
   (a) Obstacles avoiding;
   (b) Ball avoiding;
2. *bMoveToAbs* that allows the movement of the player to an absolute position in the game field;
3. *bPassiveInter*, which first moves the player to the closest point in the ball trajectory, and waits there for the ball;
4. *bDribble* is used to dribble the ball towards a given relative player direction;
5. *bCatchBall* makes a robot receive a pass;
6. *bKick* that kicks the ball accurately to a position, either for shooting to goal or passing to a teammate;
7. *bGoalieDefend* is the main behavior of the goalie.

Kind	Roles
Positioning	RoleGoalie, RoleSupporter, RoleStricker
Passing	RolePasser, RoleReceiver
Free kicking	RoleToucher, RoleReplacer
Barrier	RoleBarrier

Table 3.2: CAMBADA roles

However also information sharing and integration are some of the key aspects in multi-robot teams, because sharing perceptional information in a team can improve the accuracy of world models and team coordination. Here the system uses an implicit coordination model based on notions like strategic positioning, role and formation, i.e. Each robot uses the information shared by the other robots to improve its knowledge about the current positions and velocities of the other robots and the ball.

3.3.1 Role assignment

The algorithm for role assignment is based on the absolute positions of the ball, the robot and its teammates, whose positions are not obtained through the vision system, but from the communicated information. Multi-robot ball position integration is used to:

1. Maintain an updated estimation of the ball position, when the vision subsystem cannot detect the ball;
2. Validate robot's own ball position estimation, when the vision subsystem detects a ball.

In other words as shown in Algorithm 3.2, when the robot does not see the ball, it analyzes the ball information of playing teammates.

Algorithm 3.2 CAMBADA ball detecting

```
1 Calculate the mean and standard deviation of the ball
    positions;
2 Discard the values considered as outliers of ball
    position;
3 Use the ball information of the teammate that has a
    shorter distance to the ball.
```

Moreover robots use a similar algorithm to determine if the robot sees a fake ball validating the robot's own perception. Communication is also used to convey the coordination status of each robot allowing robots to detect uncoordinated behavior and to correct this situation reinforcing the reliability of coordination algorithms.

Regarding implemented formations, they are sets of strategic positionings, that are movement models for a specific player, and are identified by three elements:

1. *Home position*, which is the target position of the player, when the ball is at the center of the field;

2. *Region* of the field, where the player can move;

3. *Ball attraction parameters*, used to compute the target position of the player in each moment using on the current ball position.

For instance, different *home positions* and *attraction parameters* allow a simple definition of *defensive*, *wing*, *midfielder* and *attack* strategic movement models.

The algorithms used for role and positioning assignment are based on considering different priorities for roles and positionings, so that the most important ones are always covered. Moreover these algorithms are separated and run at different rates: the former, based on its current world model, is decided locally by each robot, every cycle (40 ms), while the latter is decided by the coach and communicated to the robots every second.

The positioning assignment algorithm decides the place in the formation that each robot should occupy. Considering a formation with N positionings and a team of $K \leq N$ players (not counting the goalkeeper which has a fixed role) and N target positions (TP), Algorithm 3.3 take as inputs:

1. POS, an array of N positionings;

2. $BallPos$, the ball position;

3. PL, an array of K active players.

Then It returns as output the array of players PL. In depth, the algorithm consists of these main steps:

1. Calculate the distances of each of the robots to each of the target positions;

2. Assign the closest robot to the highest priority strategic positioning, which is in turn the closest to the ball;

3. Until all active robots have positionings assigned, from the remaining $K-1$ robots, assign the defensive positioning to the robot closest to that location.

In such away the robot assigned to the highest priority positioning will in most cases be locally assigned to *RoleStriker:* it do not move to that positioning and assures the stability of the assignment by placing itself close to the ball. Therefore after the *RoleStriker* role first defensive positionings are assigned, which are followed by the other supporter positionings.

Algorithm 3.3 CAMBADA positioning assignment

```
1  begin
2  clearAssignments(PL);
3  TP = calcTargetPositions(POS, BallPos);
4  for each POS[i] in descending order of priority
5    if there is no free player then return
6    p = the free player closest to TP[i];
7    PL[p].positioning = i;
8    PL[p].targetPosition = TP[i];
9  end for
10 end
```

Passing is one of the most important coordinated behavior involving two players, in which one kicks the ball towards the other, so that the other can continue with the ball. Until now, MSL teams have shown limited success in implementing and demonstrating passes.

In CAMBADA the player running *RoleStriker* may decide to switch to *RolePasser*, choosing the player to receive the ball, which in turn takes on the *RoleReceiver*. As described in Table 3.3, robots start from their own side of the field and, after each pass, the passer moves forward in the field, then becoming the receiver of the next pass (see Figure 3.5).

The coordination between passer and receiver is based on passing flags, one for each player, which can take the following values: ready, *tryingToPass* and *ballPassed*.

RolePasser	RoleReceiver
PassFlag TRYING_TO_PASS	
Align to receiver	Align to Passer
	PassFlag READY
Kick the ball	
PassFlag BALL_PASSED	
Move to next position	Catch ball

Table 3.3: CAMBADA coordinated action in pass

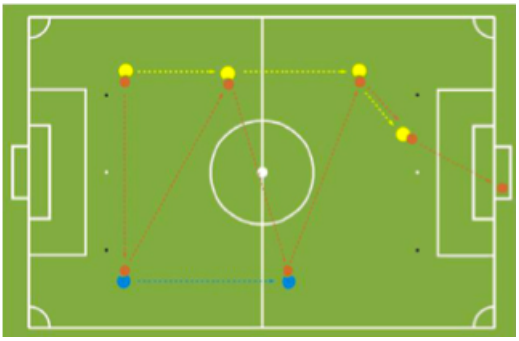


Figure 3.5: Sequence of passes

Other coordinated procedures regards the set plays[4]. In a toucher–replacer procedure (see Algorithm 3.4) the purpose of *RoleToucher* is to touch the ball and leave it to the *RoleReplacer*: the replacer handles the ball only after it has been touched by the toucher, allowing the replacer to score a direct goal if the opportunity arises.

Algorithm 3.4 CAMBADA corner kicks replacer role

```
1 begin
2 if I have the ball then shoot to opponent goal;
3 else if Ball close to me then move to Ball;
4 else if Toucher already passed Ball then catch Ball;
5 else wait that Ball is passed;
6 end
```

Another toucher–replacer procedure is used in the case of throw-in, goal kick and free kick set plays. In this situations, the toucher:

1. Approaches the ball;
2. Touches the ball pushing it towards the replacer until the ball is engaged by the replacer;
3. Withdraws leaving the ball to the replacer.

On the contrary, the replacer:

1. Moves towards the ball;
2. Grabs the ball;
3. Waits that the toucher moves away;

[4]Set plays are situations when the ball is introduced in open play after a stoppage, such as kick-off, throw-in, corner kick, free kick and goal kick.

4. Shoots to the opponent goal.

On the other hand in the case of opposer's set pieces, *RoleBarrier* is used to protect the goal from a direct shoot (see Figure 3.6). The line connecting the ball to the own goal defines the barrier positions:

1. One player places itself on this line, as close to the ball as it is allowed;
2. Two players place themselves near the penalty area;
3. One player is placed near the ball, 45 degrees from the mentioned line;
4. One player positions itself in such a way that it can oppose to the progression of the ball through the closest side of the field.

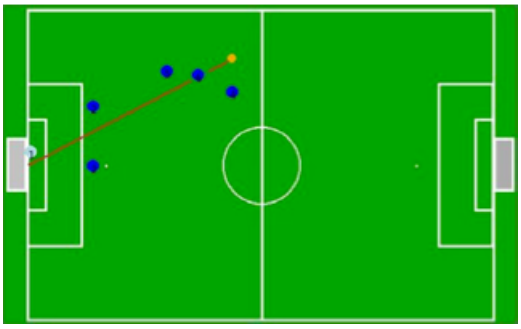


Figure 3.6: Placement of RoleBarrier players

Chapter 4

Heterogeneous architectures

4.1 Schema-based framework

Recalling that a behavior-based approach assumes a robot to be situated within its environment, and that a robot interacts with the world on its own, without any human intervention, thus its perspective is different from the observer's. Moreover, since robots are not merely information processing systems, their embodiments require that both all acquired information and all delivered effector commands must be transmitted through their physical structure.

Here given primitive behaviors are implemented with one motor schema[1], representing the physical activity, and one perceptual schema which includes sensing. The resulting architecture, whose reactive/deliberative trade-off stems from its hierarchical organization of its behaviors, is hybrid [7]: each behavior is implemented at some level, i.e. k, and can use perceptual schemas coming from the underlying $k-1$ level, eventually triggering one selected behavior at that level. The overall architecture is organized at 7 levels of abstraction shown in Figure 4.1 and here itemized from the lowest to the highest one:

- *Perception*;
- *Reactive*;
- *Implicit coordination*;
- *Individual goal triggering*;
- *Dynamic role assignment*;
- *Deliberative*;
- *Learning*.

[1] According to schema-based theories, a schema is a generic template for doing some activity which is parametrized, that is a schema composed of the basic units of behavior to construct basic actions.

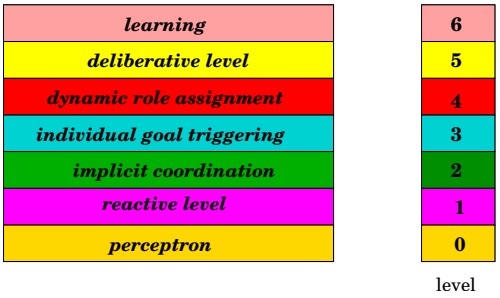


Figure 4.1: Behavior hierarchical organization

4.1.1 Ball exchanging

As came out so far, a pure reactive level would fail to provide a robot team with the required cooperation capabilities, because of the lack of some sort of mechanism, which allows the behavior of each individual robot to take into account other robots' behavior. Even a coordinated behavior among a group of robots based only on implicit communication could fail to exhibit collective behaviors, because implicit communication in itself does not guarantee cooperation.

The solution to the more general problem of making a collective behavior emerge from the individual behaviors of a group of robots depends on two different conditions that must be true at the same time: the first concerns the ability of any robot to recognize the circumstances under which it can be engaged in a collective behavior, while the second requires that those circumstances become effective. Hence, in this hybrid multi-level architecture, two intermediate levels have been provided to allow robots to communicate:

1. The lower implements *implicit communication*;

2. The higher deals with the *dynamic role exchange*.

Dealing with the reactive level, the former provides the necessary conditions, evaluated from the environment and specified patterns, to be verified to start cooperation, while the latter is devoted to examine and schedule the behaviors which are the best candidates for cooperation. In particular, when an individual robot succeeds in recognizing a distinguishing configuration pattern in the environment, it tries to become a *master* of a collective action indexed by that pattern. This can occur because at reactive level some conditions forces the estimation of a given quality function to evaluate over a fixed threshold. However different individual robots could evaluate over it, so that the method to acquire a master role is based the temporal ordering by which individuals try to notify the other teammates also wishing to become master.

As coordination task, consider the example two robots which try to carry the ball towards the opponent goal, passing and eventually defending it from opponents' attacks. Because robots are required to play well-specified roles, the

master role is assigned to the robot chasing the ball, whereas the other can be considered the *supporter*. Let *clampmaster/clampsupporter* be the implemented behaviors associated, where:

1. One robot is able to acquire and then to advocate a master role, showing a dominant role in the clamp action by chasing the ball;
2. The other robot is committed to acquire a supporting role in the clamp action while it is approaching the ball.

As just stated and shown in Algorithm 4.1, clampmaster and clampsupporter are complementary behaviors that must be arbitrated.

Algorithm 4.1 Clampmaster/Clampsupporter behaviors

```
1 Begin clampsupporter
2 if !acquire(master) & canBe(supporter) then assume(supporter);
3 if assume(supporter) & notify(supporter) then acquire(supporter);
4 End clampsupporter

1 Begin clampmaster
2 if haveBall(me) & !haveBall(mate) then acquire(master);
3 if acquire(master) & notify(master) then advocate(master);
4 End clampmaster
```

The basic rule is that a *master* role must be advocated whereas the supporter role should be acquired, so that two reciprocity rules are required, i.e. only if provided that a notification is made to the referred teammate:

1. A role is switched from *acquire* to *advocate;*
2. A role is switched from *assume* to *acquire.*

Such rules imply a direct communication between teammates to assign the role on the first notified/first advocated basis, where the robot carrying the ball advocates the *master* role for its-self and commit the teammate to acquire the supporter role. By doing so, both robots issues the behavior *haveBall*, while the former also issues the behavior *chaseBall*, whereas the latter exhibits the behavior *approachBall*.

4.2 Hybrid automata

A framework can capture both the discrete and continuos dynamics of hybrid systems [8], making possible to model cooperative tasks and dynamic role assignment in MRS. Here for each robot, *hybrid automata*[2] makes the mechanism for coordination is completely decentralized by representing for each robot:

[2]In automata theory, a *hybrid automaton* is a mathematical model for precisely describing systems, in which computational processes interact with physical processes. The behavior of a hybrid automaton consists of discrete state transitions and continuous evolution.

1. Roles;
2. Role assignments;
3. Continuos controllers;
4. Discrete variables.

indeed a robot has its own controllers and takes its own decisions based on local and global information; the former consists of the robot's internal state and its perception about the environment, while the latter contains data about the other robots and is normally received through explicit communication. Thus each team member has to explicitly communicate with other robots to gather information but normally they do not need to construct a complete global state of the system for the cooperative execution. For instance, part of the information necessary to role assignment, is obtained a priori, before the start of the execution (e.g. the information concerning the task), while the rest of information is obtained dynamically during the task execution.

There are three ways of changing roles during the execution of a cooperative task:

1. *Allocation*, in which a robot assumes a new role after finishing the execution of another one;
2. *Reallocation* process, in which using an utility function, a robot interrupts the performance of one role and starts or continues the performance of another role;
3. *Exchange*, in which two or more robots synchronize themselves and swap their roles.

Differently from the approaches described so far, this one allows for two types of explicit communication, the synchronous and the asynchronous one. Recalling that the former usually consists of messages sent and received continuously at a constant rate, while the latter permits interruptions when messages are received, synchronous messages are important in situations where robots must receive constant updates, on the other hand, the asynchronous ones are used when unexpected events occurs.

Let here describe in a more formal way the framework representing this architecture. First of all a MRS can be described by its state X,

$$X = [x_1, x_2, \dots x_n]^T ,$$

that is a concatenation of the states of the individual robots and varies as a function of its continuous states $\{\overset{n}{\underset{i=1}{x_i}}\}$ and its inputs vector $\{\overset{n}{\underset{i=1}{u_i}}\}$. Considering that robot i can also receive $\hat{z}_i$, i.e. the approximated information given by the rest of the system, and that it can be controlled according to its assigned role q, the state equation can be defined as

$$\dot{x_i} = f_{i,q}\left(x_i, u_i, \hat{z}_i\right).$$

But robot i is associated with a control policy

$$u_i = k_{i,q}\left(x_i, \hat{z}_i\right)$$

and $\hat{z}_i$ is a function of the state X, hence the state equation can be written as:

$$\dot{x_i} = f_{i,q}\left(X\right),$$

or, for the whole team

$$\dot{X} = F_q\left(X\right),\ F_q = \left[f_{1,q}, \ldots, f_{n,q}\right]^T.$$

These equations, which model the continuous behavior of each robot and consequently the continuous behavior of the team, in turn are modeled by a *hybrid automaton*. It is a finite *automaton*, whose arguments are a finite number of real-valued variables that change continuously and that can be formally defined as

$$H = \left\{Q, V, E, f, Inv, G, Init, R\right\},$$

where:

- $Q = \{1, 2, \ldots k\}$ is the set of discrete states, called *control models*;
- $V = V_d \cup V_c$ is the set that represents the discrete (V_d) and continuous (V_c) variables of the system;
- f is the function which describes the dynamics of the continuous variables;
- E is the set of *control switches* of discrete transitions between pairs of control modes;
- Inv is the set of *predicates*[3] related to the *control modes*;
- G is is the set of *predicates* related to the *control switches*;
- $Init$ is the initial states of the system;
- R is the set of re*set statements* for *control switches*.

Each role is a control mode of the h*ybrid automaton*, where:

1. Internal states and sensory information can be specified by continuous and discrete variables;
2. Messages are sent and received in discrete self transitions through communication channels.

As the role assignment is represented by discrete transitions, where the invariants (Inv) and guards (G) define when each robot will assume a new role, the cooperative task execution can be modeled by a parallel composition of several *hybrid automata*.

[3]In mathematics, a predicate is commonly understood to be a boolean-valued function $P : X \to \{true, false\}$, called the predicate on X.

4.3 Resources constraints

When designing a MRS performing complex tasks in a dynamic environment in addiction to a dynamic assignment of roles constraints on resources that are accessed by the robots have to take into account. However very often cooperation through dynamic task assignment and coordination on the access to shared resources are often treated, separately. The former problem is faced by splitting the tasks and assigning each robot a role without directly supporting the coordination in the access to shared resources. On the other hand the latter problem focuses on the techniques for handling the conflicts arising from the attempt to access common resources, where resource conflicts among robots can be solved:

1. By combining the plans of each robot, and producing a coordinated plan;
2. Using task networks indicating dependencies among the tasks to be executed;
3. Using ad hoc approaches, such as space partition in foraging task.

Here it is proposed the design and realization of a MRS that takes into account at the same time dynamic role assignment and constraints on resources [9]. The approach keeps the requirements on each robot to a very abstract set of functionalities: it is not needed an explicit synchronization which is integrated within the information acquisition capabilities of the robots, hence making the implementation easier with heterogenous MRS.

The focus is on the *Sony Legged Robot League,* an highly successful Four-Legged League, based on Sony's AIBO dog robots, now replaced by the Standard Platform League based on Aldebaran's Nao humanoids (see Subsection 1.1). Indeed in the set of game rules, the *two-defender* rule prohibits the simultaneous presence of two players in the goal area, introducing a new challenge for coordination. It turns out that this rule gives rise to a scenario where the dynamic exchange of roles in not sufficient for effective performance, while a specific constraint on the access to a shared resource (the goal area) must be properly taken into consideration. The basic intuition underlying the proposed solution is to treat the role assignment as a technique for establishing the goal of each individual player, where the need of synchronization with other players must be explicitly addressed in the selection and execution of the plan devised by the player to achieve the assigned goal.

The framework here proposed is based on a hybrid robot architecture, made up of two main layers:

1. The *Operative Level*;
2. The *Deliberative Level*, which is in turn made up of:
   (a) An *On-Line Deliberative SubLevel*;
   (b) An *Off-Line Deliberative SubLevel*.

The former is based on a numeric representation of both the information acquired by the robot sensors and the data concerning the current task. The latter is based on a symbolic representation of both the information acquired by the robot sensors and the data concerning the task to be accomplished. In fact the *On-Line Deliberative SubLevel* is in charge of evaluating data during the execution of the task, while the *Off-line Deliberative SubLevel* is executed off-line before the actual task execution.

In detail, the *deliberative level* relies on a representation of the robot's knowledge about the environment (it is provided off-line and acquired during task execution) and it is formed by two main components:

1. A *Plan Execution Module*, executed on-line during the accomplishment of the robot's task, responsible for executing a plan by coordinating the primitive actions of a single robot;

2. A *Plan Generation Module*, executed off-line before the beginning of the robot's mission, generates a set of plans to deal with some specific situations.

During the execution of a plan, the *Plan Execution Module* checks for the conditions that guarantee the applicability of the current plan in the current situation (provided by the high-level state), and, if the current plan is no longer executable, it selects a new plan from the library.

A plan is represented as a transition graph, where:

- Each node denotes a state and is labeled with the state properties;

- Each arc denotes a state transition and is labeled with the action that causes the transition.

A state represents a situation the system can be in and is characterized by a set of properties which give a description of the situation. Actions are represented using preconditions and effects. Preconditions are the conditions that are necessary for activating the action and indicate what must be true before the action is executed, that is they specify circumstances under which it is possible to execute an action. Effects are the conditions that must hold after the execution of the action and characterize how the state changes after the execution of the action. Sensing actions are associated with conditions to be verified, because depending upon the runtime value of these conditions, a different part of the plan will be executed.

4.3.1 Two-defender rule protocol

A typical situation, represented by Figure 4.2 in when the goalkeeper (robot 1) is moving away from its own goal and is approaching the ball to push it away, while robot 2 is far away from the ball and it cannot help the goalkeeper immediately. It is more convenient for the team that robot 1 takes the role of attacker pushing the ball toward the opposite goal, while robot 2 goes back to

defend its own goal acting as a goalkeeper. However in performing this role exchange the two robots must comply with the two defenders rule, thus robot 2 can enter the goal area only after robot 1 has left it.

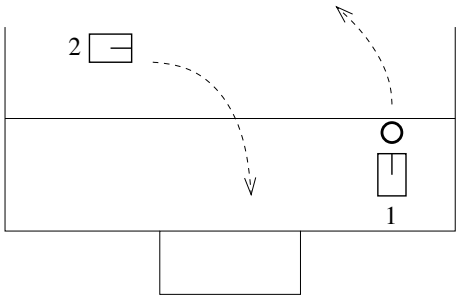


Figure 4.2: Two-defender rule

In this situation the utility functions for the role *attacker* and *goalkeeper* are defined as functions of information of the robot about the environment. The utility function for the attacker is based on the robot position and orientation and its distance to the ball and computes an estimation of the time needed to reach the ball in the direction of the opponent goal. In other words

$$f_{attacker} = -\alpha \cdot trajectorToBall - \beta \cdot |dirToOpponentGoal| + \ldots + Hysteresis,$$

where α and β are positive coefficients to be determined by experiments because can be different among heterogeneous robots. On the other hand the utility function for the goalkeeper is based on its position in the field and evaluates an estimation of the time needed to reach the goal facing the opponent goal, i.e.

$$f_{goalkeeper} = -\alpha \cdot distToMyGoal - \beta \cdot |dirToOpponentGoal| + \ldots + Hysteresis.$$

The problem of complying with the *two-defender* rule is solved by generating a plan in which one robot before entering the goal area must check if it is free. This is achieved by adding in the knowledge base of the robot that is taking the role of goalkeeper the following specification (see Table 4.1) for the actions:

- *GotoAreaLine*;
- *SenseFreeArea*;
- *GotoGoal*.

PosAreaLine represents the robot positioned close, but outside, the area line, *FreeArea* denotes the area being free from robots of its own team and *PosGoal* states for the robot being in the goal area.

Action	Preconditions	Postconditions	Effects
GotoAreaLine	NOT PosGoal	-	PosAreaLine
SenseFreeArea	PosAreaLine	-	FreeArea
			NOT FreeArea
GotoGoal	PosAreaLine AND FreeArea	PosGoal	-

Table 4.1: Goalkeeper specifications

The portion of the plan of interest for coordination is generated by an automatic plan generation system. As shown in Figure 4.3, the plan contains a while loop in which the robot waits for the condition FreeArea to become true before entering the area, allowing for synchronization of the actions between the two plans executed by the robots.

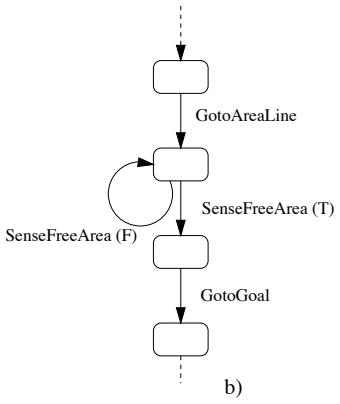


Figure 4.3: Cyclic conditional plan

The actions executed by the robots are implemented in the *operative* layer of the software architecture. The implementation of the action *SenseFreeArea* is obtained by means of explicit communication among the robots, which:

1. Broadcast a set of information about their state, including their position according to all the other teammates;

2. Check among the positions received from all the teammates that no one is inside the defense area.

When no data from a robot are acquired, *SenseFreeArea* is implemented in order to return *false*, when data are received from other robots, the action evaluates their positions, otherwise it is obviously possible to implement the sensing action used for synchronizing the robots with other techniques, e.g. the vision system.

This dynamic role assignment is tested in situations represented by Figure 4.2, where robot 1 is initially assigned to a defensive role and its position is

inside the goal area, while robot 2 start position is outside the goal area and is assigned with a non defensive role. When the ball is positioned in front of the robot 1 a role change occurs:

- Robot 1:
  - Takes the attack role;
  - Begins to push the ball toward the opponent goal escaping the goal area.
- Robot 2:
  - Takes the defender role;
  - Starts to go in the defense position but it does not enter the area until the other robot leaves it.

4.4 Interaction Nets

There are other ways to create such frameworks that model distributed multi-agent systems, for instance firstly by a graphical modeling of behaviors in form of hierarchical Interaction Nets [10] and secondly by the execution of decision trees described in *XABSL*[4] language. Here the used world model incorporates mechanisms for multi-thread save data representation, data processing and communication and each robot holds several data containers:

1. One for the locally sensed or derived data;
2. One for each robot of the team.

However only a part of locally available information of a robot is communicated to others, which use it to estimate abstract world states, i.e. application relevant features of the environment. Due to autonomy of the robots, they make their decisions locally and model behaviors by considering all available data which includes communicated status information.

The subsystem for team play is divided into two layers. One layer, called *WorldModelAdapter*, is derived from the world model and is responsible for processing the data and providing results (e.g. data sets, numerical data, boolean flags) in a format suitable for the second layer. Due to fault tolerance, it runs locally on each robot so that the characteristics of the current game are derived from a merged view of the distributed system of robotic agents. The second and higher layer constitutes the control of the of the behavior of an agent, where a behavior is a state of an agent in which certain drive command are executed.

Let here describe formally the *Interaction Nets*, whose parameters consist of:

[4]XABLS (eXtensible Agent Behavior Specification Language) is a specification language for agent behavior which can be directly executed by an execution engine.

1. A set $Ph = \{Ph_1, \ldots, Ph_n\}$ of phases, which represent the states of the agent, but imply the execution of a basic behavior;

2. A set of arcs from phases to transitions and from transitions to phases;

3. A token which indicates the current phase;

4. A set of transitions $Trans : Ph \rightarrow Ph$ which define preconditions and postconditions of a phase (to switch from Ph_n to Ph_{n+1}, Ph_n must hold the token and the transition conditions must be satisfied).

However modeling the behavior of robot agents can reach a high level of complexity with a large number of phases and transitions, hence in order to keep the system manageable, hierarchical nets are introduced, where subnets $SN \subseteq Net$ are handled like common phases. In particular such subnets have a transition for its preconditions and one for its postcondition and a certain process can leave them only after it reaches their end phases,then being free to continue on the level above with the post condition of the subnet.

4.4.1 Passplay

Remembering that all robots communicate with each other, it is possible to assign tactical roles and subroles to the different robot players. Every robot has a *tactical role* and a unique *tactical subrole* during the match, which will not be changed unless other robots fails. Here there are two main tactical roles, *Defender* and *Forward*, which are re divided into subroles in such a way they do not conflict with each other and define complementary behaviors at the same time. The advantage of the chosen tactical role hierarchical approach is a better organization of the team coordinating mechanism.

The implementation uses two ordered lists:

1. A list (*roleList*) contains the dedicated tactical roles;

2. A list (robotList) contains the available robots (robots that break down during a match are automatically removed from this list).

The tactical roles a reassigned to the robots one-on-one, depending on the order in the role list. Tactical subroles are assigned depending on the number of robots that occupy a tactical role, because every robot calculates its tactical role and subrole locally using a common algorithm. Moreover special roles exist, such that if a robot possess one of these roles, it can execute the behavior of the special role keeping its tactical role and subrole.

During the running game there are two strategies, the offensive play and the defensive play, where each strategy defines a proper set of behaviors according to the situation. The advantage of such an approach is that each role can make use of different behaviors depending on the current situation in the match, which results in a situation sensitive architecture. The *Offensive* and *Defensive* subnets are subdivided into further subnets that handle the behavior for the

different tactical roles and subroles. For instace in the subnets for tactical subroles, the behavior *PassPlay* is implemented (see Figure 4.4).

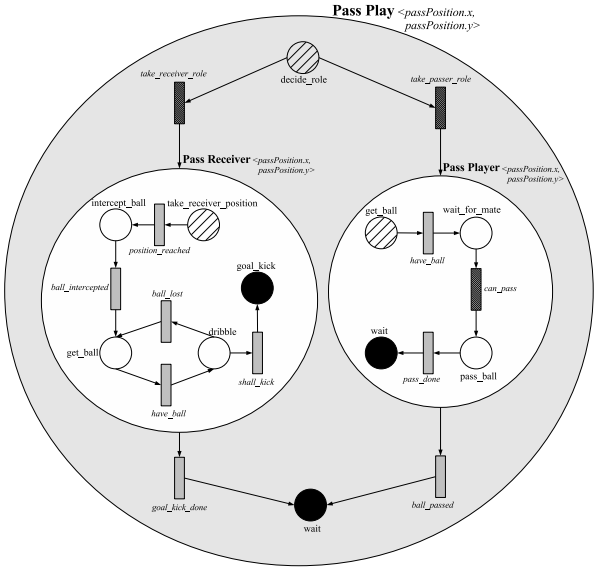


Figure 4.4: Passplay

Here, a pass is done between two players with a fixed position of the pass receiver. This intera*ction net* requires the participation of two agents, one taking the special role of a *pass player* and another the special role of *pass receiver*. The two participating agents start in the initial phase *decideRole*, where the decision of which player is assigned to which role is taken. By the function iAmNearestTo a player can calculate the current position compared to the position of the teammates relative to the distance to a given arrival point. Let here summarize the main step required:

1. The agent that has the closest position to the ball takes the special role *pass player*;

2. The agent that has the closest position to the defined pass target position takes the special role *pass receiver*;

3. The *pass player* approaches the ball oriented in the direction of the pass target;

4. The *pass receiver* drives towards the pass target position;

5. As soon as the *pass player* has the required position relative to the ball an action synchronization between the two agents takes place;

6. Only if the *pass receiver* moves into a range of tolerance around the target position the execution of the *pass play* is continued;

7. The *pass receiving* agent changes to the behavior *interceptBall* as soon as it reaches the pass target position;

8. As soon as the *pass receiver* agent gets the ball or the ball stops in a position very close to the *pass receiver*, it tries to shoot a goal directly.

The introduced interaction only contains three transitions that result in an actions coordination: the input transition of the behavior *decideRole* results in a complementary role assignment of the cooperating agents, while the transition *canPass* results in a time synchronization of the action phase.

Part II

Coalition formation for task assignment in multi-robot system

Chapter 5

Coalitions satisfaction

5.1 Problem statement

Consider the *alarm handling problem* described in Subsection 3.1.1, where alarms arise in unpredictable locations at unpredictable times, then we consider this dynamic and uncertain environment to create role allocation instances.

Hence given a robots team represented by the set $\mathbf{R}_n = \{R_1, R_2, \ldots, R_n\}$ and a set of roles, i.e. $\mathbf{r}_m = \{r_1, r_2, \ldots, r_m\}$, $m \gtreqless n$, that have to be assigned to each robot, we can cast a role allocation problem into a task allocation problem by replacing $\mathbf{r}_m$ with $\mathbf{T}_m = \{T_1, T_2, \ldots, T_m\}$ defined as a set of tasks that robots have to accomplish.

For example in the environment of Figure 5.1, given four robots, i.e. R_1, R_2 , R_3, R_4 and three tasks, i.e. T_1, T_2, T_3 Figure 5.1 represents a possible task assignment. In particular both robot R_1 and R_3 cooperate to accomplish task T_2, while tasks T_1 and T_3 are assigned to robots R_2 and R_4, respectively.

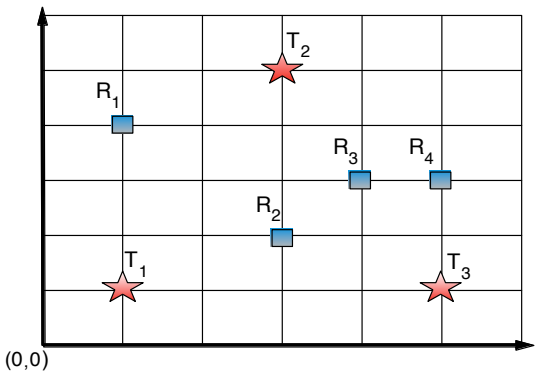


Figure 5.1: Instance of the task allocation problem

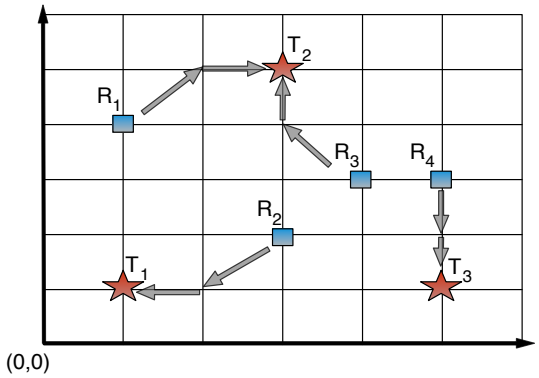


Figure 5.2: Solution of the task allocation problem

Let here explain how robots and tasks are represented. First each robot R_i is described as an agent whose position within the environment is represented by a quaternary $R_i^{pos} = (x, y, \theta)$, where:

1. x, y are the cartesian coordinates;
2. θ is the orientation angle.

A robot also knows the position and relative orientation of all the available tasks.

As for each robot the quaternary $T_j^{pos} = (x, y, \theta)$ refers to the position and orientation of task T_j. Moreover, each robot can provide a *service time* equal to R_i^{ser}, while a task possesses two parameters:

1. The *deadline service time* T_j^{ser}, that is the time after which the task leaves the environment;
2. The *satisfaction service* T_j^{sat}, that is the minimum time of service required by the task in order to be satisfied.

Each robot R_i can accomplish task T_j if the following conditions are both satisfied:

1. R_i has to reach T_j before it expires,i.e. the arrival of the robot is after T_j^{ser};
2. R_i has to satisfy T_j for at least T_j^{sat}.

This concept of satisfaction leads to the definition of an utility function $f_i^j(\cdot)$, which makes a robot R_i able to compute the time of service it can give to task T_j, and from now on we refers to this function with the name of *A function*.

5.2 Utility functions

Q function (see Subsection 1.2.3) has inspired the development of *A function*. However, considering that the proposed scenario is far from being similar to RoboCup competitions, some parameters of the *Q function* are not considered, while other ones are introduced. For instance, according to the *Q function* design, a task can take the role of the ball such that the common parameters with the *A function* are the distance d between the robot and the task and the robot approach orientation θ to the task.

Hence the *A function* can be defined as

$$f_i^j\left(R_i^{pos}, T_j^{pos}\right) = T_j^{arr} \in \mathbb{R}_0^+, \tag{5.1}$$

where the *arrival time* T_j^{arr} represents the estimated time that robot R_i spends to reach task T_j placing in front of it. Given a fixed robot scalar velocity v_i^{scalar} and rotational velocity $v_i^{angular}$, the position of the task T_j^{pos} combined with the position of the robot R_i^{pos} are responsible for the T_j^{arr} computation, which fundamentally evaluates:

1. The Euclidean distance d_i^j between T_j and R_i;
2. The angle of rotation θ_i^j which R_i has to perform to head T_j;
3. The distance m_i^j which R_i has to carry out to face T_j.

Then, according to robot velocities, the procedure uses the obtained parameters to get an estimated time.

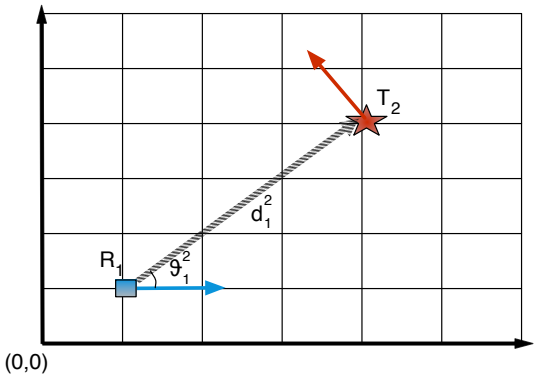


Figure 5.3: Euclidean distance and angular rotation

As shown in Figure 5.3, the function (see the pseudocode in Algorithm 5.1) first evaluates the Euclidean distance between T_2 and R_1 (line 2), then the robot minimum rotation θ_1^2 (lines 3-5).

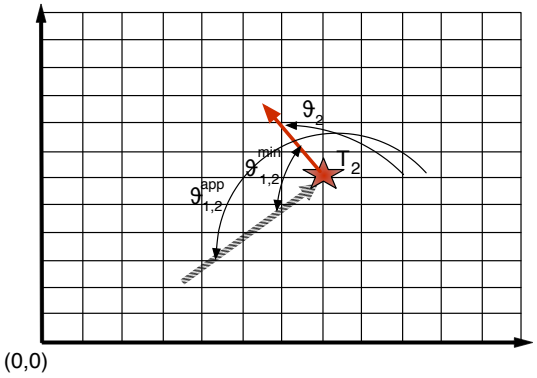


Figure 5.4: Minimum angular

Next, the robot position is also used to compute the angle $\theta_{1,2}^{app}$ (see Figure 5.4), relative to the task, which is supposed to be the robot approaching angle (line 6) . This angle combined with task orientation θ_2 permits to get $\theta_{1,2}^{min}$, that is the minimum angular gap robot has to carry out (lines 7-8).

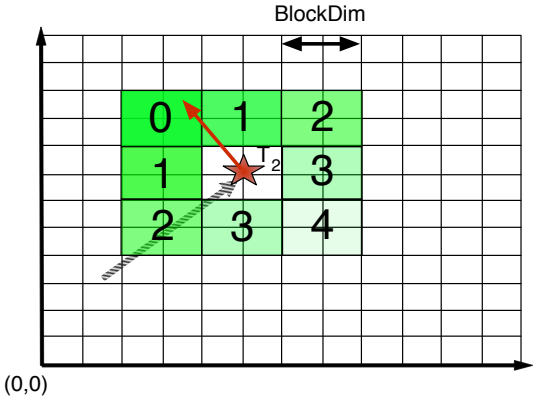


Figure 5.5: Manhattan distance approach

In order to transform this angle into distance m_1^2 we introduce a *manhattan distance* approach (see Figure 5.5). In other words an area is created around the task, which is on its center, and divided to create blocks of $blockDim^2$ area. Since the widest *angular gap* $\theta_{i,j}^{min}$ is equal to π we can obtain an estimation of the covered space by:

1. Computing of $radianForBlock = \frac{\pi}{maxBlockNum}$, that is the number of radians per block element (line 9);
2. Evaluating $blockNum = \left\lceil \frac{minimumGap}{radianForBlock} \right\rceil$, that is the number of block

elements necessary to cover the *minimum angular gap* (line 10).

Finally, after evaluating the total distance $d_{1,2}^{total}$ the robot has to achieve (see Figure 5.6) by summing the Euclidean distance d_1^2 to the *task rounding* distance m_1^2 (lines 11-12), the estimated time is that the robot spends to rotate a θ_1^2 angle at $v_1^{angular}$ velocity and cover a $d_{1,2}^{total}$ distance at v_1^{scalar} velocity. For example such approach could fit into a museum environment, where robot acts as tour guides and have to reach visitors that are looking for some objects d'art, so that a robot frontal appearance is preferred even at the cost of spending more time on coming up.

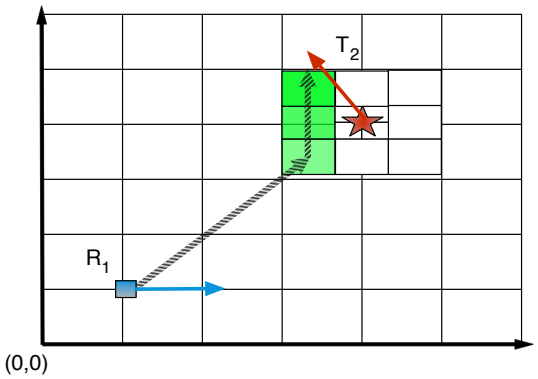


Figure 5.6: Estimated path

Algorithm 5.1 Arrival time T_j^{arr}

```
1  timeToArrive(taskPos){
2     eucliDist = getDistance(taskPos);
3     angularDrive = getDrive(robotPos,taskPos);
4     angularAdjust = abs(robotDrive - robotPos.theta);
5     minimumAdjust = minimizeGap(angularAdjust)
6     taskApproach = getApproach(robotPos);
7     angularGap = abs(taskPos.theta - taskApproach);
8     minimumGap = minimizeGap(angularGap);
9     radianForBlock = pi / maxBlockNum;
10    blockNum = ceil(minimumGap / radianForBlock);
11    taskRoundDist = blockNum * blockDim;
12    distToTask = euclidDist + taskRoundDist;
13    rotationTime = robotAdjust / angularVel;
14    distTime = distToTask / scalarVelocity;
15    return   distTime + rotationTime;}
```

For instance, if in Figure 5.1 we consider only robots R_1, R_3 and task T_2 we get Figure 5.7, where:

1. $R_1^{pos} = (1, 4, 0)$;
2. $R_3^{pos} = (4, 3, 0)$;
3. $T_2^{pos} = (3, 5, 2.26)$.

Then we evaluate *A functions* $f_1^2\,(R_1^{pos}, T_2^{pos})$ and $f_3^2\,(R_3^{pos}, T_2^{pos})$, where the scalar and rotation velocity are respectively $0.7\,^{\mathrm{m}}/_{\mathrm{s}}$ and $2.09\,^{\mathrm{rad}}/_{\mathrm{s}}$, whose results are shown in 5.1. Even if both robots have the Euclidean distance equal to 2.23 meters, their arrival time differ more than 3 seconds: this is due both to their orientation angle which implies different rotational times and the orientation angle of the task which is in favor of robot R_1 in terms of *rounding* distance.

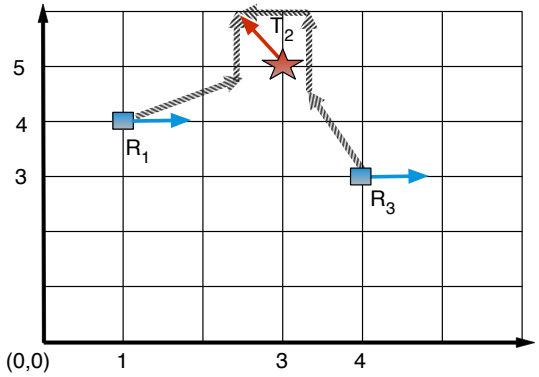


Figure 5.7: A-function examples

	R_1	R_3
euclidDist $[m]$	2.236	2.236
angularDrive [°]	26.56	116.50
angularAdjust [°]	26.56	116.50
minimumAdjust [°]	26.56	116.50
taskApproach [°]	206.56	296.56
minimumGap [°]	76.50	166.50
blockNum	2	4
taskRoundDist $[m]$	2	4
distToTask $[m]$	4.236	6.236
rotationTime $[s]$	0.221	0.973
distTime $[s]$	6.051	8.908
timeToArrive $[s]$	6.273	9.882

Table 5.1: A-function examples results

5.2.1 Coalition utility function

Inspired by the *coalition formation problem* (see Section 7.1.1), given a set of robots $\mathbf{R}_n$ and a set of tasks $\mathbf{T}_m$ as described in Section 5.1 the *self evaluations* of robot R_i are represented by vector $S_i = \langle s_i^1, s_i^2, \ldots, s_i^m \rangle$, where $s_i^j = f_i^j \left(R_i^{pos}, T_j^{pos} \right)$, while T_j^{sat} and T_j^{ser} are the *system objectives* relative to task T_j.

Recalling that a coalition C_j is the set of agents assigned to task T_j, the set $\mathbf{C} = \{C_1, \ldots, C_m\}$ is a partition of R_n and represents the coalitions assigned to all tasks. For instance in Figure 5.8 a possible coalition $\mathbf{C} = \{C_1, C_2, C_3\}$ is $C_1 = \{R_2\}$, $C_2 = \{R_1, R_3\}$, $C_3 = \{R_4\}$.

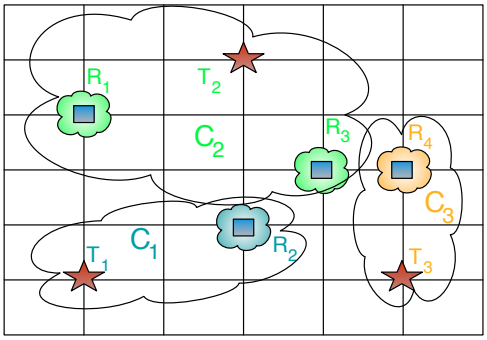


Figure 5.8: Coalition formation problem

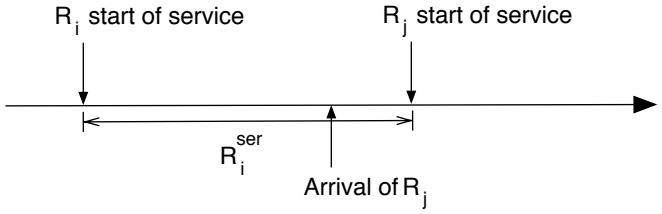
Given a coalition C and a task T_j, let here define $F\left(C, T_j\right)$ as the *total amount of service* R_C^{ser} the coalition gives to the task before it expires. The evaluation of R_C^{ser} is computed through the following steps:

1. **START**;
2. Compute the *arrival time* T_j^{arr} for each robot $R_i \in C$;
3. Sort these times in an increasing order;
4. Set the *service start time* S^{ser} with the R_i smallest *arrival time* and remove it;
5. Set the *service end time* E^{ser} and the new *service start time* with $S^{ser} + R_i^{ser}$;
6. Add R_i^{ser} to the *total amount of service* R_C^{ser};
7. If it turns out that E^{ser} has exceeded the *deadline service time* T_j^{ser}, reduce R_C^{ser} by the surplus **END** else **CONTINUE**;
8. If there are other robots:

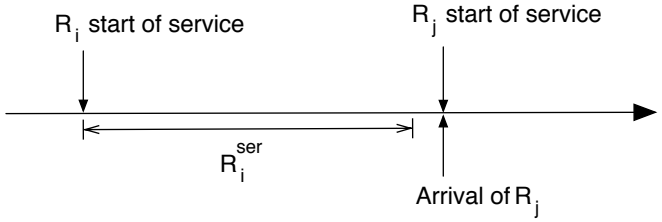
(a) Update the new *service start time* with regards the next *arrival time* (Figure shows the possible cases);

(b) **GO TO** 4;

9. **END.**



(a) Service start time overlap updating



(b) Service start time not overlap updating

Then we can define the *coalition utility function* $V_j(C) = v_j$, $0 \leq v_j \leq 1$ as

$$V_j(C) = \begin{cases} \frac{F(C,T_j)}{T_j^{ser}} & F(C,T_j) \geq T_j^{sat} \\ 0 & F(C,T_j) < T_j^{sat} \end{cases}, \tag{5.2}$$

that is the *service time* ratio with regard to the time task T_j remains in the environment, when coalition C successfully accomplishes task T_j, 0 otherwise. Therefore the aim of the proposed *coalitions satisfaction problem* is the same of Section 2.7, whose purpose is maximizing, over the possible coalitions, the summation of the V_j, the *coalition satisfactions*, i.e.

$$\arg\max_{\mathbf{C}} \sum_{j=1}^{m} V_j(C_j). \tag{5.3}$$

For instance consider again the previous proposed situation, where R_1 and R_3 are in the environment with the only task T_2. According to Table 5.1 the *arrival times* $f_1^2(R_1^{pos}, T_2^{pos})$ and $f_3^2(R_3^{pos}, T_2^{pos})$ are respectively 6.27 and 9.88 (see Figure 5.9). After its arrival robot R_1 serves immediately the task for $R_1^{ser} = 2$ seconds, then the task has to await until time 9.88 with the come of robot R_3. However R_3 can not serve T_2 for $R_3^{ser} = 2$ seconds because of the

deadline service time $T_2^{ser} = 10$, hence it real *service time* is equal to 0.11. The *total amount of service* is therefore $R_{C_2}^{ser} = 2.11$, which, combined with a task satisfaction $T_2^{sat} = 2$, implies a *coalition satisfaction* $V_2 = (C_2) = \frac{2.11}{10} = 0.21$.

Iteration number	0	1
Arrival times	$< f_1^2, f_3^2 >$	$< f_3^2 >$
Service start S^{ser}	6.27	9.88
Service end time E^{ser}	8,27	11.88
$R_{C_2}^{ser}$ surplus	0	1.88
Total amount of service $R_{C_2}^{ser}$	2	2.11

Table 5.2: Example of coalition satisfaction $V_2\,(C_2 = \{R_1, R_3\})$

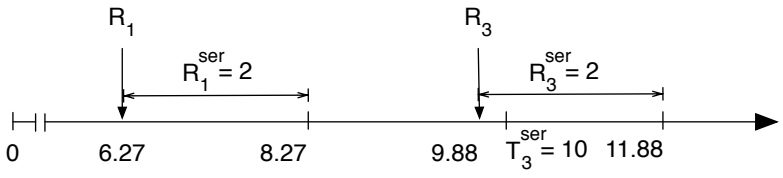


Figure 5.9: Example of total amount of service $R_{\{R_1,R_3\}}^{ser}$

5.3 Factor graph and GDL

The theoretical framework used to solve the proposed *coalition satisfaction problem* is the *factor graph* [15], which is suitable to represent the optimization problem described by Equation (5.3).

Indeed, let $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ be a collection of variables, where each variable x_i takes values in a finite alphabet A_i and let $g\,(x_1, x_2, \ldots, x_n)$ be a R-valued function of these variables, i.e. a function whose domain is

$$S = A_1 \times A_2 \times \cdots \times A_n$$

and codomain R. The domain S is called *configuration space* for this collection of variables, while each element of S is a particular *configuration* of the variables, i.e. an assignment of a value to each variable. The codomain R may generally be any semiring , so that we can also assume R is the set of real numbers.

We recall that a *commutative semiring* is a set K, with two binary operations called " $+$ " and " $\cdot$ ", which satisfy these axioms:

1. The operation "$+$" is associative and commutative and there is an additive identity element "0" such that $k + 0 = k$, $\forall k \in K$;

2. The operation " $\cdot$ " is associative and commutative and there is a multiplicative identity element "1" such that $k \cdot 1 = k$, $\forall k \in K$;

3. For all triples (a, b, c), $a, b, c \in K$ $(a \cdot b) + (a \cdot c) = a \cdot (b + c)$, that is to say the *distributive law* holds.

As stated before, the set of real or complex numbers, with ordinary addition and multiplication, forms a commutative semiring. However there are many other commutative semirings, some of which are summarized in Table 5.3. For example, consider the *Max-sum semiring* (entry 5), where:

1. K is the set of real numbers, plus the symbol ∞;
2. The operation " $+$ " is defined as the operation of *taking the maximum* with $-\infty$ as identity element (i.e. $\max(k, -\infty) = k$, $\forall k \in K$);
3. The operation " $\cdot$ " is defined as the ordinary addition with 0 as identity element and $k + \infty = \infty$, $\forall k \in K$;
4. The *distributive law* $\max(a + b, a + c) = a + \max(b, c)$ is always true.

Number	K	" $(+, 0)$ "	" $(\cdot, 1)$ "	Short name
1	$[0, \infty)$	$(+, 0)$	$(\cdot, 1)$	Sum-product
2	$(0, \infty]$	$(\min, \infty)$	$(\cdot, 1)$	Min-product
3	$[0, \infty)$	$(\max, 0)$	$(\cdot, 1)$	Max-product
4	$(-\infty, \infty]$	$(\min, \infty)$	$(+, 0)$	Min-sum
5	$[-\infty, \infty)$	$(\max, -\infty)$	$(+, 0)$	Max-sum
6	$\{0, 1\}$	$(OR, 0)$	$(AND, 1)$	Boolean

Table 5.3: Some commutative semirings

Moreover, suppose that function $g(\mathbf{x})$ can be decomposed into a summation of different functions, that is

$$g(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{r} F_i(\mathbf{x}_i), \ \mathbf{x}_i \subseteq \mathbf{x} \tag{5.4}$$

a *factor graph* is defined as a *bipartite graph*[1] that shows the structure of this summation. In particular a factor graph $FG = \{\mathbf{F}, \mathbf{x}\}$ is made up of *variable nodes*, one for each x_i, i.e. $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ and *function nodes,* one for each $F_i(\cdot)$, that is $\mathbf{F} = \{F_1, F_2, \ldots, F_n\}$, where a variable node x_i is connected to the function node F_j if and only if that variable is one of the arguments of that function, i.e. $x_i \in \mathbf{x}_j$.

Consider for example the function

$$g(x_1, x_2, x_3) = F_1(\mathbf{x}_1) + F_2(\mathbf{x}_2),$$

[1] In the mathematical field of graph theory, a *bipartite graph* is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V, that is U and V are independent sets.

where $\mathbf{x}_1 = \{x_1, x_2, x_3\}$ and $\mathbf{x}_2 = \{x_1, x_2\}$. This structure is shown by the *graph* of Figure 5.10 with *function nodes* $\mathbf{F} = \{F_1, F_2\}$ and *variable nodes* $\mathbf{x} = \{x_1, x_2, x_3\}$ and edge connections represented by set

$$E = \{(F_1, x_1), (F_1, x_2), (F_1, x_3), (F_2, x_1), (F_2, x_2)\}.$$

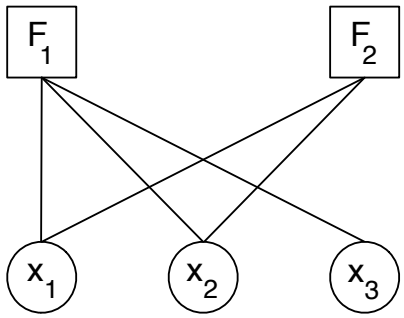


Figure 5.10: Factor graph example

In the proposed approach, given the set of robots $\mathbf{R}_n$, we suppose each robot $R_i \in \mathbf{R}_n$ locally possesses and can control only a function $F_i(\mathbf{x}_i)$ and a variable x_i and has knowledge of, and can directly communicate only with its neighboring[2] robots.

Our supposition is that $\mathbf{x}_i = \{x_1, x_2, \ldots, x_n\}, \forall i$, i.e. the *factor graph* is a *complete bipartite graph*[3], which also means that all robots belong to the same neighboring. For instance in Figure 5.11, there are two robots, R_1 and R_2 which respectively possess their pair function-variable (F_1, x_1) and (F_2, x_2) but are neighbors and can communicate each other, because both $\mathbf{x}_1$ and $\mathbf{x}_2$ equal $\{x_1, x_2\}$.

[2]Two robots are neighbors if there is a relationship connecting variables and functions that robots control.

[3]In the mathematical field of graph theory, a *complete bipartite graph* is a special kind of bipartite graph where every vertex of the first set is connected to every vertex of the second set.

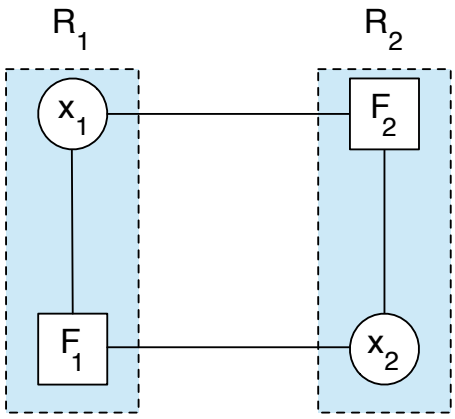


Figure 5.11: Robot-controlled factor graph

5.4 Max-sum algorithm

The *Max-sum algorithm* belongs to the family of *iterative message passing algorithms* called *Generalized Distributive Law* (*GDL*) [16], which can be combined with *factor graphs* to efficiently compute functions with the same structure of Equation (5.4).

Given a set of robots $\mathbf{R}_n$, a set of tasks $\mathbf{T}_m$ and a complete *factor graph* such as the one of Figure 5.11, the *Max-sum algorithm* computes

$$\mathbf{x}^{\star} = \arg\max_{\mathbf{x}} \sum_{i=1}^{n} F_i\left(\mathbf{x}_i\right), \tag{5.5}$$

an optimization similar to the proposed *coalitions satisfaction problem.* Indeed this algorithm can be very adapt when dealing with such *coalition formation* problems even if in completely different environment such as the *RoboCup Rescue* [17] hence presenting a distributive approach to a problem whose solution algorithms are generally centralized. Let each variable x_i represent the possible tasks a robot can satisfy and can take values over a finite domain $d_i \subseteq \mathbf{T}_n$, hence each function $F_i\left(\mathbf{x}_i\right)$ represents the amount of the utility given to the system if robot R_i acts for task x_i, eventually supported by other robots. This is a different point of view of the same optimization problem, where the maximization is shifted from task to robot, and the evaluation of $F_i\left(\mathbf{x}_i\right)$ trades off between these representations. Hence we have to introduce $V_j^i\left(x_i\right)$, an utility function, which connects the *coalition utility function* $V_j\left(C\right)$ to the *real service time* R_i^{rs} given by robot R_i to task T_j,

$$\begin{cases} V_j^i\left(\mathbf{x}_i\right) = \frac{R_i^{rs}}{F(TM_i(\mathbf{x}_i),T_j)} \cdot \frac{1}{T_j^{ser}} & V_j\left(TM_i\left(\mathbf{x}\right)\right) > 0 \\ 0 & otherwise \end{cases},$$

where $TM_i(\mathbf{x})$, $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ represents the set of robots $\{R_j\}$ whose variable x_j has the same argument of x_i. In other words $V_j^i(\mathbf{x}_i)$ is the percentage of service robot R_i offers respect to the *total amount of service* $F(TM_i(\mathbf{x}), T_j)$ with regard to the *deadline time service* T_j^{ser}. Let describe the main steps $F_i(\mathbf{x}_i)$ entries are evaluated:

1. **START**
2. Find the *robot variable* between the set $\mathbf{x}_i$, that is x_i, the variable controlled by robot R_i;
3. For each entry:
   (a) Evaluate $TM_i(\mathbf{x}_i)$, i.e.:
      i. Evaluate the argument of *robot variable*, i.e. T_j;
      ii. Add R_i to set $TM_i(\mathbf{x}_i)$;
      iii. For each variable $x_k \in \mathbf{x}_i$:
      iv. If x_k has T_j as argument, add robot R_k to $TM_i(\mathbf{x}_i)$;
   (b) Evaluate $F_i(\mathbf{x}_i)$ as $V_j^i(\mathbf{x}_i)$.
4. **END**

For instance, consider the environment of Figure 5.7 with the addition of task T_3 which is described by $T_3^{pos} = \left\{5, 1, 0.1, \frac{\pi}{2}\right\}$, $T_3^{ser} = 8$ and $T_3^{sat} = 2$. The situation is different from before and robots have to choose if join forces to accomplish that or the other task or act in a *divide-and-conquer* approach.

First of all let Table 5.4 show the table form of function $F_1(x_1, x_3)$, where the *main variable* is x_1, such that the real F_1 evaluations are:

- $F_1(T_2, T_2) = V_2^1(\{R_1, R_3\})$;
- $F_1(T_2, T_3) = V_2^1(\{R_1\})$;
- $F_1(T_3, T_2) = V_3^1(\{R_1\})$;
- $F_1(T_3, T_3) = V_3^1(\{R_1, R_3\})$.

During the developing of these procedures, we notice that this approach always consider the alphabet d_i of each variable x_i equal to $\mathbf{T}_m$ even if it is not necessary. In other words, supposing the current value of the *main variable* is T_j, a robot can take into account and eventually insert into the coalition, robots which can not reach T_j before its *deadline service time*, that is robots which can not offer any service to the task.

Therefore, in order to get the updated tabular form (see Table 5.5a), we erase from variable x_1 all the tasks unreachable by R_1; in the example shown in Figure 5.12 robot R_1 reaches task T_3 at time 10.30, exactly 2.30 seconds after the task leaves the environment, such that $|d_1|$, the cardinality of x_1 alphabet

is decreased by one unity (the same procedure is applied to $F_3(\mathbf{x}_3)$ obtaining Figure 5.5b).

F_1	x_1	x_3
0.090	T_2	T_2
0.100	T_2	T_3
0	T_3	T_2
0	T_3	T_3

Table 5.4: Evaluation of $F_1(x_1, x_3)$ table form

F_1	x_1	x_3
0.090	T_2	T_2
0.20	T_2	T_3

(a) Reduced evaluation of $F_1(x_1, x_3)$ table form

F_3	x_1	x_3
0.005	T_2	T_2
0.125	T_2	T_3

(b) Reduced evaluation of $F_3(x_1, x_3)$ table form

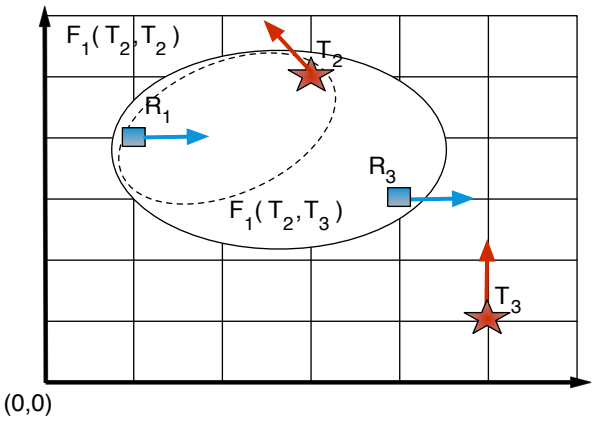


Figure 5.12: Example for $F_1(\mathbf{x}_1)$ table form representation

After this table filling, the optimized $\mathbf{x}^\star$ of Equation (5.5) is achieved by repeatedly passing messages within the *factor graph* (Figure 5.13 shows the messages exchanged between robots R_1 and R_2 in the *factor graph* Figure 5.11):

1. From variable nodes to function nodes, called *q-messages*;

2. From function nodes to variable nodes, called *r-messages*.

Formally, a *r-message* $r_{j\to i}(x_i)$ from function F_j to variable x_i is defined as

$$r_{j\to i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} \left[F_j(\mathbf{x}_j) + \sum_{k \in \mathcal{N}_j \setminus x_i} q_{k\to j}(x_k) \right], \tag{5.6}$$

where $\mathcal{N}_j$ is the set of variable indexes, indicating which variable nodes in the *factor graph* are connected to function node F_j and $\mathbf{x}_j \setminus x_i \equiv \{x_k : k \in \mathcal{N}_j \setminus i\}$. On the other hand a *q-message* $q_{i\to j}(x_i)$ from variable x_i to function F_j is defined as

$$q_{i\to j}(x_i) = \alpha_{ij} + \sum_{k \in \mathcal{M}_i \setminus j} r_{k\to i}(x_i), \tag{5.7}$$

where $\mathcal{M}_i$ is the set of function indexes, indicating which function nodes in the *factor graph* are connected to variable node x_i and α_{ij} is the *message normalization factor* such that $\sum_{x_i} q_{i\to j}(x_i) = 0$[4].

When the *factor graph* is cycle free, the algorithm is guaranteed to converge to the global optimal solution $\mathbf{x}^\star$, thereby optimally solving the proposed optimization problem. Moreover, this convergence can be achieved by first calculating

$$z_i(x_i) = \sum_{j \in \mathcal{M}_i} r_{j\to i}(x_i), \tag{5.8}$$

we have trivially called *z-message*[5], and then computing

$$\arg\max_{x_i} z_i(x_i). \tag{5.9}$$

However it can be show that, despite the lack of convergence guarantees, the *GDL algorithms* generates good approximate solutions when applied to cyclic graphs [18].

In order to better explain this kind of messages exchanging, let we apply the algorithm to the last presented system, i.e. that one with robots set $\mathbf{R}_2 = \{R_1, R_3\}$ and tasks set $\mathbf{T}_2 = \{T_2, T_3\}$. Figure 5.13 shows both q and r messages for a single iteration exchanged over a *factor graph* with the same structure of that in Figure 5.11 and the table form of the functions are those of Table 5.5a and 5.5b.

[4]In cyclic factor graphs such normalization prevents messages to increase endlessly, on condition that there are not negative infinity utilities, which are usually taken into account to represent hard constraints on the solution.

[5]The *z-message* is considered a message because it is made up of the sum of other messages, but at the same time it is judge trivial because it is calculated locally and is not exchanged between robots.

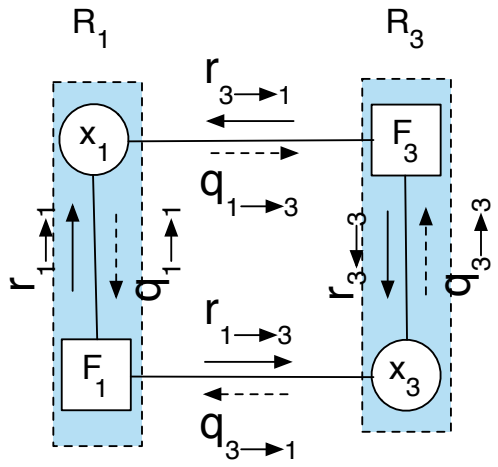


Figure 5.13: Max-sum messages example

Since at the fist iteration, whose messages are indicated with the superscript 0, *q-messages* $q^0_{i\to j}(x_i)$ equal 0, and consequently *r-messages* become $r^0_{j\to i}(x_i) = \max_{\mathbf{x}_j \setminus x_i} F_j(\mathbf{x}_j)$, in our example we get:

- $q^0_{i\to j}(x_j) = 0,\ j \in \{1,3\},\ i \in \{1,3\}$;
- $r^0_{1\to 1}(x_1) = \max_{x_3} F_1(x_1, x_3) = \{\langle T_2, 0.100\rangle\}$;
- $r^0_{1\to 3}(x_3) = \max_{x_1} F_1(x_1, x_3) = \{\langle T_2, 0.090\rangle, \langle T_3, 0.100\rangle\}$;
- $r^0_{3\to 1}(x_1) = \max_{x_3} F_3(x_1, x_3) = \{\langle T_2, 0.125\rangle\}$;
- $r^0_{3\to 3}(x_3) = \max_{x_1} F_3(x_1, x_3) = \{\langle T_2, 0.005\rangle, \langle T_3, 0.125\rangle\}$.

Next we applying Equation (5.8) and (5.9) we obtain:

- $T_2 = \arg\max_{x_1} [\{\langle T_2, 0.100\rangle\} + \{\langle T_2, 0.125\rangle\}]$;
- $T_3 = \arg\max_{x_3} [\{\langle T_2, 0.090\rangle, \langle T_3, 0.100\rangle\} + \{\langle T_2, 0.005\rangle, \langle T_3, 0.125\rangle\}]$.

This means that variables x_1 and x_3 after computing their *z-message*, choose the value that maximize it, that is T_2 and T_3 respectively, with a system utility with this which equals 0.225.

	F_1	F_3
x_1	-0.125	-0.100
x_3	-0.065	-0.0095

Table 5.5: Example of message normalization factors

However the algorithm does not end at the first iteration and immediately computes the second iteration *q-messages*, which are normalized by setting each α_{ij} according to Algorithm 5.2. For instance we get $\alpha_{31} = -0.065$ and consequently $q^1_{3\to1}(x_3)$ following these steps:

1. Component-wise sum all $r_{3\to3}$ vectors (lines 2-5);
2. Sum the resulted components (lines 6-8) $(0.005 + 0.125 = 0.130)$;
3. Change sign to the obtained value (line 9) (-0.130);
4. Divide the previous result by the vector cardinality $(-\frac{0.130}{2} = -0.065)$ (line 10);

Then if we add α_{31} to each $r_{3\to3}$ component we get $\{\langle T_2, -0.06\rangle, \langle T_3, 0.06\rangle\}$.

Therefore if the same procedure (Table 5.5 summarize the computed *message normalization factors*) is applied to other messages we also obtain:

- $q^1_{1\to1}(x_1) = \{\langle T_2, 0.0\rangle\}$;
- $q^1_{1\to3}(x_1) = \{\langle T_2, 0.0\rangle\}$;
- $q^1_{3\to3}(x_3) = \{\langle T_2, 0.005\rangle, \langle T_3, -0.005\rangle\}$.

At this point the algorithm has the required messages to computed the relative *r-messages*, whose evaluation is left to the reader (for instance Figure 5.14 shows how the *r-message* $r^1_{3\to1}(x_1)$ is computed). The most important fact is that the *z-messages* obtained using the $r^1_{j\to i}(x_i)$ messages make the system evaluate $x_1 = T_2$ and $x_3 = T_3$ again, which means the algorithm has converged to a solution, ending its execution.

Algorithm 5.2 Message normalization factor evaluation

```
1  computeAlpha(Rmessages){
2    messageQ messQ;
3    foreach message in Rmessages{
4      messQ .+ message;
5    }
6    foreach value in messQ{
7      alpha += value;
8    }
9    alpha *= -1.0;
10   alpha /= messQ.size();
11   return alpha;}
```

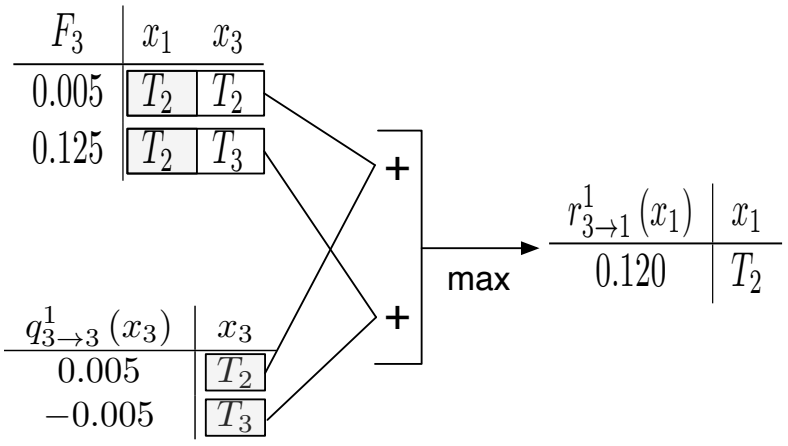


Figure 5.14: Evaluation of $r^1_{3\to1}(x_1)$ message

5.4.1 Bounded Max-sum algorithm

The *Max-sum* algorithm is extremely attractive for the decentralized coordination of computationally and communication constrained devices for the following reasons:

1. The messages are small and scale with the domain of the variables;
2. The number of messages exchanged typically varies linearly with the number of agents within the system;
3. The computational complexity of the algorithm scales exponential with just the number of variables on which each function depends [19].

However, the lack of guaranteed convergence and guaranteed solution quality, limits the use of the standard *Max-sum* algorithm in many application domains, and such exponential computation behavior is undesirable in systems composed of devices with constrained computational resources. The *Bounded Max-sum* [20], a variation of the M*ax-sum algorithm*, whose approach removing cycles from the *factor graph*, by ignoring some of the dependencies between functions and variables, ensures the convergence of the algorithm to a bounded approximate solution.

Since our proposed approach works with *complete factor graphs*, which means each function depends upon each variable, and the cardinality of the task to be accomplished $\mathbf{T}_m$ is in general of the order of the number of robots $|\mathbf{R}_n|$ the exponential behavior is here stressed. Nevertheless, due to the characteristics of the proposed environment and optimization problem, we choose to adopt and implement a system which use the *simple Max-sum* algorithm (see Section 7.3 for more exhaustive considerations and Section 7.1.1 for some experiments).

Chapter 6

Software framework

6.1 ROS

ROS[1] (*Robot Operating System*) is a *robot software system,* whose development was started in the 2007 with the name of *switchyard* by Morgan Quigley and his team at the *Stanford Artificial Intelligence Laboratory*, but in 2008 was taken on by the *Willow Garage*[2] group and the current release is called *Electric Emys* (see Figure 6.1). ROS is very adapt to the development of robotics software, indeed it provides interesting services, such as hardware abstraction, devices control, *message passing* between processes, source codes and dependencies management, but relies on the hosting operating system for low level services, e.g. processes scheduling, memory management and network communications.

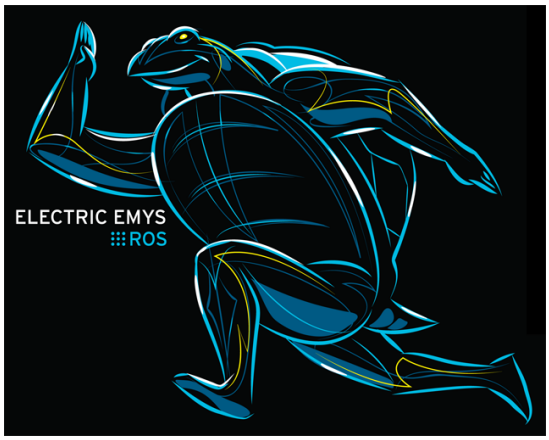


Figure 6.1: ROS release *Electric Emys*

[1] www.ros.org
[2] *www.willowgarage.com*

ROS architecture is component-based, whose components are organized in packages, stacks and repositories (see Figure 6.2), where:

1. *A Package* is at the lowest level of ROS software organization and contains libraries, executable and configuration files;
2. A *Stack* is a packages collection providing aggregated high level functions;
3. A *repository* is where a set of *stacks* is collected to allow the software distribution.

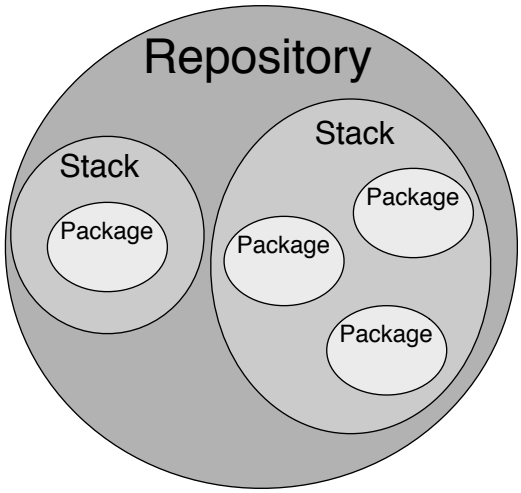


Figure 6.2: ROS File System

In detail ROS architecture is based on the *ROS Computational Graph*, a *peer to peer* network of *loosely coupled*[3] processes, whose components are:

1. The *master* node;
2. The *nodes*;
3. The *parameter server;*
4. The *bags.*

As the main process of ROS execution, the *Master* handles the *parameter server,* the network resources and *service naming,* allowing all the other processes to communicate each other. Indeed *processes* in ROS are *nodes* which after registering at the *master* node, can communicate each other using *topic, service*

[3]Loose coupling is an approach to interconnecting the components in a system or network so that those components depend on each other to the least extent practicable. that is coupling refers to the degree of direct knowledge that one element has of another.

and the *parameter server*. Handled by the *master* node,The *parameter server* is a shared dictionary that the nodes access to save or recover runtime system parameters. Finally through the *bags* ROS records in *log* files all the communications occurred in the system.

In ROS the communication between two nodes do not imply their mutual knowledge, but only the *name* on the network of the *service* offered or the *topic* used. Indeed ROS uses three mechanisms to make nodes communicate and interact each other:

1. *Service;*
2. *Topic*;
3. The shared memory of the *parameter server.*

A *service* represents the RPC procedure on ROS, where a *node* in the computation graph can be a *provider* or *client:* the former provides at least one *service*, which have to be identify by a unique *name*, in a hierarchic namespace, while the latter sends requests to *providers* specifying the *name* of the required service, then waiting the response.

A *topic* provides a unidirectional and asynchronous streaming communication, letting *nodes* exchange messages assuming the role of *publisher* or *subscriber*: a node which generate datas can be a *publisher* with regards to a specific *topic, s*pecifying the *name* and kind of message it publishes, while a node which desires receiving communications have to subscribe a topic, specifying its *name,* the kind of message and the function which handles the received messages.

Because of ROS architecture and the choice of using a *belief propagation algorithm*[4], such as *Max-sum*, we choose to implement the framework to solve the proposed *coalitions satisfaction problem* over the ROS system which highly favors a distributed arrangement. However the focus is here on how the structure of the Max-sum algorithm has been implemented rather than centering the attention on how messages are really exchanged.

6.2 Factor graph

The *factor graph* is the support where the *Max sum* algorithm is applied, hence the *FactorGraph* object was the first to be implemented, together with its main components, i.e. *NodeFunction* and *NodeVariable* objects. According to our definition of *factor graph* (see Subsection 5.3), given a *FactorGraph*, a robot is able to gain access of its local data but possesses a reference for every neighboring *variable* and *function* node. Consider Figure 5.11, which represents a *complete factor graph*, where R_1 and R_2 respectively own *function nodes* $F_1(\mathbf{x}_1)$ and $F_2(\mathbf{x}_2)$ and *variable nodes* x_1 and x_2, but can fully communicate with the other

[4]Generally a belief propagation is a message passing algorithm for performing inference on graphical models.

robot, then Figure 6.3 gives the idea of how this example of mathematical *factor graph* has been implemented over ROS. Indeed each robot here possesses a reference, represented by the $\star$ symbol, to neighboring *NodeVariable* and *NodeFunction* objects.

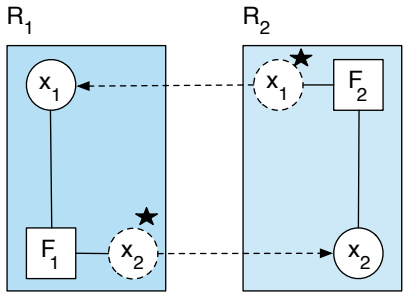


Figure 6.3: Factor graph implemented structure

After the implementation of *NodeVariable* and *NodeFunction*, two other objects was considered: the *NodeArgument* and the *TabularFunction*. A *NodeVariable* x_i contains a *NodeArgument* object A_i^j for each element of its alphabet, i.e. $A_i = \left\{A_i^1, A_i^2, \ldots, A_i^m\right\}$, $m = |x_i|$, while a *TabularFunction* represents the table form of a *function node*.

Figure 6.4 shows how both these objects are connected within the *FactorGraph* nodes: each robot can access both its local and neighboring *NodeFunction* and *NodeVariables* objects and can evaluate its *TabularFunction* with regards to the *NodeArguments* taken by the *NodeVariable*s. For instance if we consider Table 5.5a relative to the proposed example with robots R_1 and R_3, the complete *FactorGraph* becomes that of Figure 6.5.

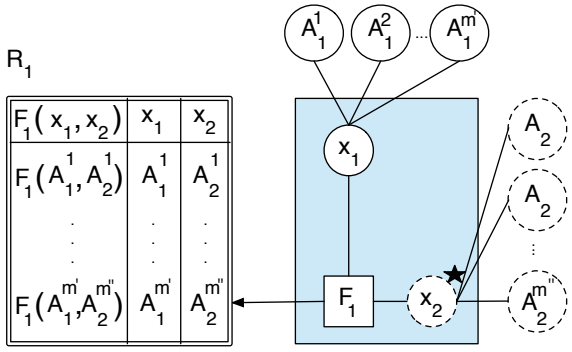


Figure 6.4: TabularFunction and NodeArguments

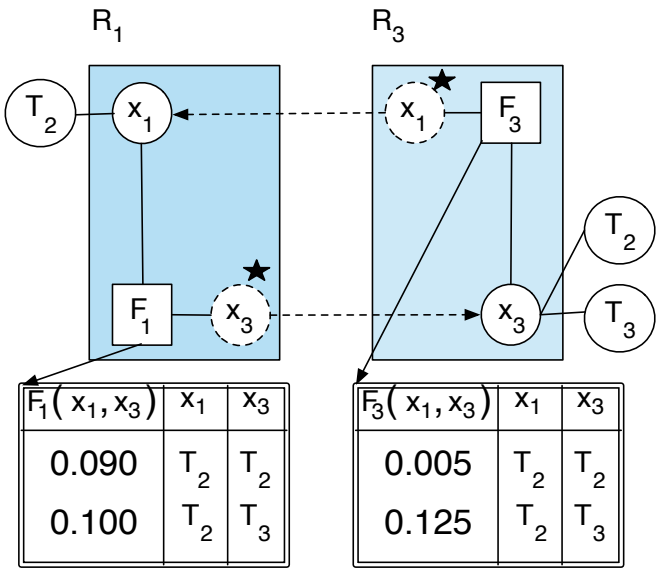


Figure 6.5: Complete FactorGraph

6.3 Max-sum algorithm

In Section 7.1 we describes the procedures robots exploit along with the *Max-sum* algorithm to solve the proposed *coalitions satisfaction problem* but, before that, it is important to point out the main steps which compose this algorithm. The Max-sum algorithm, as an *iterative message passing* algorithm, relies on the messages exchange between *factor graph* nodes, hence we first developed *a distributed mail manager*, represented by a *DMailMan* object. This message manager, combined with ROS architecture and its communication mechanisms, allowed the development of transparent procedures which make robots communicate and collaborate each other.

Indeed, according to this software transparency this section do not place emphasis on how the *mail manger* has been realized or how the message are really exchanged, but supposes the *factor graph* nodes can interact in message exchanging. Hence the *Max-sum* procedure is here presented as composed of four main block steps (see Figure 6.6), where only the last one properly executes the *Max-sum* algorithm:

1. The *startConnection*;
2. The *initialization;*
3. The *makeTabularFunction*
4. The *computeItaration;*

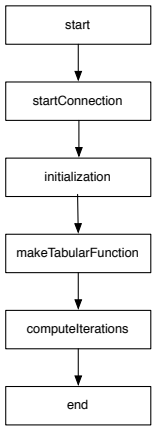


Figure 6.6: Max-sum procedure

In the *startConnection* steps robots synchronize each other first by *registering* to the ROS *master node*, then by querying this node about the *position* on the network of other robots. Then in the *initialization* step each robot first creates its local *factor graph*, which consists of a function and a variable node, and then share it to other robots shaping the needed *complete factor graph*. During this steps robots also create the zero *q-messages* $q^0_{i\to j}(x_i)$ as required by the first iteration of the *Max-sum* algorithm.

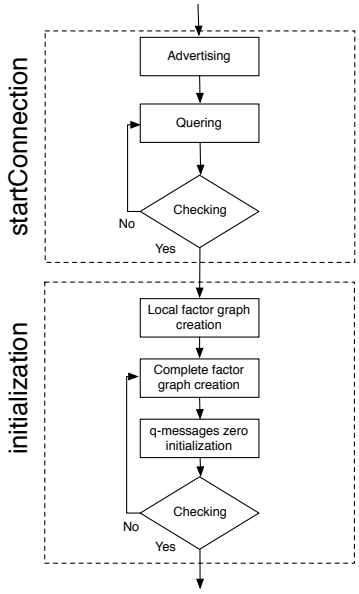


Figure 6.7: startConnection and Initialization steps

After both steps terminate without errors in the *makeTabularFunction* block robots firsts creates the *TabularFunction* for their local functions, then considering the alphabets of the *NodeVariables* they gain from other robots and those they possess, they evaluate each table entries. Next the procedure enters the *computeIteration* block where it loops until the loop checking *conditions* are satisfied (see Figure 6.8).

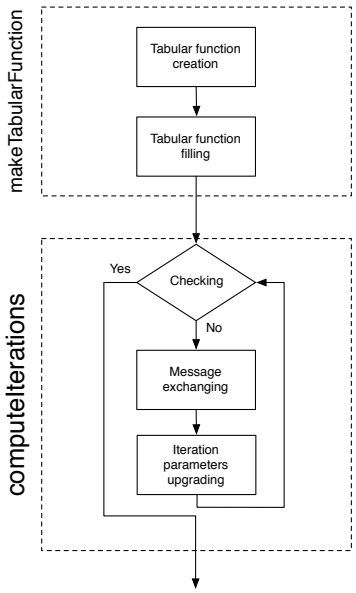


Figure 6.8: makeTabularFunction and computeIteration steps

The *computeIterations* block is the core of the *Max-sum* procedure, hence let we describe more in detail how it works, for example starting from the conditions which make the procedure stops (the flow-chart is shown in Figure 6.12). First define the summation $U_i = \sum_{i=1}^{n} F_i(\mathbf{x}_i)$ of Equation (5.5) as the utility the system can gain if it stops at the ith iteration. Except for the first one, suppose the system previously computed the ith iteration, then before continuing with the (i+1)th iteration, it enters the *checking* sub-step of Figure 6.8, that is shown in detail in Figure 6.9. The procedure controls if:

1. A fixed point is reached, that is to say the system utility is the same of the previous iteration, i.e. $U_{(i-1)} = U_i$;

2. The maximum number of iterations scheduled has been reached.

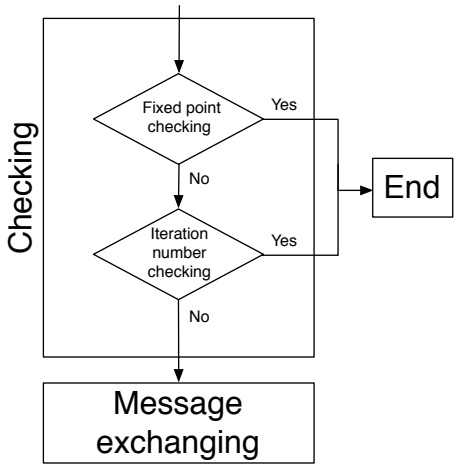


Figure 6.9: Checking sub-step

This last point is due to the fact that robots executes on-line this algorithm and sometimes a sub-optimal but fast-computed solution is preferred to the optimal solution which can spend too much time and resources. Then If one of these conditions are satisfied the *Max-sum* procedure ends, otherwise each robot enters first the *message exchanging*, then the *iteration parameters upgrading* sub-steps (see Figure 6.10 and Figure 6.11, respectively). Respect to the ith iteration previously computed each robot:

1. Sends and receive the *q-messages* obtained by the previous iteration, i.e. $q^{i}_{i\to j}(x_i)$;

2. Builds the *r-messages*, i.e. $r_{j\to i}(x_i)$;

3. Sends and receives the *r-messages.*

Next it upgrades all the parameters needed for the (i+1)th iteration by executing the following steps:

1. Builds the *q-messages* $q^{i+1}_{i\to j}(x_i)$;

2. Builds the *z-messages* $z_i(x_I)$;

3. Upgrades the algorithm iteration number from i to $t+1$;

4. Updates the *system utility* according to the maximization get by Equation (5.9).

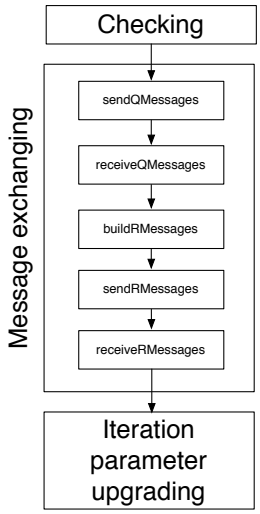


Figure 6.10: Message exchanging sub-step

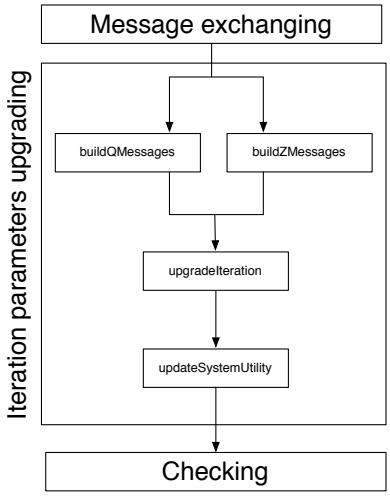


Figure 6.11: Iteration parameters upgrading sub-step

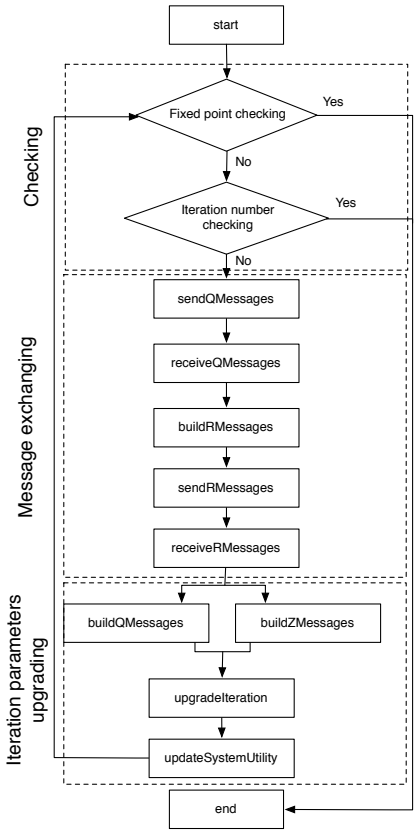


Figure 6.12: ComputeIteration block

Chapter 7

Experiments

7.1 Problem solution

Since we introduced the utility functions (see Section 5.2), the theoretical framework (see Section 5.3) algorithm (Section 5.4) chosen to solve the proposed problem and we presented the main concepts of their software implementation (see Chapter 6), we focus our attention on how robots use them to deal with the *coalition satisfaction problem*.

The solution of such problem instances can be approached with a PDCA method[1], that is by iteratively executing the following steps (see Figure 7.1):

1. Establish the objectives and processes necessary to deliver results in accordance with the expected output ($PLAN$);

2. Implement the plan, execute the process (DO);

3. Study the actual results (measured in DO above) and compare against the expected results (targets or goals from the $PLAN$) to ascertain any differences ($CHECK$);

4. Request corrective actions on significant differences between actual and planned results (ACT);

[1]PDCA is an iterative four-step management method used in business for the control and continuous improvement of processes and products.

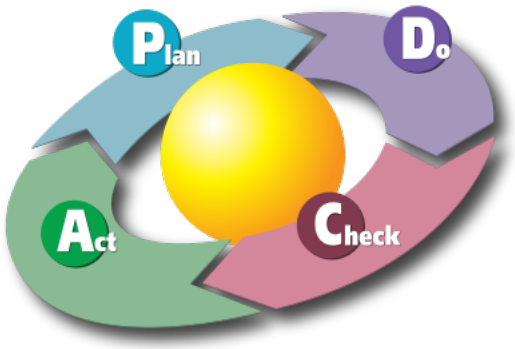


Figure 7.1: PDCA

In more concrete terms, let the system start with a robots set $\mathbf{R}_n$ and the initial tasks set $\mathbf{T}_m^0$, it is supposed robots completely know each task T_j, that is:

1. Its position T_j^{pos};
2. Its *deadline service time* T_j^{ser};
3. Its *satisfaction service* T_j^{sat}.

Then robots, after building their local *variable and function nodes* x_i and $F_i(\mathbf{x_i})$, have to communicate each other in order to exchange their position and create the *complete factor graph* described in Section 6.2 (*PLAN*).

Afterwards each robot R_i fills its x_i with the proper alphabet and the *function node* with the relative $F_i(\mathbf{x_i})$ table form by evaluating with the *A-function* and getting the subset of tasks, i.e. $\mathbf{T}_{i,m}^0 \subseteq \mathrm{T}_m^0$, it can reach before they leaves the environment. According to Section 6.3 applying the *Max-sum* algorithm over the built *factor graph* (*DO*) robots:

1. Find a task allocation within a before-know maximum number of iterations;
2. Form coalitions which move towards and reach the tasks;
3. Serve the task.

Within the *CHECK* step, robot R_i after updating its reachable tasks set $T_{i,m}^1 \subseteq T_{i,m}^0$ in accordance with the elapsed time:

1. If $T_{i,m}^1 = \emptyset$, it stops and attends the arrival of other reachable tasks;
2. Otherwise, it enters the *ACT* step.

In particular each robot which enters into this step clears:

1. The alphabet A_i of its local *variable node* x_i;

2. The table form entries of its local *function node* $F_i(\mathbf{x}_i)$.

Then all the remaining robots start a *PLAN* step communicating each other to update their positions and *factor graph* neighbors.

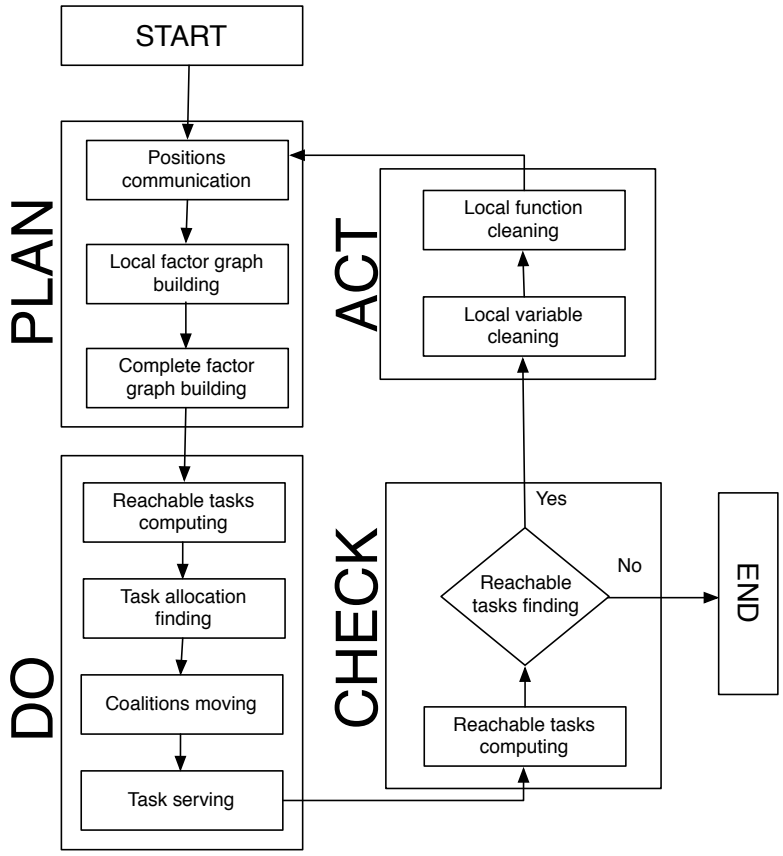


Figure 7.2: System execution flow-chart

Such system has been tested using *Adept*[2] Pioneer 3-AT robots (see Figure 7.3), small four-wheel, four-motor skid-steer robots ideal for all-terrain operation or laboratory experimentation. A Pioneer 3-AT comes complete with one battery, emergency stop switch, wheel encoders and a micro-controller with ARCOS firmware.

[2]http://www.mobilerobots.com

Figure 7.3: Pioneer 3-AT

However this robots is simulated under ROS using its specifications e.g. its dimension shown in Figure 7.4 or its forward/backward speed ($v^{scalar} = 0.7\,^{\mathrm{m}}/_{\mathrm{s}}$) and its rotation speed ($2.09\,^{\mathrm{rad}}/_{\mathrm{s}}$) to create an URDF[3] whose graphical representation is shown in Figure 7.5.

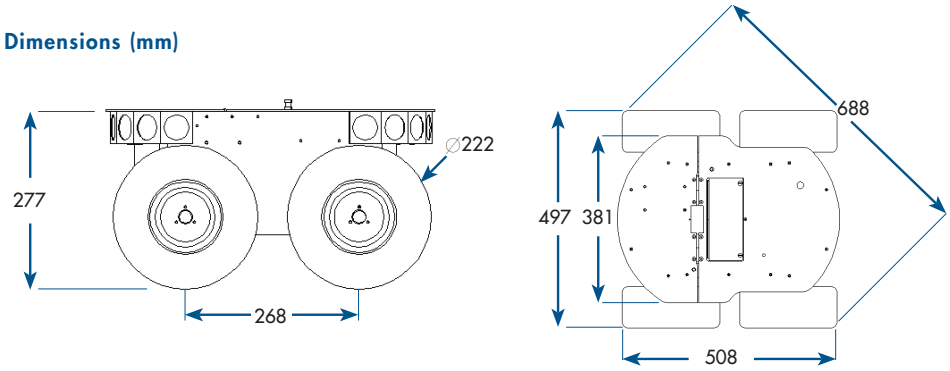


Figure 7.4: Pioneer 3-AT dimensions

Figure 7.5: Simulated pioneer 3-AT

Consequently the environment itself is simulated within ROS middleware under *Gazebo*[4], a 3D multi-robot simulator with dynamics, capable of simulating

[3]The Unified Robot Description Format (URDF) is an XML format for representing a robot model.

[4]http://gazebosim.org/

articulated robot in complex and realistic environments, combined with another 3D visualization environment for robots, called *rviz* (respectively see Figure 7.6 and Figure 7.7).

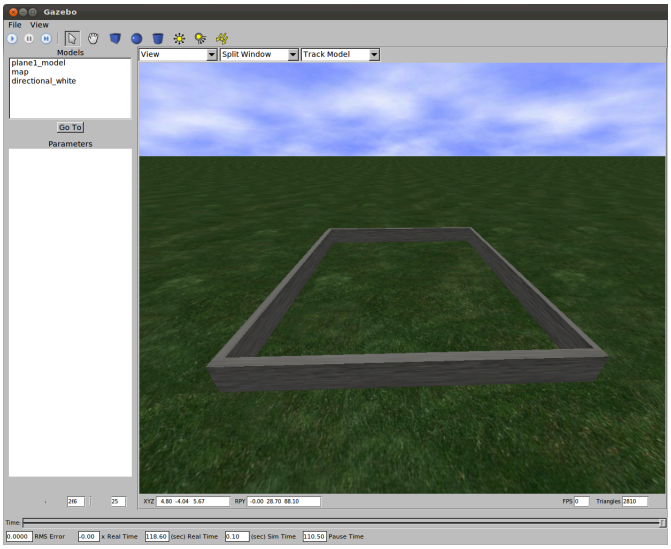


Figure 7.6: *Gazebo environment*

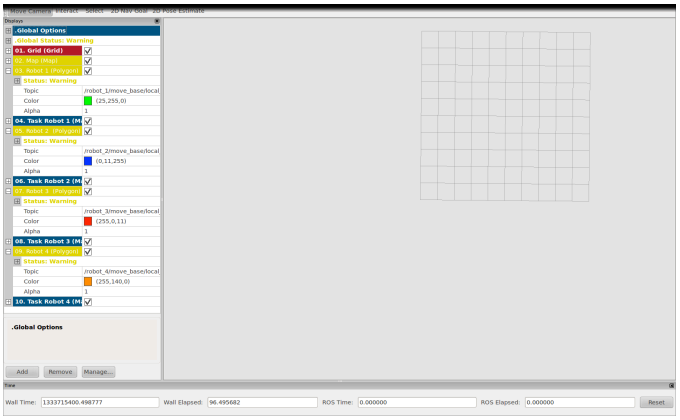


Figure 7.7: Rviz environment

We represent here the instance of Figure 5.1, whose robot parameters are as follows:

1. $R_1^{pos} = (1, 4, 0)$, $R_1^{ser} = 1$;

2. $R_2^{pos} = (3, 4, \pi)$, $R_2^{ser} = 1$;

3. $R_3^{pos} = \left(4, 3, \frac{\pi}{2}\right)$, $R_3^{ser} = 1$;

4. $R_4^{pos} = \left(5, 3, \frac{3\pi}{2}\right)$, $R_4^{ser} = 1$.

The system in Figure 7.8 shows three tasks, i.e. T_1, T_2 and T_3 with parameters:

1. $T_1^{pos} = \left(1, 1, \frac{\pi}{4}\right)$, $T_1^{ser} = 7$ and $T_1^{sat} = 1$;

2. $T_2^{pos} = \left(3, 5, \frac{3\pi}{1}\right)$, $T_2^{ser} = 9.5$ and $T_1^{sat} = 1.5$;

3. $T_3^{pos} = \left(5, 1, \frac{\pi}{2}\right)$, $T_3^{ser} = 5.5$ and $T_1^{sat} = 1$.

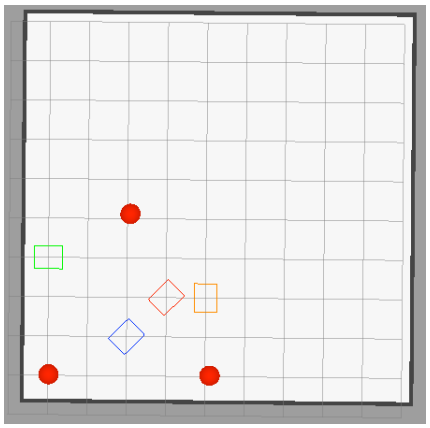


Figure 7.8: Simulated coordination instance

Then after applying the PDCA cycle, all tasks colors change from red to yellow, in other words they are all chosen by a robot and wait for service (see Figure 7.9): both R_1 and R_3, the green and red robots, choose to serve and positively accomplish task T_2, while the blue robot R_2 and the the orange robot R_4 respectively choose task T_1 and T_3 (see Figure 7.10).

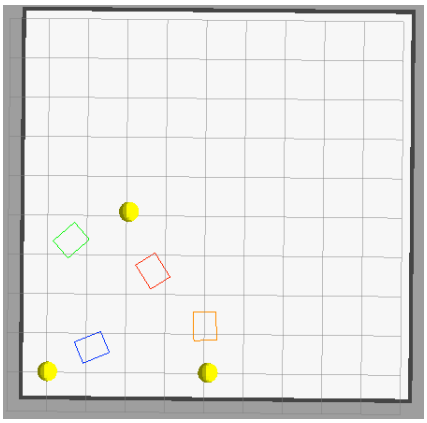


Figure 7.9: Robot's chosen tasks

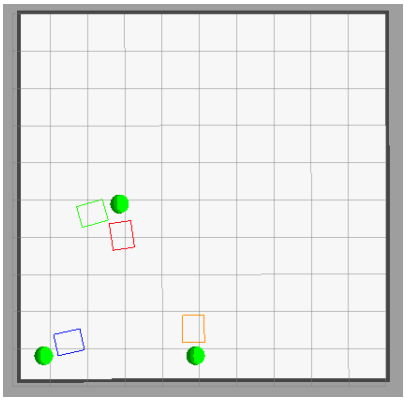


Figure 7.10: Coalition task serving

Another simulation example is given by that of FIgure 7.11 where there are four robots but only two tasks which however require the intervention of the entire system (Table 7.1a and Table 7.1b summarize the whole system parameters).

R_i	R_i^{pos}	R_i^{ser}
1	$(1, 6, 0.2, 0)$	1
2	$\left(4, 3, 0.2, \frac{\pi}{2}\right)$	1
3	$\left(6, 4, 0.2, \frac{3\pi}{4}\right)$	1
4	$(8, 5, 0.2, \pi)$	1

(a) Robot parameters

T_i	T_i^{pos}	T_i^{ser}	T_i^{sat}
1	$(1, 6, 0.2, 0)$	14	2
2	$\left(4, 3, 0.2, \frac{\pi}{2}\right)$	12.5	1.5

(b) Task parameters

Indeed the task satisfaction levels are such that one robot cannot achieve the goal: task T_2 with its *deadline service time* equal to 12.15 and 1.5 *satisfaction service* can be accomplished only by R_4 and R_3, the first two fastest robots. In order to gain an higher utility the system can try to accomplish also task T_1, but suppose robots R_4 and R_3 are unavailable, the only possibility is to assign it to robots R_1 and R_2. The algorithm makes this choice to optimize the total utility whose result is shown in Figure 7.12.

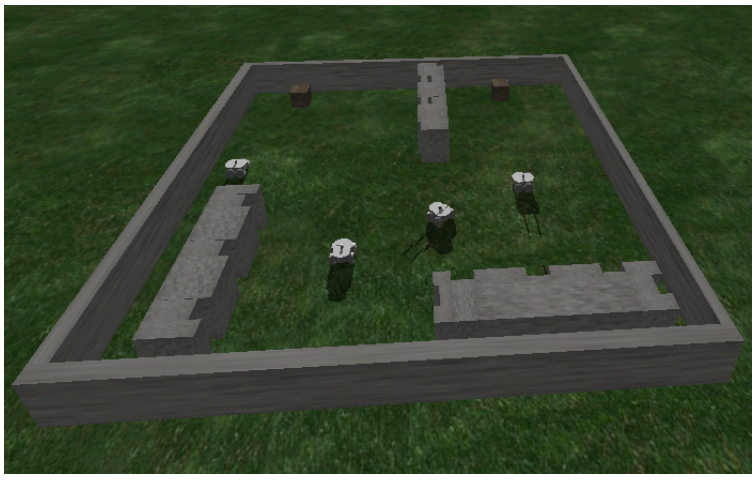


Figure 7.11: Task allocation instance under Gazebo

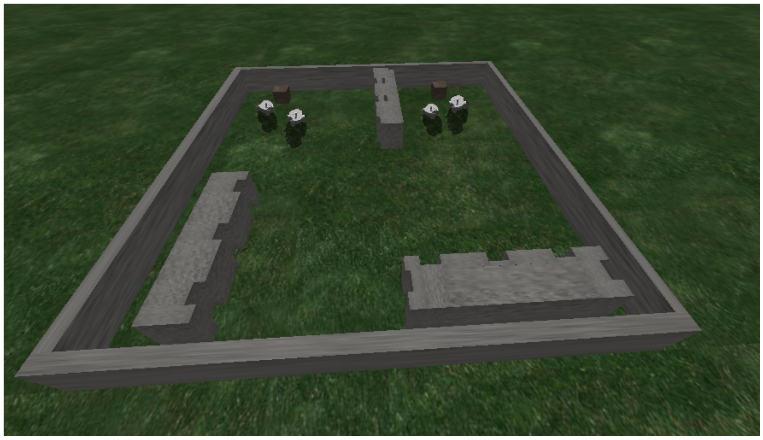


Figure 7.12: Task allocation solution under Gazebo

7.1.1 Agent Satisfaction

According to Section 2.7 we implemented a version of Algorithm 2.10: while the A-function is kept as in our proposed solution, the *coalition utility function* becomes, i.e. *aggregate function*

$$V_j(C) = F(C, T_j),$$

that we recall represents R_C^{ser}, the *total amount of service* coalition C can offer to task T_j. With these modifications the algorithm undergoes some changes: the *system objective* relative to task T_j is only represented by the *satisfaction service* T_j^{ser}, while the *self evaluations* of R_i respect to T_j are computed according to Equation (5.1), which evaluates the arrival time to that task. Even if partially modified the pseudocode shown in Algorithm 7.1 still represents a greedy approach, which as such can leads to sub-optimal solutions, while the Max-sum algorithm guarantee optimality.

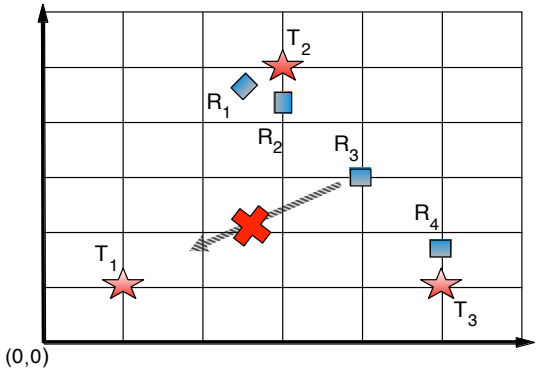


Figure 7.13: Agent satisfaction algorithm failure

For instance in the previous scenario of Figure 7.8 the resulted assignment is different from the one given by the *Max-sum* algorithm. In detail T_3 with a $\frac{T_3^{sat}}{T_3^{ser}}$ ratio equal to 0.18 is the task with the highest priority and is served and satisfied by robot R_4 (here this is the same of our approach). Then the task with higher priority is T_2, where the faster robots which can satisfy the task are in order: R_1, R_2 and finally R_3, which arrive at times 4.84, 7.52 and 8.38. In this case, because of the *satisfaction service* equal to 1.5, two robots are needed, so R_1 and R_2 are subsequently assigned to that task. As Figure 7.13 the last task is T_1, but the only available robot is R_3 which can reach the task only at time 6.67, after its deadline time service, making this task be unserved. Indeed differently from the Max-sum algorithm, this procedure has assigned robots to tasks with greedy choices without considering robots as a whole system, so that in some way the algorithm puts robots' satisfactions before system's performance.

Algorithm 7.1 Modified Agent satisfaction

1. Input: Tasks, Agents, SystemObjectives, SelfEvaluations
2. **Output**: TaskToExecute
3. SortedTasks ← Sort Tasks according to $\frac{T_i^{sat}}{T_i^{ser}}$, where higher values correspond to higher priorities
4. **while** SortedTasks $\neq \emptyset$ **do**
5. Task ← **Pop**(SortedTasks)
6. SortedAgents ← Sort Agents given Self Evaluation for Task, where faster arrivals correspond to higher priorities
7. AgentSatisfaction ← 0
8. AssignedAgents(Task) ← $\emptyset$
9. **while** AgentSatisfaction < SystemObjectives(Task) ∧ SortedAgents $\neq \emptyset$ **do**
10. AssignedAgents(Task) ← AssignedAgents(Task) ∪ **Pop**(SortedAgents)
11. AgentSatisfaction ← **Aggregate(**AssignedAgents(Task)**)**
12. **end while**
13. **if** AgentSatisfaction < SystemObjectives(Task) then
14. AssignedAgents(Task) ← $\emptyset$
15. **else**
16. Agents ← Agents \ AssignedAgents(Task)
17. **end if**
18. **if** mySelf ∈ AssignedAgents(Task) then
19. TaskToExecute ← Task
20. **return** TaskToExecute
21. **end if**
22. **end while**
23. **return** NoTask

The same result is gotten if this algorithm is applied to the problem instance of Figure 7.11. Indeed in this case T_1 is the task with the highest priority, hence Algorithm 7.1 first assign R_1 and R_4 to accomplish that task, then as Figure 7.14 shows, R_3 and R_1 are the only available robots, which cannot reach T_2 in time.

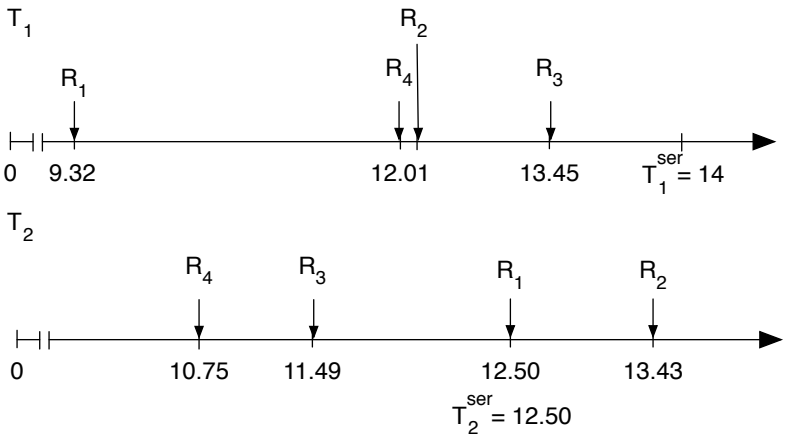


Figure 7.14: Arrival times to T_1 and T_2

7.2 A lower level of coordination

The main assumption made so far is that robots safety reach their chosen tasks, i.e. there are no collisions and the time evaluated by the *A-function* is a good estimation of the time needed by robots to arrive at their chosen task position. However, considering that such assumption is not borne out of the facts and as it is stated in the state of the art, that coordination in multi-robot systems is a such complex and challenging task that composite architectures are generally needed (see Chapter 4), a lower level of coordination, i.e a *collisions avoidance* is here introduced [3].

Before discussing how the *collisions* are avoided in the proposed system, let briefly introduce how in literature the problem is handled. The *collisions avoidance problem* in a static and multi-robot known environment can be dealt with a *reactive* or a *predictive* approach: the former is a class of methods that permits robots to avoid collisions on a dynamic environment without explicit communication, such as the *Dynamic Window Approach* [21], while the latter has his most recent extension on the *Optimal Reciprocal Collision Avoidance* (ORCA) [22], which can be used to simulate thousands of moving agents without collisions and achieve this objective without communication. In turn the *predictive* approaches can be addressed either with *coupled* or *decoupled* approaches: the former guarantee completeness but generate an exponential dependence on the number of robots and use a centralized computation [23], the latter allow robots to compute their own paths and then resolve conflicts, so that feasible solutions are usually incomplete, but computed in a decentralized and faster way.

Hence consider $\mathbf{R_m}$, i.e. a set of robots with a second order dynamics rule by the time constraint

$$\dot{x_i}(t) = f(x_i(t), u_i),$$

where $g\left(x_i\left(t\right), \dot{x}_i\left(t\right)\right) \leq 0$, $\forall t \in \mathbb{R}$, $x_i(t)$ and u_i respectively represent a system state and a robot control and functions f and g are both smooth. Let $E \subseteq \mathbb{R}^2$ be the *environment* where the robot operates and $FE \subseteq E$ the *free environment*, i.e. the free space of that *environment*, then given a point $p \in \mathbb{R}^2$, for each robot R_i:

1. $f_P\left(R_i, p\right)$ is called the *footprint* on the point p, i.e. the subset of FE occupied by the robot;

2. $c\left(R_i\right) \subseteq \mathbb{R}^2$ is the center of that *footprint;*

Therefore we call *the safe environment,* which is represented with SE_i, the 2D local subspace of FE where robot R_i can perceive and move, or more formally every $c\left(R_i\right)$ such that $f_P\left(R_i, c\left(R_i\right)\right) \subseteq FE$. For instance in Figure 7.15...

Now suppose that each robot R_i starts possessing a *local goal* list LG_i filled up with 2D space points $p \in SE_i$ and that R_i has to reach a *global goal* G_i by passing through a sequence of local goals. Thus after robot R_i has reached a chosen *local goal* by covering a linear trajectory at $v_i \in V_i$ speed, first it has to compute a new list LG_i, next it has to choose a new *local goal* $lg_i \in LG_i$ and a velocity $v_i \in V_i$ such that, until it does not reach it, $\forall t$ and $\forall i \neq j$

$$\begin{cases} f_P\left(R_i, v_i \cdot t\right) \in SE_i \\ f_P\left(R_i, v_i \cdot t\right) \cap f_P\left(R_i, v_j \cdot t\right) \neq \emptyset \end{cases}.$$

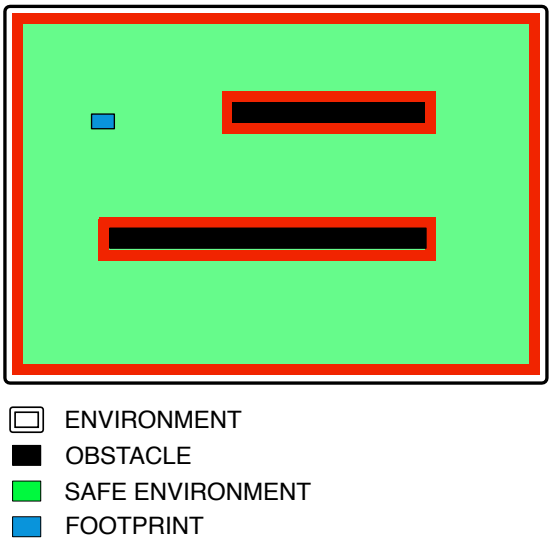


Figure 7.15: An example of *E, FE, SE_i*

The presented problem is therefore solved by means of the *collisions avoiding* system shown in Figure 7.16 which execute a planning cycle composed of the following modules:

1. The *Environment Model Builder*;
2. The *Local Goals Generator*;
3. The *Communication framework*;
4. The *Motion Planner;*
5. The *Controller.*

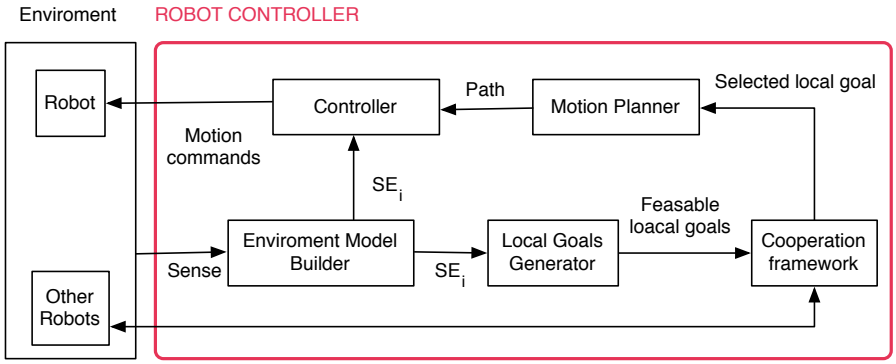


Figure 7.16: Collisions avoidance system

In detail the *Environment Model Builder* retrieves sensor and odometry environment data, which uses to compute a *costmap*[5] and sends the evaluated *self environment* to the *Local Goals Generator* module, which produces as output a set of *feasible*[6] *local goals* around the robot position after taking into account the robot *global goal.* Then in the *Cooperation framework* each robot first broadcast its position, then compares it with the ones received by other teammates and according to these comparisons it can choose to:

1. Reach its *local goal* because it is not near to other robots;
2. Cooperate with its closer teammates which are within a *cooperationDistance radius,* e.g. 2 meters, because some collisions can arise.

The former case is the most interesting one because we choose again to rely on the *Max-sum* algorithm over the *graph* mathematical framework to assure a *cooperative collisions avoidance* approach. Even for this problem solution,

[5] A *costmap* is a discrete grid inflated with costs obtained from environment data.
[6] With *feasible* we means that for sure there is a path between the robot position and the local goal.

each robot R_i possess a local variable node x_i and a local function node $F_i(\mathbf{x}_i)$, where:

1. The variable x_i represents the paths towards the candidate *local goal*s;
2. The function $F_i(\mathbf{x}_i)$:
   (a) computes the minimum distance between all possible *local goals*, logarithmically weighted the Euclidean distance to the *global goal,* if there are no collisions;
   (b) is set to a small positive ϵ value, otherwise.

However in this case the *factor graph* is not necessary a *complete factor graph* as in Section 6.2 because the concept of neighboring between nodes is here related to the distance between robots. In other words, consider the situation presented in Figure 7.17 where robots R_1, R_2 R_3 and R_4 have to reach *global goal* G_1, G_2, G_3 and G_4, respectively. In particular robot R_1 is not near enough to other robots, hence do not create any *factor graph* and consequently do not cooperate within the system, while both R_2 and R_4 are within the cooperation area of R_3 they have to interact with.

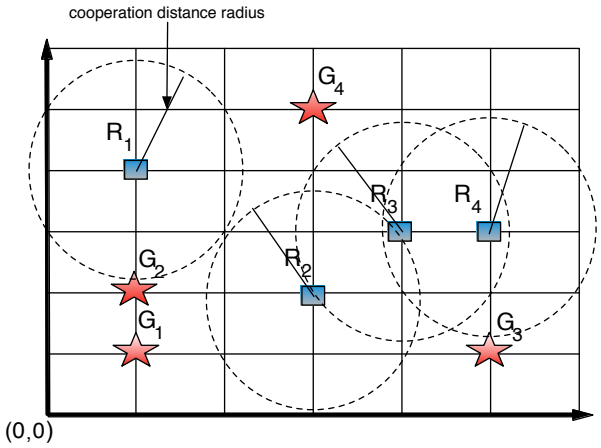


Figure 7.17: Example of cooperative collisions avoidance

Therefore the *Max-sum* executes the messages exchanging over the *factor graph* of Figure 7.18 with the result of assigning such *local goals* to R_2, R_3 and R_4 such that the *system utility* $U = F_2(x_2, x_3) + F_3(x_2, x_3, x_4) + F_4(x_3, x_4)$ is maximized. Respect to other approaches the *Max-sum* algorithm plays a leading role on exploiting as best as possible to the trade off between avoiding collisions and getting closer to the *global goal* G_i represented by function $F_i(\mathbf{x}_i)$. Indeed in the example of Figure 7.19 we apply:

1. A greedy algorithm based on the *global goal* distance;

2. A general collisions avoiding algorithm;
3. The *Max-sum* algorithm.

As result the greedy algorithm fails and leads robots to a collision, the *collisions avoiding* algorithm positive handles the collision, but makes robots moving away from their *global goals,* while the *Max-sum* algorithm chooses the path which mediate the other algorithms' goals.

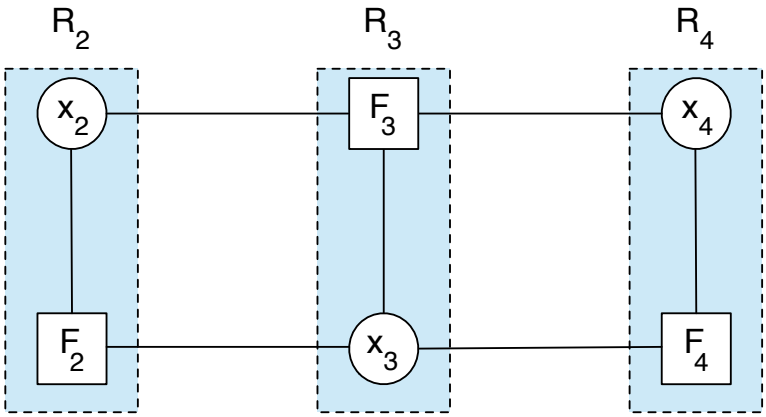


Figure 7.18: Example of a collisions avoidance factor graph

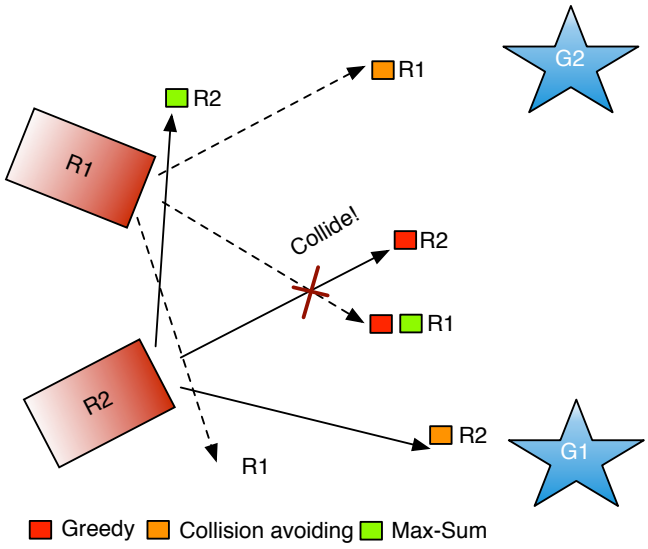


Figure 7.19: Collisions avoidance algorithms comparisons

Before proceeding with the description of left system modules, i.e. the *Motion planner* and *Controller*, let first define *SD* as the *safe distance* (e.g. 0.5 meters) the space needed by the robot to safely carry out on of its ICS[7] escape maneuverers, i.e. if a robots has to cover a distance of length $L \geq SD$ it will move for that length minus the *safe distance.* Nevertheless if recovery actions are needed, the robot first try to slowly rotate to escape obstacles, eventually looking for other path. Then the *Motion Planner* first computes a path towards the selected local goal using the $A^\star$ algorithm [25] next it cover that distance with the Dynamic Window approach shown in Algorithm 7.2, which fundamentally:

1. Computes the goals directly reachable from the current robot position;
   (a) Selects the *safe*[8] goals around robot position not near to *useless* goals;
2. If at least a goal is found, it chooses that one which minimizes the distance to the *global goal*, otherwise it takes recovery actions.

Algorithm 7.2 High level navigation procedure

```
1  geostructure gs;
2  position globalGoal;
3  while(globalGoal is not reached){
4    currPos = getCurrentPosition();
5    gs.add(currPos);
6    gs.find(currPos).type = GOOD;
7    localGoals = computeGoalsFrom(currPos);
8    if(localGoals.size() > 0){
9      newLocalGoal = selectBest(localGoals);
10     gs.add(newLocalGoal);
11     moveTo(newLocalGoal);
12   }else{
13     gs.find(currPos).type = USELESS;
14     recoveryGoal = gs.findGoodNeighbour(currPos);
15     if(not set recoveryGoal)
16       contingencyPlan();
17     else
18       moveTo(recoveryGoal);
19   }
20 }
```

7.2.1 Coordinated collisions avoidance

Since both *collisions avoidance* problem and *coalitions satisfaction* problems rely on the *Max-sum* algorithm executed over the same framework, i.e. a *factor graph*, a system, where both the presented problems solutions coexist, is here described introducing a different logical point of view. Indeed a typical solution,

[7] A state is an *Inevitable Collision State* (ICS) [24] if every next state involves a collision.
[8] A goal is defined as safe when the trajectory towards it does not make the robots collide.

which involves the integration of these systems can be that of the *coalitions satisfaction* problem (see Section 7.1), with the difference that in the *DO* step robots move towards and serve tasks with the so called *cooperative* collisions *avoidance* algorithm. However, since time is a key feature in our task allocation solution, such approach arose some problems during experiments because even if robots chose the optimal task allocation and avoided collisions at the same time, the time spent on *coordinating* and *cooperating* thorough the Max-sum sometimes was not suitable for real-time systems and they arrived late on the tasks positions.

This is the main reason that led the development of an hybrid framework, we called *coordinated collisions avoidance,* where greedy and sub-optimal approaches are combined together, so that the execution scheme shown in Figure 7.2 becomes the PDCA of FIgure 7.20. Given the assumption the system starts with a set $\mathbf{R}_n$ of robots and an initial tasks set $\mathbf{T}_m^0$, robots do not spend the same time in the *PLAN-DO* steps because, considering their own capacities on satisfying tasks, they adopt a simpler and faster greedy choice. In fact in the *PLAN* step for each task T_j and independently from other teammates, each robot R_i:

1. Evaluates the *A-function* f_i^j;

2. Determines the *coalition utility function* $V_j\left(\{R_i\}\right)$.

Next in the *DO* step each robot first applies the greedy choice on the computed values, i.e. it sorts them with an increasing order and chooses the highest one, then it move towards the relative chosen task. In this way each robot satisfy itself, but that does not mean the system as a whole is satisfied and some tasks cannot be considered at all.

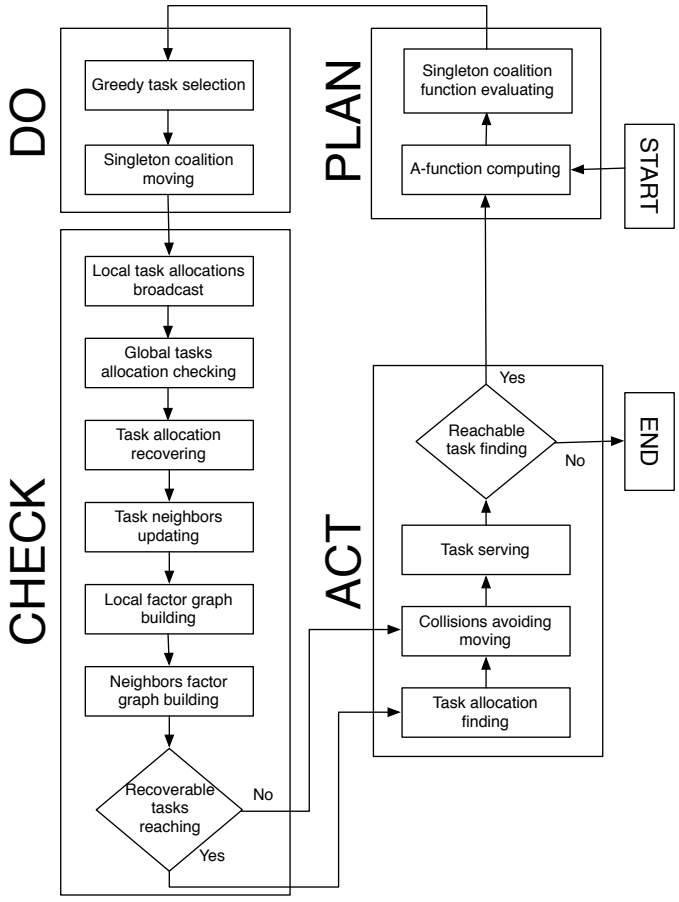


Figure 7.20: Coordinated collision avoidance

Indeed, in order to adjust such undesired situation, the *CHECK* step is redesigned in such a way some robots decide to *coordinate* and *cooperate*, while others continue standalone: such decision is made by each robot after the broadcasting of their chosen task, so that the whole system knows where each robot has decided to move. First each robot R_i considers as neighbor each robot within the *cooperation distance radius* creating the relative *factor* graph, moreover if a robot is also within the *emergency distance radius* $d_{i,j}^{emer}$ of an unchosen task T_j in the system, where

$$\left(\left(T_j^{ser} - R_i^{ser}\right) - \frac{\pi}{v_i^{angular}} - \frac{maxBlockNum}{v_i^{scalar}} \right) \cdot v_i^{scalar},$$

i.e. the maximum distance which permits a robot to reach and serve the task in time, it updates its *factor graph* communicating with the other robots within this task area.

After that in the ACT step, if necessary, robots carry out a task allocation step and eventually change their initial chosen task in favor of a task previously unconsidered, and avoid collisions at the same time. Then if there are other tasks in the environment to be satisfied a robot enters a new $PLAN$ step, otherwise it ends its system execution.

For instance let we take in consideration the situation described in Figure 7.17, where goals G_i are now tasks T_i to be accomplished. Without entering in details, also suppose robots greedily choose task as follows: robots R_1 and R_2 choose task T_2 while task T_3 is chosen by robots R_3 and R_4.

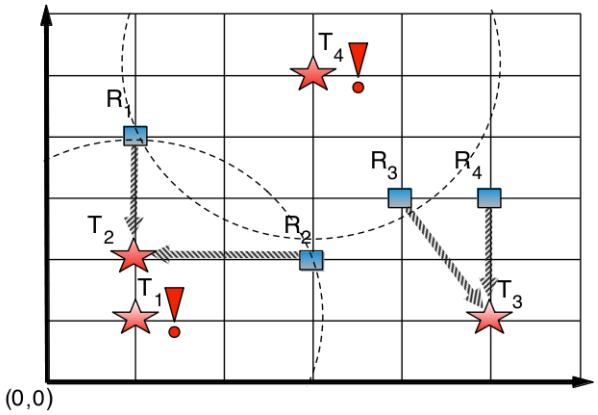


Figure 7.21: Example of coordinated collision avoidance

As stated before some tasks are not chosen at all, i.e. T_1 and T_4, so robots have to update the system factor graph connections in order to handle all task: robots R_3 and R_4 are linked, because they both desire to accomplish task T_3 and for the same reason robots R_1 and R_2 are connected because of T_2. However both robot R_1 and R_3 are within the *emergency area* of T_4 and the same fact occurs with regard to robots R_1 and R_2 and task T_1, hence getting the neighborhood of Figure 7.22.

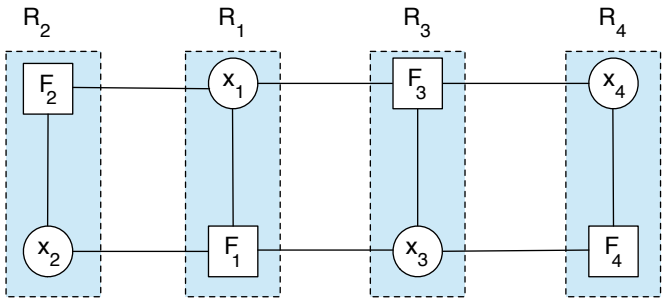


Figure 7.22: Example of a coordinated collision avoidance factor graph

7.3 Conclusions

As experiments have shown the *Max-sum algorithm* applied over the proposed *factor graph* framework, where each robot has its own function and variable nodes, is very attractive for the *coalition formation problem*. Such procedure in a distributive manner but with a few messages exchange guarantees optimal solutions and by the periodic updating of the system neighborhood also the *fault tolerance*.

However this system, which relies on time-based *utility functions,* needed to be completed by a lower level of coordination, i.e. the robotics collision avoidance, which could make robots reach their chosen task without collisions and navigation faults. This is the reason we integrate our high level coordination with a kinodynamic but distributed collision avoidance system [3], we called *cooperative collision avoidance system*.

These systems are merged together thank to their common framework, the *factor graph*, and the distributed procedure, the *Max-sum algorithm,* however it was interesting to introduce and develop a theoretical architecture completely different from those studied in the state of the art, we called *coordinated collision avoidance*.

Such hybrid structure has permitted to trade off between greedy approaches and optimal solutions algorithms, making robots able to avoid collisions, rapidly choose tasks and optimizing those choices at the same time. However all carried out experiments are simulated within the Gazebo simulator over ROS middleware, hence future works could concern the tests and analysis of the proposed system on real Pioneer 3AT robots and real-life environments.

Acknowledgements

The author wishes to thank Michele Roncalli for giving some important features in order to develop the Max-sum algorithm and Nicolò Boscolo for helping the implementation of such distributing algorithm and giving a kinodynamic collision avoidance system. Moreover the author also wishes to thank Matteo Munaro and Stefano Micheletto for giving precious advise about ROS middleware.

Bibliography

[1] O. Shehory and S. Kraus. Methods for task allocation via agent coalition formation, Artificial Intelligence, Volume 101 (1–2), 1998, 165–200.

[2] T. Rahwan, S. D. Ramchurn, A. Giovannucci, V. D. Dang and N. R. Jennings. Anytime optimal coalition structure generation, in Proceeding of the 22nd conference on artificial intelligence, AAAI-07, Vancouver, Canada, 2007, 1184–1190.

[3] N. Boscolo, R. De Battisti, M. Munaro, A. Farinelli and E. Pagello. A distributed kinodynamic collision avoidance system under ROS, in Proceedings of 12th Int. Conference on Intelligent Autonomous Systems (IAS-12), JeJu Island, Korea, June 26-29, 2012.

[4] C. Candea, L. Iocchi, H. Hu, L. Iocchi, D. Nardi and M. Piaggio. Coordination in Multi-Agent RoboCup Teams, Robotics and Autonomous Systems, 2001, Volume 36(2-3), 67-86.

[5] L. Iocchi, D. Nardi, M. Piaggio and A. Sgorbissa. Distributed Coordination in Heterogeneous Multi-Robot Systems, Autonomous Robots, 2003, Volume 15(2), 155-168.

[6] E. Pagello, A. D'Angelo, C. Ferrari, R. Polesel, R. Rosati and A. Speranzon. Emergent Behaviors of a Robot Team Performing Cooperative Tasks, Advanced Robotics, 2003, Volume 17(1), 3-19.

[7] E. Pagello, A. D'Angelo and E. Menegatti. Cooperation Issues and Distributed Sensing for Multi-Robot Systems, Proceedings of the IEEE, 2006, Volume 94(7), 1370-1383.

[8] L. Chaimowicz, R. Kumar and M. Campos. A Mechanism for Dynamic Coordination of Multiple Robots, Autonomous Robots, 2004, Volume 17(1), 7-21.

[9] A. Farinelli, G. Grisetti, L. Iocchi and D.Nardi. Coordination in dynamic environments with constraints on resources, IROSWK02, Dept. of Informatics and Systems, University "La Sapienza", 2002.

[10] O. Zweigle, R. Lafrenz, T. Buchheim, H. Rajaie, F. Schreiber and P. Levi. Cooperative Agent Behaviour Based on Special Interaction Nets, Intelligent Autonomous Systems 9, 2006, Volume 0, 651-659.

[11] A. Farinelli, H. Fujii, N. Tomoyasu, M. Takahashi, A. D'Angelo and E. Pagello. Cooperative control through objective achievement, Robotics and Autonomous Systems, 2010, Volume 58(7), 910-920.

[12] B. Gerkey and M. Matarić. On Role Allocation in RoboCup, Computer Science, 2004, Volume 3020/2004, 43-53.

[13] B. Gerkey and M. Matarić. Are (explicit) multi-robot coordination and multi-agent coordination really so different?, Proceedings of the AAAI Spring Symposium on Bridging the Multi-Agent and Multi-Robotic Research Gap, 2004, 1-3.

[14] B. Gerkey. On multi-robot task allocation, Technical Report CRES-03-012, University of Southern California, 2003.

[15] M. Matarić, G. Sukhatme, E. Ostergaard. Multi-Robot Task Allocation in Uncertain Environments Autonomous Robots, 2003, Volume 14(2-3), 255-263.

[13] M. Isik, F. Stulp, G. Mayer and H. Utz. Coordination without Negotiation in Teams of Heterogeneous Robots, Computer Science, 2007, Volume 4434/2007, 355-362.

[14] N. Lau, L. Lopes, G. Corrente, N. Filipe and R. Sequeira. Robot team coordination using dynamic role and positioning assignment and role based setplays, Mechatronics, 2010.

[15] F. R. Kschischang, B. J. Frey and H. Loeliger. Factor Graphs and the Sum-Product Algorithm, IEEE TRANSACTIONS ON INFORMATION THEORY, 1998, Volume 47, 498-519.

[16] S. M. Aji and R. J. McEliece. The generalized distributive law, IEEE Transactions on Information Theory, 2000, Volume 46(2), 325-343.

[17] S. D. Ramchurn, A. Farinelli, K. S. Macarthur and N. R. Jennings. Decentralized Coordination in RoboCup Rescue, The Computer Journal, 2010, Volume 53(9), 1447-1461.

[18] B. J. Frey and D. Dueck. Clustering by passing messages between data points, Science 315(5814), 2007, 972–976.

[19] A. Farinelli, A. Rogers, A. Petcu and N.R. Jennings. Decentralized coordination of low-power embedded devices using the max-sum algorithm, Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems, 2008, 639–646.

[20] A. Farinelli, A. Rogers and N. Jennings. Bounded Approximate Decentralized Coordination using the Max-Sum Algorithm, IJCAI-09 Workshop on Distributed Constraint Reasoning (DCR), Pasadena, California, USA, 46-59.

[21] O. Brock and O. Khatib. High-speed navigation using the global dynamic window approach, Robotics and Automation, 1999, Proceedings 1999 IEEE International Conference, Volume 1, 1999, 341–346.

[22] J. Van Den Berg, S. Guy, M. Lin and D. Manocha. Reciprocal n-body collision avoidance, Robotics Research, 2011, 3–19.

[23] C. Clark, S. M. Rock and J. C. Latombe. Motion planning for multiple mobile robot systems using dynamic networks, IEEE Int. Conference on Robotics and Automation, 2003, 4222–4227.

[24] T. Fraichard and H. Asama. Inevitable collision states. A step towards safer robots?, Intelligent Robots and Systems, 2003, (IROS 2003), Proceedings 2003 IEEE/RSJ International Conference on, Volume 1(1), 2003, 388–393.

[25] P. Hart, N. Nilsson and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems Science and Cybernetics, Volume 4(2), 1968, 100–107.

[26] J. Bruno, E. G. Coffman and R. Sethi. Scheduling independent tasks to reduce mean finishing time, Commun. ACM 17(7), 1974, 382-387.