Aalto University

LULEÅ UNIVERSITY OF TECHNOLOGY

David Fernàndez López

Towards an autonomous rescue protocol for the Marsu fleet.
Design and analysis of an autonomous rescuer selection method.

School of Electrical Engineering
Department of Electrical Engineering and Automation

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Technology

Espoo, 21.8.2015

Instructor: David Leal Martinez
Aalto University
School of Electrical Engineering

Supervisors: Professor Arto Visala
Aalto University
School of Electrical Engineering

Professor Thomas Gustafsson
Luleå University of Technology

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: David Fernàndez López

Title: Towards an autonomous rescue protocol for the Marsu fleet.

Date: 21.8.2015 Language: English Number of pages: 6+84

Department of Electrical Engineering and Automation

Professorship: Automation Technology (AS-84)

Supervisors: Prof. Arto Visala, Prof. Thomas Gustafsson

Advisor: M.Sc. David Leal Martínez

Energy autonomy is one of the main challenges in robot exploration. The Marsu fleet, a marsupial robot society, is composed of two kinds of robots, the Marsu-bots (explorers) and the Mother-bot (a mobile charging station). The Marsu-bots are able to connect to each other and recharge batteries if necessary. A protocol will be implemented on the Marsu fleet so when a Marsu-bot runs out of battery another Marsu-bot in the fleet will move to its location and recharge its battery. Such a protocol has two main tasks: first, determine how good a robot may be at solving this task, and second, create a distributed algorithm that allows the Marsu-bots to autonomously select the best one. In this paper it will be shown that leader election algorithms can be used to implement the negotiation. It will also be shown that, while it is impossible to accurately predict the performance of a rescue operation based on distance and battery alone, it is possible to find a cost function that allows the selection of a robot able to perform the rescue operation without further assistance. Rules for defining the specific function to be used according to the scenario in which the robots are present will also be given.

Keywords: Swarm Robotics, Cooperating Robots, Autonomous Robotics, Energy autonomy, Search and Rescue

Preface

To send humans back to the moon would not be advancing. [..] But we should return to the moon without astronauts and build, with robots, an international lunar base, so that we know how to build a base on Mars robotically.

Buzz Aldrin

Otaniemi, 16.1.2015

Contents

CHAPTER 1

Introduction

Using robots to explore remote or dangerous environments its not a novel idea. Robots, opposed to humans, are expendable and highly resilient, which makes them a natural choice for such endeavours. Unfortunately, robotic exploration presents a myriad challenges, from physical design and reliability to autonomous decision-making.

One of the major challenges on robotic exploration of remote or dangerous environments is the energy constraint, that is, the limitation of its ability to perform the required tasks due to the limited amount of energy available to it. If a robot is exploring a low-resource, scarcely-populated (or even non-populated) area, such as another planet or a disaster area, it may become difficult to recharge its batteries, leading to the unfortunate event of a robot running out of power and preventing the completion of the assigned task.

Using a swarm of robots able to recharge each other's batteries in case of need is a very promising solution (such as the MARSU fleet presented in section 1.1.1). Using such a swarm if a robot runs out of power another robot in the swarm may recharge it, preventing the loss of the robot. The use of a robotic swarm adds several additional benefits: more robots means higher area coverage, which leads to less time required to explore an area; it also means that the loss of a single robot does not prevent the completion of the task; and much more.

Even then, a situation may arise in which, unexpectedly, one of the robots finds itself in need of a battery recharge and unable to do so by himself (from now on called **victim**). In that case, the swarm should become aware of this situation and select one of its members to save the victim (from now on called **rescuer**).

This thesis aims to design a protocol to autonomously select the best robot to become the rescuer and perform the rescue operation.

1.1 State of the Art

Several techniques to provide energy autonomy to single robots, that is, the capability of a robot to collect and manage energy without the intervention of a human operator, have been studied. Systems such as solar panels [1] or Radioisotope Thermal Generators (RTGs) [2] provide solutions to extend the lifespan of a robot by including on-board energy generators. Other interesting approaches include solutions like the one presented by the EcoBot-III robot [3] a robot with an artificial digestion system that allows it to recharge its batteries by consuming flies.

Those approaches have, however, a common drawback. The addition of on-board generators to a robot create additional challenges, such as the mechanical integration of the generator. In addition an on-board generator adds weight to the robot, which in turn leads to increased power requirements for movement, which lead to higher energy consumption. A simple option to avoid this issue is to separate the generator from the robot, and have the robot move to it when energy acquisition is required.

When using the charging station approach it is fairly simple to add additional robots to the system, all of them sharing the same generator system. This approach adds flexibility and reliability to the system, by adding replication (multiple robots).

Robotic swarms and societies are a novel and promising field in robotics. Though many different studies have been done in robotic swarm performance for a wide range of different tasks, energy autonomy is one of the least studied fields inside robotic swarms.

In general, most current robot swarms have no energy autonomy, and they require to be charged manually by a human operator. Some studies do have a certain degree of energy autonomy, by implementing a protocol in which when a robot has low battery it will move towards a charging station and recharge its batteries there. However, if a robot is, for any reason, unable to reach the charging station before running out of battery there is no protocol in place to autonomously rescue it, that task must be done by a human operator.

Swarms like the Kilobot [4], an experiment on self-organizing big swarms which uses 1024 simple robots that move on a surface to attain a predefined formation, the ChIRP [5], a design for low-cost robotic swarms, or the Roombots [6], a self-aggregating robotic swarm designed to create "intelligent furniture" (cube-like robots that attach to each other to create chairs and tables), assume full battery at the beginning of their experiments. If a robot loses power during the experiment, the event is ignored and the experiment is restarted after manually charging it.

The CISSBot [7] robotic swarm, in contrast, achieves energy autonomy by exchanging batteries. This battery exchange is based on a random epidemic model, in which there is a certain possibility that batteries are exchanged when two CISSBots are close.

Grady *et al.* present a behaviour for the swarm-bot robotic swarm in which a robot is able to rescue another broken robot that belongs to the swarm [8]. A single robot is made to roam around the area trying to find broken robots and, when one is found, it is dragged to a predefined location for recovery. The main difference with the work proposed in this paper is the fact that a robot is preselected to perform this task, instead of selecting a specific robot on demand.

In both studies, the swarm is dense enough to ensure that any member of the swarm will come in close proximity to another one quite frequently, which removes the need to select a specific robot to carry out the rescuer task, as the first robot to come close to the victim will become the rescuer. Due to the high density of robots in the area this close proximity event is bound to happen eventually.

The MARSU fleet, in contrast, forms a sparse swarm, as the robots try to maximize area coverage to reduce the exploration time of an area. Due to this sparseness the same methods are not applicable, as two Marsu robots will rarely come into close proximity during their normal operation, and it is in fact quite likely that a robot that has lost most of its battery is far away from any other robot.

Traub *et al.* presented an algorithm for selecting the best suited robot to move to a goal location and carry a task in that location [9]. The study, however, focuses on the “inherent uncertainty in path traversal times”, and assumes a centralized model (an external monitor has full awareness of the scenario and robots costs, and is responsible for performing the selection algorithm).

1.1.1 The MARSU fleet

The MARSU fleet [10] is a marsupial robot society [11], composed of a number of small robots, called Marsu-bots, and a single tank-like robot able to carry and recharge the Marsus’ batteries called the Mother-bot.

The Marsu-bots

Forming the bulk of the swarm, the Marsu-bots are the main exploratory robots. Small and agile, the Marsu-bots are designed to move in the environment exploring it and recording its features to build a map afterwards. The swarm uses several Marsu-bots.

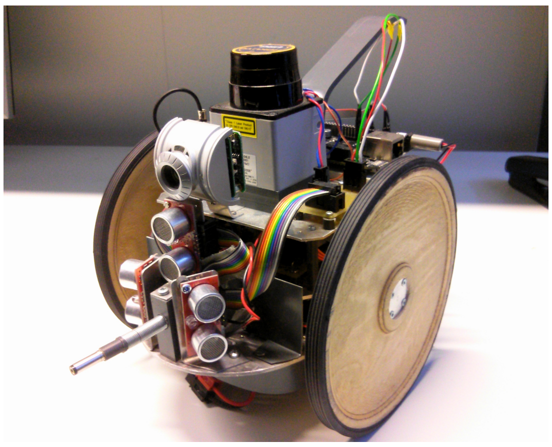


Figure 1.1: A Marsu-bot.

The Marsu-bots are small, differential drive robots equipped with wheel encoders, a small laser range-finder, a set of sonar range-finders (three of them) and a small camera. Powered with a LIFEPO4 battery, the Marsu-bots have a connection port that allows two of them to connect and transfer power.

The Marsu-bots have a Beaglebone black as their processor, running ROS on an ARM Ubuntu. The Marsu-bots have also a USB WiFi dongle connected to provide communication in the swarm.

The Mother-bot

The Mother-bot is a tank-like robot that can carry up to three Marsu-bots inside it and charge its batteries. It serves as a mobile charging station for the Marsu-bots and a transport to overcome obstacles that are too big for the Marsu-bots. In addition the Mother-bot serves as a coordinator for the Marsu-bots, maintaining a global map, merging the partial maps discovered by the Marsu-bots and distributing them to maximize area coverage.

The Mother-bot has much more powerful sensors than its smaller counterparts. It also carries a WiFi antenna.

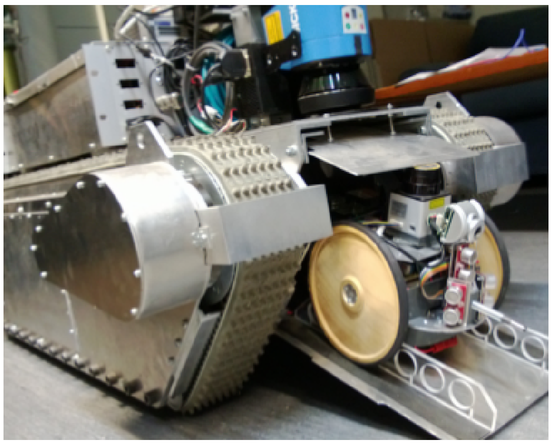


Figure 1.2: The Mother-bot.

The charging capabilities of the Mother-bot are completely decoupled from its processor, as is the WiFi antenna. This way, if any issues appear on the processor of the Mother-bot, even if it can't be moved, it can still be used as a recharge station by the Marsu-bots.

MarSim

MarSim is a NetLogo-based simulation, built by David Leal Martínez [12]. It is designed to simulate the behaviour of the MARSU fleet in different environments and present a graphical simulation. The MarSim simulation allows the modification of several parameters, such as number of robots in the swarm or the battery threshold for recharge and rescue operations.

NetLogo is a "multi-agent programmable modelling environment" [13]. Designed to provide a simple framework to build multi-agent simulations and provide a graphical representation of its execution. NetLogo was built by Uri Wilensky and is available free of charge.

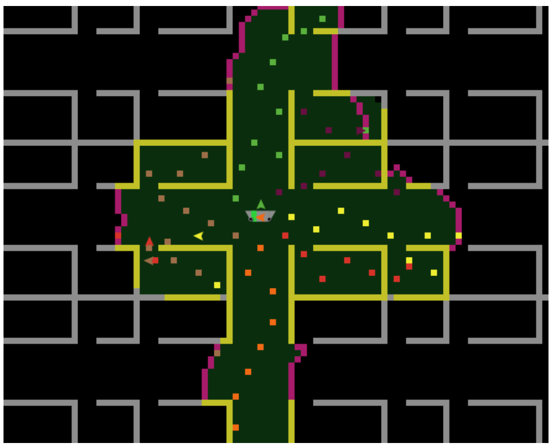


Figure 1.3: A MarSim simulation in an office environment, with 6 Marsu-bots.

1.1.2 MARSU fleet operation

Figure 1.4 shows the flowchart for the Marsu-bot operation. Two main behaviours exist: exploration and recharge. Most of the time the Marsu-bots will follow the exploration behaviour, but, when a Marsu-bot finds itself in a low battery state it will engage the recharge behaviour.

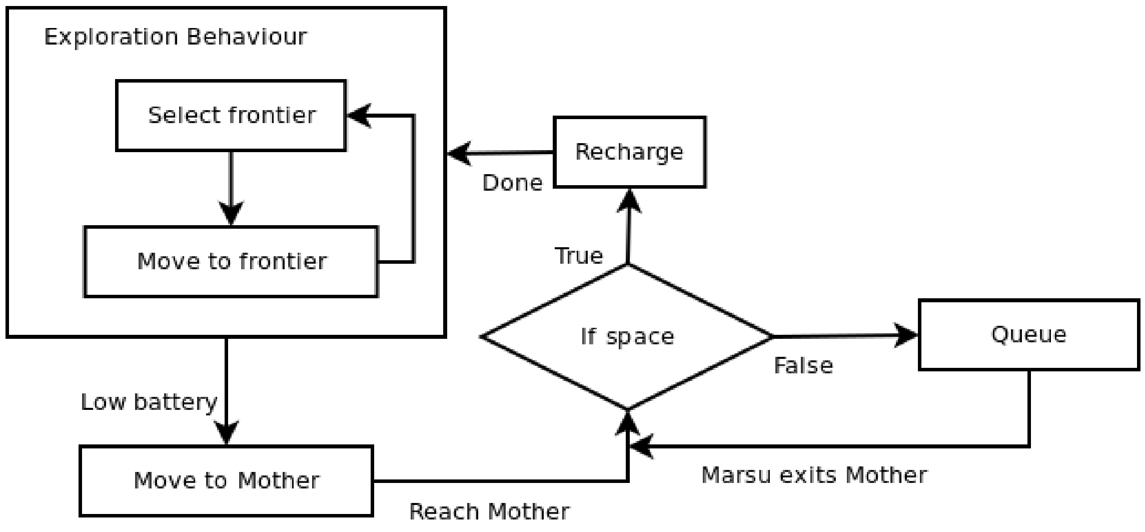


Figure 1.4: Flowchart of the standard Marsu-bot operation.

Exploration

The main task of the Marsu-bots is to explore their current environment. To do so each Marsu-bot keeps a representation of its environment as an occupancy grid. Each one of the cells in the grid contain information about their exploration status (explored or not) and occupancy status (obstacle or no obstacle).

When a Marsu-bot is ready to explore it will select a “frontier cell” (that is, a non-occupied, explored, cell that is next to, at least, one non-explored cell) and move to that position, updating its map while driving. Once the target cell is reached, the Marsu-bot will share its map with the other Marsu-bots. Then the Marsu-bot is ready to explore again and start anew.

If, at any point during this operation, the battery of the Marsu-bot falls below a specific threshold it will stop its exploration and start a recharge operation.

Recharge

When the battery of a Marsu-bot is below a certain percentage the Marsu-bot stops its exploration and starts a recharge operation.

To do so the Marsu-bot first moves towars the Mother-bot. When the Mother-bot is reached, if there is enough space inside the Mother-bot, the Marsu-bot will enter it and recharge. If there isn’t, it will queue outside the Mother-bot and wait.

Every time a Marsu-bot exits the Mother-bot, the first Marsu-bot waiting in the queue enters the Mother-bot and recharges, and the rest advance in the queue.

Once the Marsu-bot has been charged to full capacity, it starts the exploration behaviour by selecting a new frontier cell.

1.2 Rescue protocol

Figure 1.5 shows the complete protocol for a rescue operation.

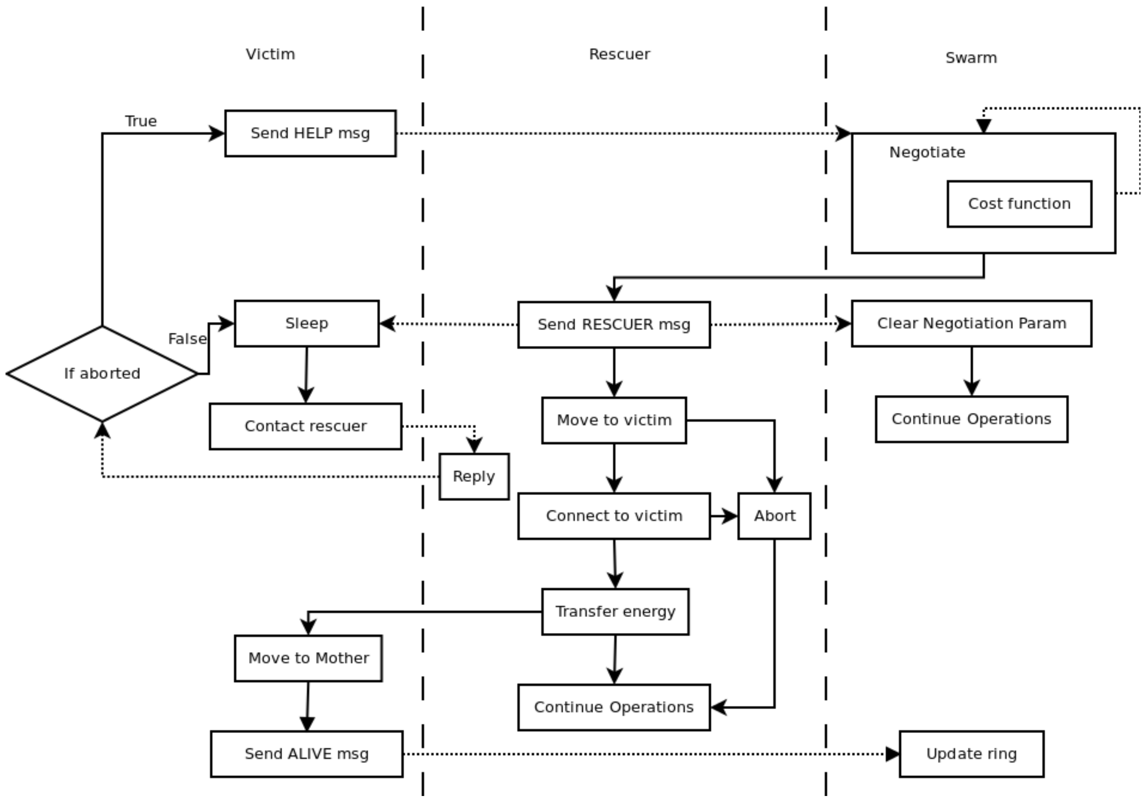


Figure 1.5: Flowchart of the rescue protocol. Dotted arrows represent messages.

When the battery of one of the Marsu-bots falls below the critical threshold that Marsu-bot becomes a victim. At this point, it will turn down most of its sensors and actuators, and send a HELP message through the network containing its current coordinates (position and orientation) and Marsu-bot id (“Send HELP msg” on figure 1.5). Then, the negotiation algorithm will be engaged by the swarm to select a rescuer (“Negotiate” on figure 1.5).

1.2.1 Rescuer behaviour

Once the rescuer has been selected, it will inform of its new state to the rest of the robots in the swarm (“Send RESCUER msg” on figure 1.5).

Then, it will move towards the position of the victim, connect to it and transfer energy to it. Once this is done, it will resume its normal operations.

If, while moving towards the victim or while trying to connect to it, the operation is determined to not be feasible, the protocol will be aborted. The rescuer will then resume normal operation as well.

1.2.2 Victim behaviour

When the victim receives the message informing of who the rescuer is, it will enter a sleep state ("Sleep" on figure 1.5). During the duration of this state the sensors and actuators of the victim will be powered down to save battery.

After some time has passed, the victim will wake up and send a message to the rescuer ("Contact rescuer" on figure 1.5). If the rescuer is still performing the rescue operation, it will reply and the victim will go back to the sleep state.

If the rescuer has aborted the behaviour but it's still functional, it will reply to the message with an aborted message. If the victim receives no reply to this message, which means that the rescuer Marsu-bot is no longer functional, or if it receives an aborted message, the victim will restart the protocol by sending the HELP messages again ("If aborted" on figure 1.5).

If the rescue operation succeeds, the victim will then move to the Mother-bot and fully charge its batteries in the usual way. Once this is done, it will send a message to inform the swarm that it is no longer in the victim state.

1.2.3 Swarm behaviour

Once the rescuer is selected, the rest of the Marsu-bots resume its normal operations. In addition, the negotiation algorithm is finished and all its variables are reset to its default values.

1.2.4 Cascade prevention

There are two mechanisms implemented in order to prevent a cascade event.

On one hand, a Marsu-bot is only allowed to start a rescue event a certain number of times before reaching the mother-bot. To control that the robot keeps track of how many HELP messages it has sent, if this number reaches a certain threshold it will completely power down. The counter is reset when the Marsu-bot reaches full battery again.

On the other hand a "dangerous area" mechanism has been implemented. When a Marsu-bot initiates a rescue event a dangerous area is defined around it. When the victim robot moves away from that area (after the energy transfer) it sends a message to all other robots to communicate that the area is safe. Any rescue event that happens inside a dangerous area will then be delayed until that area is marked safe again.

1.3 Scope of the Thesis

Most of the components required for the rescue protocol have been extensively researched. Algorithms such as path-planning (to devise an action plan to reach the victim or the Mother-bot), collision-avoidance or energy transfer mechanisms have been extensively researched.

There are two key components, however, that have never been studied. This study will focus on those two components.

One is the required negotiation algorithm that needs to be used to allow the autonomous selection of a rescuer. The other is the cost function required to predict the performance of a specific robot (which is required for the negotiation algorithm).

1.4 Next Chapters

Chapter 2 presents the study to select a negotiation algorithm, using leader election algorithms. This chapter presents the current state of the art in leader election algorithms, the selected algorithms for this study, the experiments performed and finally, the conclusions of this study and the specific selected algorithm.

Chapter 3 presents the study to define a cost function for the Marsu-bots and its parameters. In order to find the optimal parameters for such a function an evolutionary programming technique called Cultural Algorithm was used. The chapter presents the state of the art in evolutionary programming, the design of the performed experiments and its analysis and conclusions.

Chapter 4 presents a summary of the conclusions reached in previous chapters. In addition further work to extend the studies performed are presented.

CHAPTER 2

Negotiation Algorithm

As explained in the previous chapter a negotiation algorithm is required so the Marsu-bots can autonomously select a specific Marsu-bot to engage in the rescue behaviour. This negotiation algorithm needs to ensure that one, and only one, Marsu-bot will be selected, and the select Marsu-bot has to be the best one to carry out that task.

In this chapter the election algorithms will be presented, showing how this kind of algorithms may be used for this negotiation phase. Finally, an algorithm will be selected to be used in this protocol.

2.1 Introduction

On distributed environments, a common problem is the election of a leader (or coordinator) for the network. Several algorithms like *mutual exclusion* or *synchronization* algorithms depend on the existence of a unique coordinator node. As shown in [14], the *Initial Distinct Values* restriction is necessary for the election problem to be solvable, that is, each node in the system must have a globally unique value (sometimes called *identifier*) in order to break the system symmetry.

An algorithm needs to fulfil certain requirements to be considered an election algorithm. Those requirements are:

- Uniqueness: After the execution of the algorithm, one and only one node can be the leader.
- Finiteness: After a certain amount of time, the algorithm must finish.
- Awareness: After the execution of the algorithm, all nodes must know which node has been elected leader.

When applied to the problem at hand, those characteristics ensure that the algorithm will, after a certain amount of time, decide on a single Marsu robot from the pool of available robots to become a rescuer. Note, however, that there is no requirement for an election algorithm to select the best robot. Fortunately, most election algorithms can easily be modified to ensure that the best node (following a specific criteria) will be elected. In addition, there is a subset of leader election algorithms, the so-called extrema-finding algorithms, which ensure that the node either maximizing or minimizing a certain value will be selected.

By using the cost to rescue the victim as the optimization value, an election algorithm can be adapted to solve the problem at hand. To ensure the uniqueness of the solution the node id (a unique value for each node, such as its IP address) will also be taken into account, to break symmetry situations (two Marsu-bots having the same cost). For the sake of clarity, and without loss of generality, the cost will be considered a directly dependent to the distance and inversely dependent to the remaining battery (so, smaller cost means closer and more remaining battery).

It is important to notice that usually the unique id used to break the symmetry is a static, preassigned value, which implies that only the time to send and process a message will have an effect on the total time taken by the algorithm to finish its task. In the studied case, however, the cost needs to be calculated when required, and this is not a trivial task. This may significantly impact the performance of the algorithm.

2.1.1 State of the Art

Much research has been done on leader election algorithms, as those form the basis of distributed algorithms. The specific case of complete networks has seen an increase in interest since the apparition of wireless networks, which allow complete topologies to be built inexpensively (at least compared to the costs of building such a network with wires).

For this study, the algorithms presented in [15], [16] and [17] have been implemented and in some cases slightly modified. More information on these algorithms and the modifications implemented can be found in section 2.3.

Other well known algorithms, such as the *bully algorithm*[18] have not been implemented as it requires all nodes to have *a priori* knowledge of the id of the other nodes, which is unfeasible in the current scenario.

There hasn't been, however, too much research on the applicability of leader election algorithms for distributed robotics. Baca *et al.* used a leader election algorithm to improve the locomotion algorithm of the ModRED system [19](a Modular Self-reconfigurable robot) but simply used a modification of the *bully algorithm*, without further study of other possible algorithms.

2.2 Problem definition

When one of the Marsu-bots enters a victim state, the rest should initiate the election algorithm to determine which one of them is going to become the rescuer.

The Marsu-bots form a complete topography from an algorithmic point of view, as all are connected to the same WiFi network and aware of the address of every other robot in the fleet, allowing them to communicate between any two arbitrary robots. The completeness of this network allows the use of any network algorithm by building a virtual network on top of it (for example, building a virtual ring as done in section 2.3.3).

It is considered that the Marsu-bots know their own position in the map as well as the position of the victim, which will then be used to calculate the cost for the rescue operation on demand.

2.2.1 Requirements

For an algorithm to be considered suitable for electing the rescuer it needs to fulfil the following requirements:

Election requirements

It needs to fulfil the uniqueness and finiteness requirements as any election algorithm. The awareness requirement, however, may be reformulated. In the current scenario, there is no need for all the robots to know who the rescuer is, it is enough that all nodes know if they are the rescuer or not.

Completeness

All non-victim Marsu-bots must have the chance to participate in the algorithm to ensure that the best cost found is the absolute best cost of the whole swarm.

Optimization

For the algorithm to be suitable, it needs to guarantee that the elected rescuer will have the smallest possible rescue cost. If the function was defined as a fitness function (higher values for better conditions) the elected rescuer should have the highest possible value. In general, most leader election algorithms may be trivially modified to search the higher value instead of the lower one.

Encapsulation

The algorithm has to be as non-invasive as possible. An algorithm that requires extensive modifications on the existing software of the MARSU fleet, or that requires continuous calculations that interfere with the normal operation of the robot will not be considered. To comply with this requirement the cost of rescuing the victim needs to be calculated when the robot becomes aware of the issue, and cannot be precalculated.

Minimal interaction

As sending messages consumes power, algorithms that minimize the amount of messages required will be favoured. Additionally, due to the limited bandwidth available, and to avoid corruption from message collision, algorithms that minimize the amount of simultaneous messages required will also be favoured.

2.3 Algorithms

A total of 4 different algorithms, plus a centralized model, were tested in this study. Some of the studied algorithms required small modifications in order to comply with the specified requirements.

Throughout this explanations, a node (or value) will be considered better if it minimizes the cost function (meaning lower values are better). The id of the node (a unique integer assigned at start) will be used when two costs are equal, in this case the higher id is the “better” value.

2.3.1 Centralized optimization

To prove that a decentralized algorithm is preferable in the current study, a centralized algorithm was also tested under the same scenarios.

The algorithm

The algorithm is started by a broadcasted help message sent by the victim. Then all nodes compute its rescue cost and send it to a central robot (the Mother robot), who then selects the node with better cost and informs it that it has become the rescuer.

Requirement compliance

This algorithm complies with all requirements except the minimal interaction one, as several messages (one for each non-victim bot) will be sent simultaneously.

In addition, the main issue associated with such a centralized approach is that if, for any reason, the Mother became unavailable at some point during the execution, the algorithm would be unfinished and the Marsu-bots would assume that another robot has been selected as the rescuer when in reality there is none.

2.3.2 Brute force algorithm

The Brute force algorithm has been designed for this experiment to provide a baseline for comparison between the algorithms. This algorithm is a modified version of the *bully algorithm*[18], based on the broadcasting capabilities at the disposal of the MARSU fleet and adapted to the requirements presented in section 2.2.1.

The algorithm

The algorithm is started by a broadcasted help signal sent by the victim. When a node receives this help message it will compute its cost to rescue the victim and broadcast a message with this cost and its id, henceforth called *cost message*.

When a node receives a cost message, if it hasn't computed its own cost yet it will do so. After calculating its own cost, it will compare it to the values in the message, and, if better, it will broadcast a new cost message with its own information. If its own cost is worse it will store this value in its memory and broadcast the received cost message again (with the cost and id of the other node).

If the node receives a cost message and already knows of a node with better cost than itself, or it has already calculated its own cost, it will compare the known values to those of the message (best known and own respectively), and if they are better it will update its memory and broadcast the received message again.

If a node spends a certain amount of time without sending or receiving any messages and does not know of any node with a better cost than its own it will elect itself as a rescuer and engage the rescue behaviour.

The id of the nodes is used in the unlikely event that two or more nodes share the same cost, in which case the node with a higher id is considered better.

Requirement compliance

This algorithm complies with the finiteness requirement. The best node will keep discarding messages and never store anything in memory so, after a certain amount of time, there will be a node that will elect itself leader. Due to the fact that all messages are broadcasted, as long as the network forms a complete topology the algorithm will also comply with the completeness requirement. Lastly, the algorithm meets the encapsulation requirement, as no *a priori* knowledge of the system is required.

Unfortunately the uniqueness requirement may not be satisfied. If the time taken for a node to elect itself after sending the last message is too short, or the network forms a non-complete topology it is possible that a node elects itself as a rescuer prematurely, and a second node becomes a rescuer as well. This fact implies that the optimization requirement may not be met as well.

Lastly the minimal interaction requirement is most definitely not satisfied with this algorithm. High amount of simultaneous messages will be sent, most of them completely unnecessary, as each node will broadcast the value of the best node it has heard of so far if its better than the received message.

2.3.3 Algorithm for complete networks without sense of direction

Villadangos *et al.* presented a " leader election algorithm for complete networks without sense of direction"[15]. The proposed algorithm, however, does not fulfil the optimization requirement, so certain modifications need to be made.

The algorithm

The algorithm assumes a virtual ring connecting all the nodes exist. This ring is ordered, that is, messages can only be sent through the ring to the next node in the ring (though the network its still complete, so messages can be sent between any two arbitrary nodes, regardless of their position in the ring). When a robot becomes a victim, it will send a message to the previous node on the ring informing of its state and the next member of the ring, so the ring can be patched leaving the victim out of it.

Initially all nodes are in a *passive* state. When a node receives the message from the victim it upgrades its neighbour in the ring, updates its state to *candidate* and calculates its cost. Once done it sends an *ALG* message containing its cost and id to the next node in the ring.

When an *ALG* message reaches a node, if this node is in a *passive* state, it calculates its own cost. If its cost is worse than the received value it updates its status to become *dummy* and forwards the message to its successor in the virtual ring. Otherwise, it becomes a *candidate* and sends an *ALG* message containing its cost and id to the next node in the ring. It also sends an *AVS* message to the node contained in the received message and waits for that node to reply informing of the best node he knows of.

When an *AVS* message is received by a node, it will store the information contained in this message. If the receiving node knows of any better node it will answer with an *AVSRSP* message containing the cost and id of the better node and become *dummy*. Otherwise, it will store the information contained in the *AVS* message.

Finally, when a node receives an *AVSRSP* message, if the contained value corresponds to itself, it becomes the rescuer. Otherwise, if it knows of a better node, it will become *dummy* and forward the *AVSRSP* message to the better node. If it doesn't know of a better node, the information contained in the *AVSRSP* message it's not its own and its own cost is better, it will reply with an *AVS* message containing its own information.

It is important to note that, on the original algorithm, only a subset of nodes are considered and they become *candidates* by themselves. If a *passive* node receives an *ALG* message, it just becomes *dummy* and forwards the message. This implied that the selected node would have the best cost of all the considered nodes, but there could have been other nodes with better cost. To comply with the optimization requirement, though, it is necessary that all nodes have the opportunity to become candidates. In order to ensure that the algorithm has been modified so when a node receives the *ALG* message it first calculates its own cost to decide if it should become a *dummy* or a *candidate*.

Requirement compliance

The algorithm complies with the finiteness and uniqueness requirements, as proven in [15]. Thanks to the modifications done, the algorithm also complies with the completeness and optimization requirements, as the virtual ring covers all the nodes and all of them have an opportunity to become candidates.

The algorithm requires a certain architecture built, the virtual ring, which breaks the encapsulation requirement, though this is not too critical as the ring architecture is not expensive to build and maintain.

Finally, from an interaction point of view the algorithm is quite minimal. Few simultaneous messages will appear and not many messages are sent overall.

2.3.4 Decentralized extrema-finding

Belonging to the extrema-finding subset of leader election algorithms, the algorithm presented by Hirschberg and Sinclar in [16] requires very few modifications to comply with the desired requirements. This algorithm is designed for undirected ring structures, which can be simply simulated by building a virtual ring as done in section 2.3.3.

The algorithm

The algorithm assumes all nodes begin in a *passive* state, and a bidirectional ring connecting all nodes exists. This is accomplished by building a virtual ring, in which each node is connected to the nodes with immediate smaller and bigger id. When a robot becomes a victim it will send a help message to both neighbours, informing of the id of the other neighbour, so the ring can be rebuilt leaving the victim out.

When a node receives a help message, it will calculate its rescuing cost, update its status to *candidate* and send a *FROM* message with its cost and id in the opposite direction to the dead node.

When a node receives a *FROM* message, if it is in a *passive* state, it will calculate its own rescuing cost. If the cost is worse, it will update its state to *lost* and forward the message in the same direction of the ring without modifying it. If the cost is better, it will update its state to *candidate*, send a *NO* message to the originating node and begin a new election process by sending a new *FROM* message with its own information in both directions of the ring. If the node is already in a *lost* state, it will simply forward the received message along the ring, keeping the direction of the message. Finally, if the node is still a *candidate* and receives a message with its own information, it will elect itself as the rescuer.

When a node receives a *NO* message it will update its status to *lost*.

Though the idea of upgrading a node to a *candidate* status if its cost is better is presented in the original paper, some additional modifications were needed. The original algorithm uses an incremental search pattern, scanning paths of a certain length and, if no better node has been found in the path, restarting the algorithm with a longer path. That idea was determined unnecessary and counterproductive for the current scenario, as it increases the number of necessary messages and the time required to end with no associated benefit. In addition, when the algorithm starts the two neighbours of the victim send only a single message, instead of one in each direction. This is due to the fact that, after modifying the ring to leave the victim out of it, the two starting nodes are neighbours. Then, by sending one message in the opposite direction the whole ring is covered, as there are no other nodes between the two initiators.

Requirement compliance

The algorithm complies with the finiteness, uniqueness and optimization requirements by design, as it is an extrema-finding algorithm. With the upgrade to *candidate* status modification, it complies with the completeness requirement as well.

As with the algorithm presented in section 2.3.3 the encapsulation requirement is broken by the necessity of a virtual ring for the algorithm to operate, but once again the ring architecture is simple to build and maintain, even if it is more complex due to the fact that it is a bidirectional ring.

Lastly, from an interaction point of view, the algorithm does not generate a high number of total messages, though it may have more simultaneous messages than other algorithms in a worse case scenario.

2.3.5 Algorithm for complete networks with sense of direction

Louie *et al.* presented an algorithm that "uses $O(n)$ messages to solve the Election Problem in a complete network with a sense of direction"[17]. This algorithm is an extrema-finding algorithm, and makes use of the fact that the network is completely connected, thus requiring no modifications.

The algorithm

This algorithm assumes a directed virtual ring structure, in which each node has the next node in the ring (in an arbitrary direction, though all nodes use the same) identified as its prey. When a node enters the victim state, it will broadcast a *help* message containing the id of its prey.

When a node receives a help message, it will first become *alive*, calculate its own rescue cost and then send a *hunt* message to its prey containing its own cost and id. Due to the construction of the ring, there will be exactly one node whose prey is the victim. This node will update its prey information to point towards the prey of the victim (sent in the *help* message) and send the *hunt* message to its new prey.

When an *alive* node receives a *hunt* message, it will compare the received cost and id to its own. If they are better, it will become *dead* and send a *hunt_successful* message to its hunter, informing it of who its prey is. If the id of the hunter matches the id of the receiving node, that means that all other nodes are *dead* and it has won the election. If its rescuing cost is better than the received one, it will do nothing and wait for new messages. If a *dead* node receives a *hunt* message, it will forward it to its prey.

When a node receives a *hunt_successful* message, it will update its prey id to that of the defeated prey and, if it is still *alive*, it will send a new *hunt* message (with its cost and id) to its new prey. If the new prey received in the *hunt_successful* message is itself, then all other nodes are *dead*, which means it has won the election.

Requirement compliance

The algorithm complies with the finiteness, uniqueness, completeness and optimization requirements by design.

As with the algorithm presented in section 2.3.3 the encapsulation requirement is broken by the necessity of a virtual ring for the algorithm to operate, but once again the ring architecture is simple to build and maintain.

Lastly, from an interaction point of view, the algorithm generates a high number of simultaneous messages (as many messages as nodes in the network minus one in the first step). On the other hand it requires few total messages and time.

2.4 Experiment

In order to identify the best algorithm for the scenario a series of simulations were carried out. As the algorithm will be implemented on the MARSU fleet some key parameters of the scenarios were derived from that. In all scenarios a complete topology (as discussed in section 2.2) was used.

2.4.1 Simulations

To ensure the usability of the algorithm two different scenarios were used on the experiments.

Scalability

In order to test that the algorithm is scalable, each algorithm was tested in a network with 4, 6, 10, 20, 50 and 100 nodes. For each of these networks, the costs were distributed in such a way that no two costs are equal (permutations of number of nodes elements). As the permutations increase in a factorial way, which would lead to an unpractical number of simulations to be performed, no more than 10000 simulations were performed in any combination of network and algorithm. If the number of permutations were bigger than this threshold, a representative subset was used, that is, a subset of permutations such as all possible values appear in all possible positions at least once.

In addition, for every node number and algorithm combination, a case in which all costs are equal except for a single node with better cost was also tested.

Timings

The scalability test uses always the same computing time (time required to calculate the rescuing cost) for all nodes. The second scenario is designed to investigate the effect in the performance of the algorithms that different computing times for each node may have.

To that end a similar experiment was conducted in networks with 6, 7, 8, 9, 10, 12 and 15 nodes. Then, the computing time taken by each node was set to be proportional to the cost of each node. This proportionality came from the fact that the most expensive task (computationally speaking) in the calculation of the rescue cost is the calculation of the distance, so, the farther away a node is the bigger the cost and the longer the computation takes.

Once again, the same cost distribution used in the Scalability scenario was used. This implies that, in the case in which no two costs are equal, no two computation times are equal either, while in the case of a single best cost all computation times but the best one are equal.

2.4.2 NetSim

NetSim is a Java program created to carry out these experiments. The program allows the user to generate a custom defined network (with any arbitrary number of nodes and topology), specify the times taken for message sending, message processing and internal calculations, and simulates an execution of a specific Algorithm on that network.

Each algorithm is implemented as a class extending an abstract *Communication* class, which makes the program easily customizable for many algorithms.

Each scenario is defined with an *ae* file (stands for algorithm experiment), which encodes all the required information for either a single experiment or a set of experiments with common values, such as number of nodes, total simulated time, algorithm used, timings, network topology (as a connection matrix describing a directed graph) and more. Once the execution is finished, the results are both shown on screen and stored in a Matlab file for posterior analysis (both outputs may be suppressed if desired).

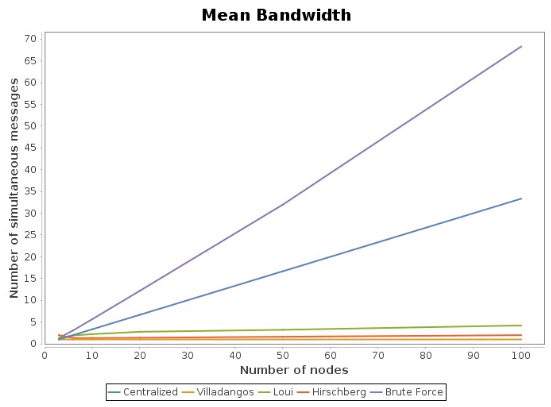
A series of plots for different parameters (total time, total number of messages and bandwidth occupancy) are generated. First linear plots are generated for minimum, maximum and mean time and number of messages, and also for maximum and mean bandwidth occupancy. These plots show number of nodes versus parameter value. In each of those plots the different algorithms are shown as different series. Second, for each algorithm and parameter a linear plot showing maximum, minimum and mean as different series is created. These plots depict number of nodes versus parameter value as well. Finally, a column plot for each algorithm shows the percentage of correct, unfinished, non-optimal and non-unique executions for each set of experiments with a specific number of nodes.

2.5 Results

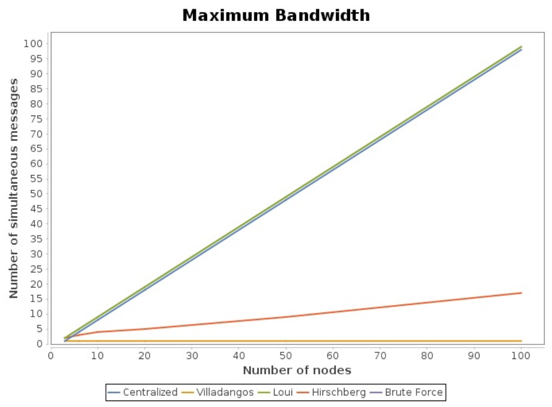
As explained in previous sections NetSim generates several plots to study the parameters of interest. In the following sections the results of both scenarios will be studied independently.

2.5.1 Scalability

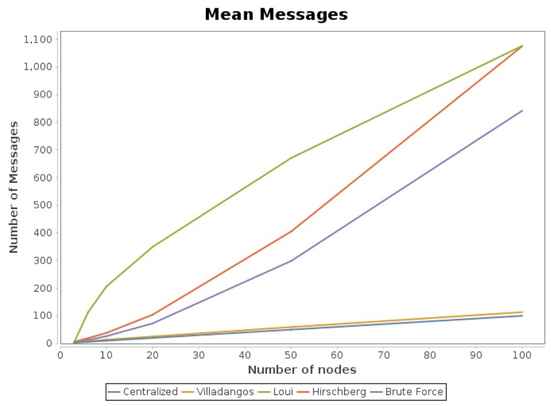
The mean and maximum values for the parameters of interest (total time, total number of messages and simultaneous number of messages) are shown in figure 2.1.



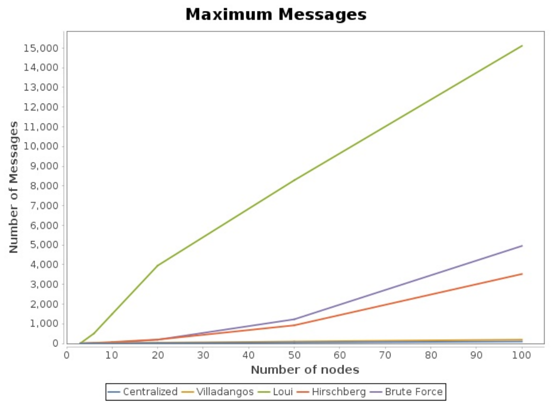
(a) Mean number of simultaneous messages.



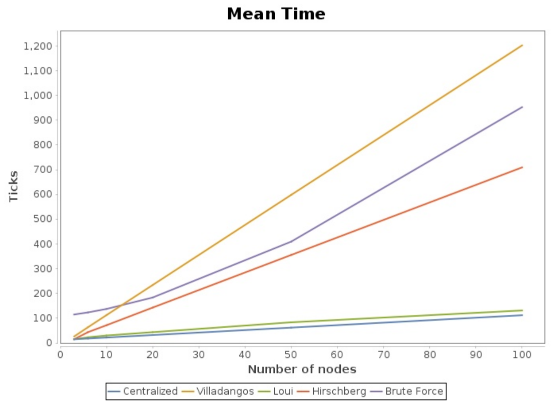
(b) Max. number of simultaneous messages.



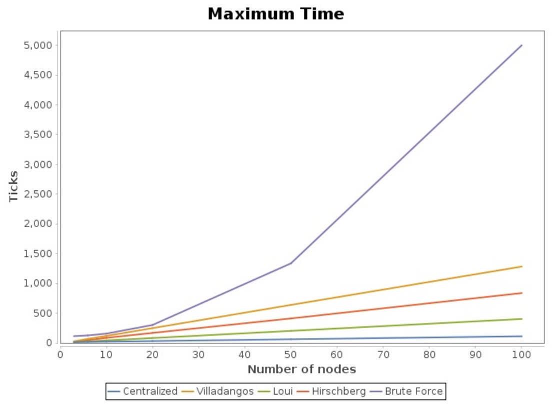
(c) Mean number of total messages.



(d) Maximum number of total messages.



(e) Mean time taken to select a rescuer.



(f) Maximum time taken to select a rescuer.

Figure 2.1: Mean and maximum value analysis.

The first conclusion is a trivial one: the number of simultaneous messages and total time taken are inversely proportional. That was to be expected as, the higher the parallelization, the shorter it takes for the algorithm to finish, but more nodes finish simultaneously which leads to a higher number of simultaneous messages.

Figure 2.1b shows that the algorithms “Complete without direction” and “Decentralized extrema-finding” provide the smaller bandwidth occupancy, even in a worse-case scenario. On the other hand figure 2.1e shows that both algorithms have the biggest total time in general. From the point of view of total number of messages both algorithms are quite low.

The “Complete without direction” algorithm is a very promising algorithm in terms of bandwidth and messages, but in its current implementation takes too long to complete. One solution for that is to have the victim node send the starting message to two different nodes on the ring instead of one. This will, of course, slightly increase the mean bandwidth and maximum bandwidth used but, as the current values are quite small (only two simultaneous messages at worst) this won’t have a huge impact on the performance of the system.

Figure 2.2 shows the results of the same tests adding the “Complete without direction” algorithm with a set of two starter nodes instead of one. To increase the performance of this approach the second starter node is selected to be the node that is further away in the ring in both directions.

As can be seen in figure 2.2f, initializing the algorithm on two nodes (shown as “Villadangos (2)” in the figure) reduces the total time taken by half, with close to no impact on the total number of messages and bandwidth parameters.

Figure 2.3 shows the specific plots for the selected algorithm. As can be seen the algorithm is quite stable as the variance between mean, maximum and minimum values is small, even in the 100 node case. Another interesting feature is that the value of the mean is biased towards the minimum, which means that most executions have values closer to the minimum value, which is a desirable trait.

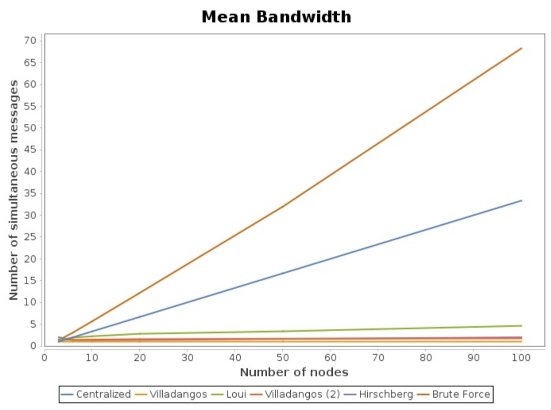
2.5.2 Timings

Figure 2.4 shows the results for the timings experiments.

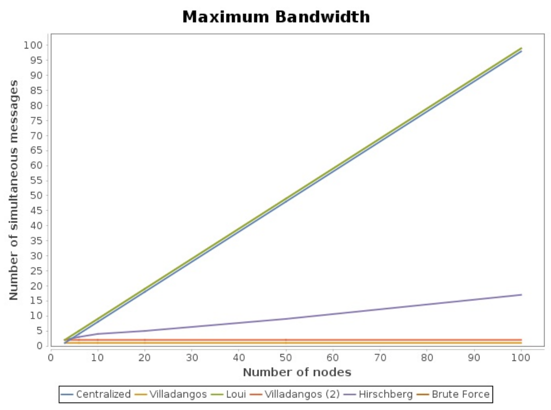
It is interesting to note that, in this experiment, when the number of nodes increase the number of simultaneous messages initially increases, and, when a certain number of nodes are present on the network, it decreases steadily when more nodes are added. That is a direct consequence of the fact that in most experiments the computation times of the different nodes are different. As a consequence, messages that have to be sent after the end of the computation are evenly distributed in time, instead of sent out simultaneously.

As in the previous scenario the “Complete without direction” algorithm initializing two nodes simultaneously has the best overall results, as the bandwidth used is quite low and stable, and the time required for the algorithm to finished is one of the best (only “Complete with direction” and “Centralized” have better times, but the bandwidth used is worse).

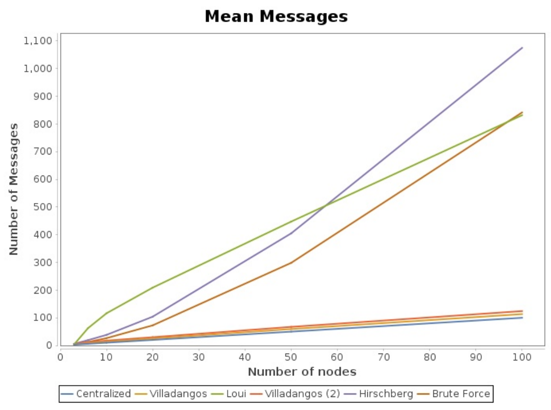
Finally, figure 2.5 shows the specific plots for the “Complete without direction” algorithm with 2 initialized nodes. As in the previous scenario the results are quite



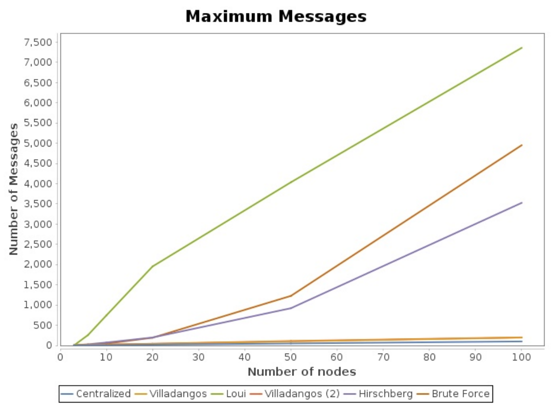
(a) Mean number of simultaneous messages.



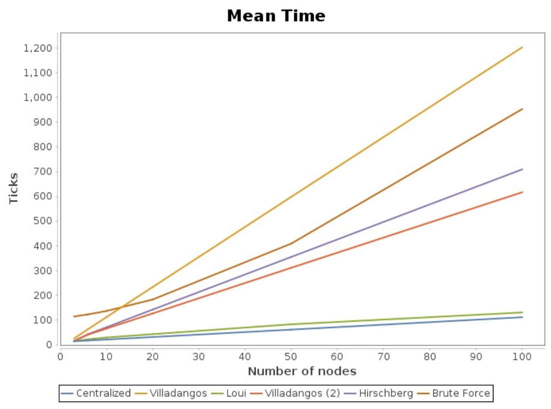
(b) Max. number of simultaneous messages.



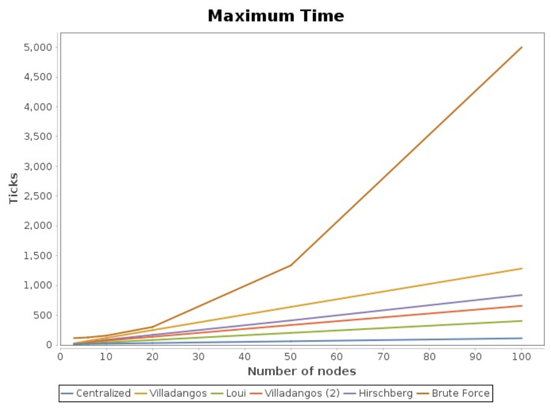
(c) Mean number of total messages.



(d) Maximum number of total messages.



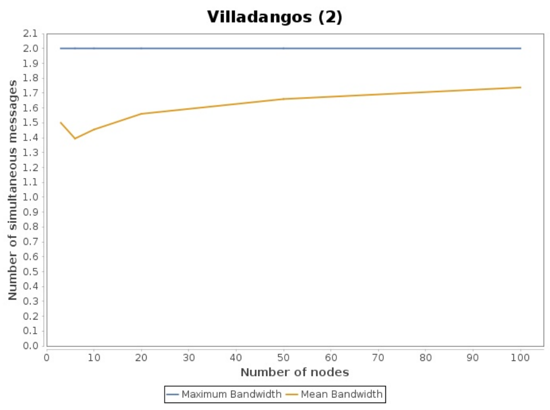
(e) Mean time taken to select a rescuer.



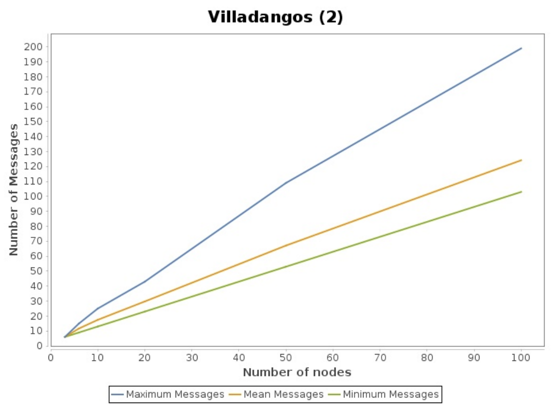
(f) Maximum time taken to select a rescuer.

Figure 2.2: Mean and maximum value analysis.

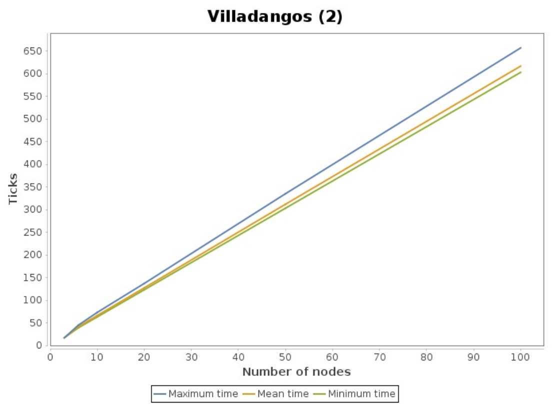
stable and, in general, the mean values are biased towards the minimum, which implies that most executions are closer to the optimal case.



(a) Mean & max. simultaneous messages.



(b) Min., mean & max. total messages.



(c) Min., mean & max. total time taken.

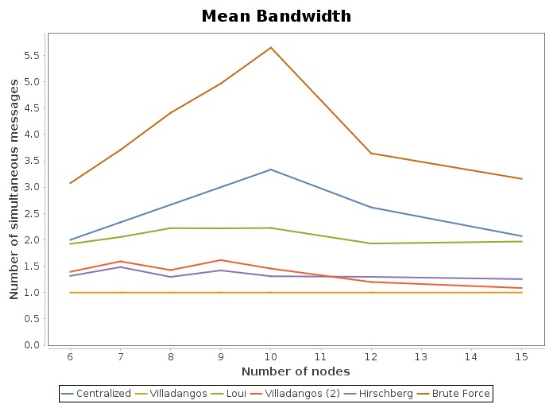
Figure 2.3: Complete without direction analysis.

2.6 Conclusion

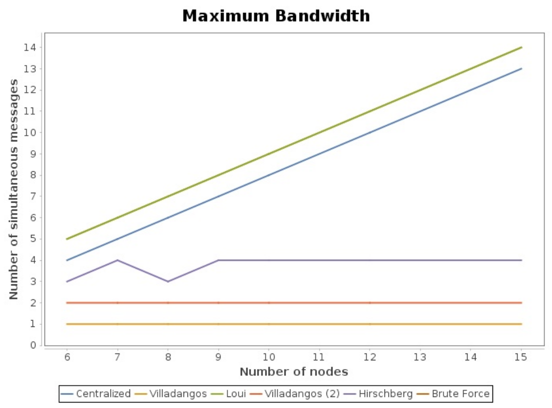
The tests show that using the algorithm presented in section 2.3.3, written by Villadangos *et al.*, and sending the start message to two different nodes provides the best results overall in both scenarios.

Initializing two nodes simultaneously has also a secondary advantage, as then the algorithm becomes more resilient to unexpected errors and “disappearances” from nodes.

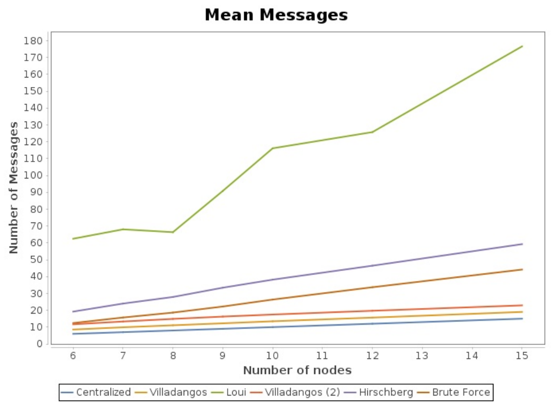
For those reasons this algorithm will be used for the negotiation phase of the rescue protocol.



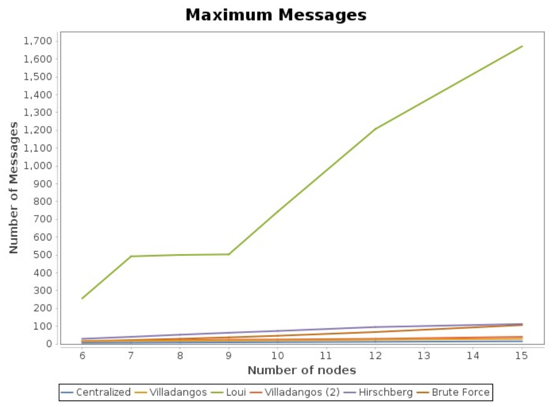
(a) Mean number of simultaneous messages.



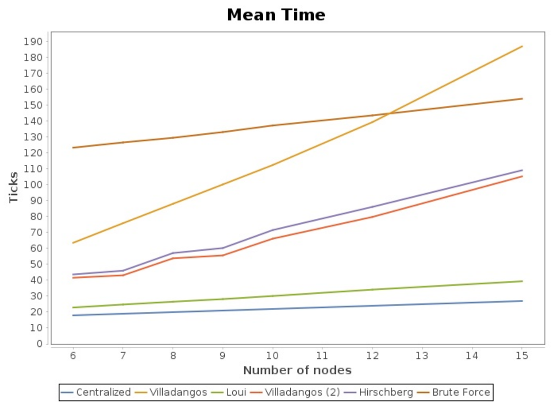
(b) Max. number of simultaneous messages.



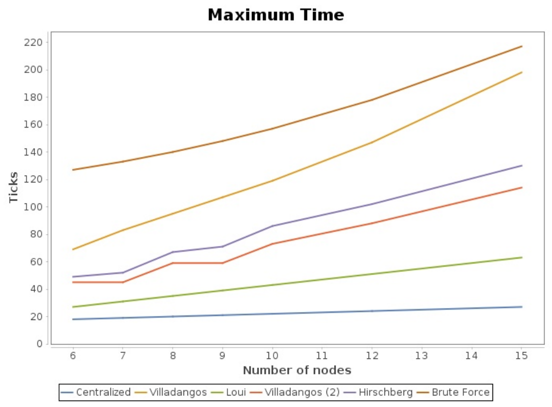
(c) Mean number of total messages.



(d) Maximum number of total messages.

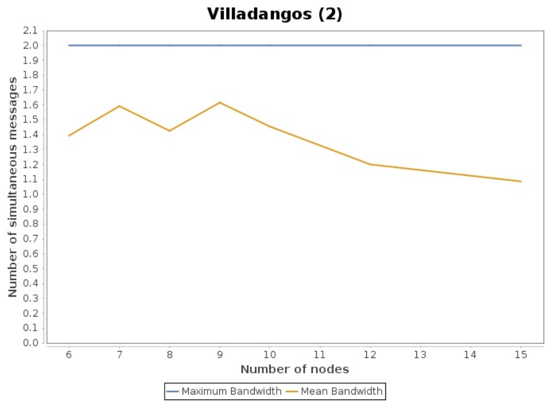


(e) Mean time taken to select a rescuer.

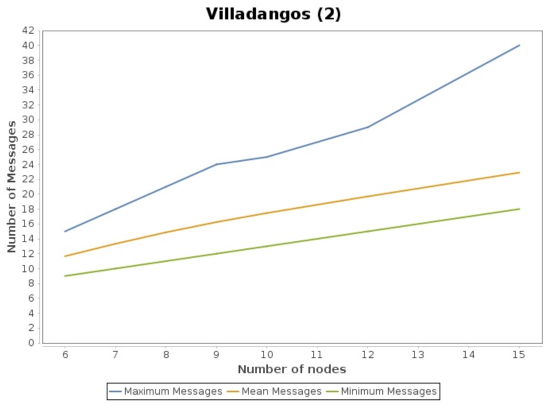


(f) Maximum time taken to select a rescuer.

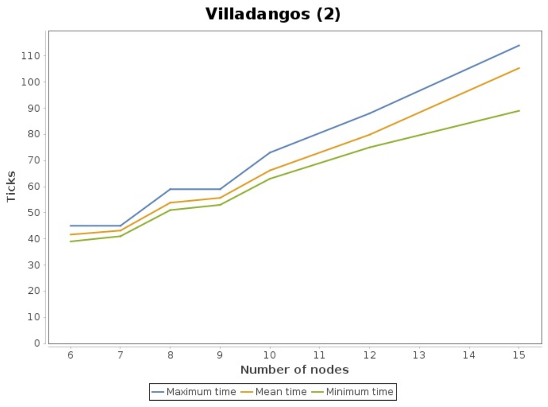
Figure 2.4: Mean and maximum value analysis.



(a) Mean & max. simultaneous messages.



(b) Min., mean & max. total messages.



(c) Min., mean & max. total time taken.

Figure 2.5: Complete without direction analysis.

CHAPTER 3

Cost Function

As explained in the previous chapters, for the negotiation algorithm to autonomously select the most suitable robot for the rescue operation, a cost function needs to be designed. This cost function should have an absolute minimum, which correspond to the most suitable robot.

3.1 Introduction

It is clear that the cost function should measure the distance of each Marsu-bot to the victim, as well as the battery available to be transferred.

It is hard to hypothesize which is the relation between distance and battery in the cost formula. In addition certain design parameters, such as the specific distance formula used, or the amount of battery transferred will affect this relation.

In order to find that formula, evolutionary computation will be used. The quantity of tests to be performed in order to obtain relevant data, and the complexity of those tests require heavy computing resources. The Triton supercomputer, a supercomputer provided by the Aalto University School of Science "Science-IT" project was used in order to be able to process the required amounts of data.

3.1.1 State of the Art

"Evolutionary computation has as its objective to mimic processes from natural evolution, where the main concept is survival of the fittest: the weak must die" [20]. Evolutionary computation has been used to solve optimization problems, providing a great performance improvement on complex problems over analytical optimization algorithms.

There are several different classes of evolutionary algorithms: Genetic Algorithms (GA), Evolutionary Programming (EP) or Genetic programming to name but a few (see [20]).

GAs model genetic evolution of a population focusing on “reproduction”, that is, merging two solutions to generate a new, different third solution. GAs are heavily influenced by the type of reproduction used, more specifically, by the selection algorithms to determine the parents of a new solution. In the presented algorithm “tournament selection” is used, a selection algorithm that consists in selecting n individuals at random and then comparing their fitnesses to choose the best one [21].

EPs, on the other hand, focus on the use of “mutation”, that is, modifications to the current value of a solution. This difference leads to different applicable domains. GAs have a better performance in domains with discrete and bounded values, while EPs exhibit better performance in domains with continuous values.

Cultural Algorithms (CA) are an evolution to Genetic Algorithms (GA) presented by Reynolds in 1994 [22]. CAs use both a solution set (called population) and a knowledge set (called belief space). The population is then evolved used standard evolutionary computation techniques and at each generation the knowledge obtained is extracted and used to modify the following population.

The belief space consists in five different knowledge sources: situational, normative, topographical, domain and historic knowledge [23]. A CA then defines a set of communication strategies between the population and the belief space to extract the knowledge gained by each generation. Afterwards the gained knowledge is used to influence the new population.

One of the main drawbacks of evolutionary computing is the tendency of those algorithms to get “stuck” in local minima, terminating prematurely and returning a non-optimal solution. One technique to prevent such a situation is the use of “niche clearing” [24], in which each niche contains part of the population and only a few individuals (the fittest) of each niche are allowed to survive.

Ali *et al.* presented a “Hybrid Niched Cultural Algorithm” (HNCA) framework [25], which uses the niching technique in a CA to reduce the population size and improve performance while avoiding local minima. In addition a “tabu list”, a list of rules that all individuals in a population have to comply with, is used to avoid individuals that are not beneficial. In the study this is done by using both a set of specific rules and also adding each individual that will be tested to the list, to avoid unnecessary repetition.

Another technique used in evolutionary computation to avoid local minima is the use of “extinction events” [26], which consist in applying a certain stress value to each species (subset of the population) which determines the likelihood of survival, species with lower fitness become extinct.

3.1.2 Definitions

Evolutionary computation draws its ideas from real-world natural evolution, and as such several names from the field of genetics and evolution have been adapted. This section presents a lists of such names and their meanings when referring to evolutionary computation.

Individual A possible solution to the problem studied by the algorithm.

Chromosome A specific variable part of an Individual.

Population A set of individuals.

Generation A time step in the algorithm. Usually refers to the population at that time step.

Reproduction The process of creating a new individual by merging two (or more) individuals together (all individuals may be the same individual).

Parent One of the individuals involved in a reproduction operation.

Mutation The process of modifying a chromosome without regard to the chromosomes of the parents.

3.2 Pre-analysis

As this specific field has never been studied before, it is not know if any such formula exists, or if the selected factors (battery and distance) are enough to approach it. In order to find out if such a formula exists, and which shape it may have, an initial study has to be performed.

Using the MarSim simulation built by David Leal Martínez [12] a series of tests were conducted on the swarm using different strategies and scenarios.

3.2.1 Strategies

For this study three different sets of strategies were considered: the distance function used, the specific rescue protocol and the amount of battery to transfer.

Distance function

Three different distance functions were used:

- Optimist distance: The euclidean distance between the two robots, without any regard to obstacles that may exist between the two.
- Pessimist distance: The euclidean distance between the two robots, squared, once again without any regard to obstacles that may exist between the two.
- Realist distance: The distance of a planned path between the two robots, avoiding any obstacles and constrained to the known map. As all robots begin in the same position and explore the world from there, a path exists that is constrained to the known map.

Protocol

Three different protocols were used:

- Always save: A robot of the swarm is always selected and sent to rescue the victim. In this case a robot may be selected that has not enough battery to reach the victim. Distance from victim to robot and percentage of depleted battery of the robot are used to compute the cost.
- Always mother: When a robot of the swarm is selected as the rescuer it always begins by moving to the Mother-bot and fully recharging its batteries prior to reaching the victim. Distance between the robot and the Mother-bot and the amount of depleted battery when reaching the victim are used to compute the rescue cost.
- Hybrid: When computing the cost, first the distance between victim and robot and the amount of energy required to reach it is computed. If the robot has enough battery to reach the victim the "Always save" protocol is used, otherwise, the "Always mother" protocol is used.

Transfer type

Two different transfer types are used:

- Half: Transfer half the available battery.
- Enough: Transfer enough energy to allow the victim robot to reach the Mother-bot, plus a certain margin for safety.

3.2.2 Scenarios

There are four different scenarios used: an office like scenario (see picture 3.1a), an environment with randomly positioned obstacles, sparsely distributed (see picture 3.1b), a randomized labyrinth (see picture 3.1c) and a scenario with no obstacles (see picture 3.1d).

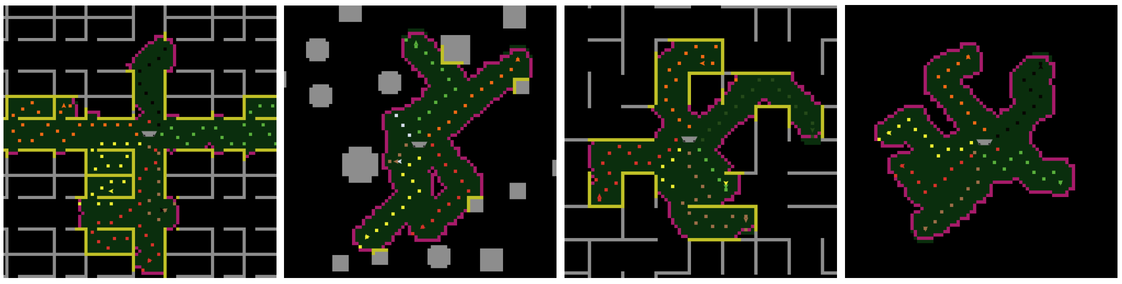

(a) Office. (b) Randomized. (c) Labyrinth. (d) Empty.

Figure 3.1: Testing scenarios.

The distribution of free and occupied spaces in a scenario will have a huge impact in the distance calculation. Scenarios with less obstacles will have straighter paths, and using an optimist distance will be more similar to using a realist one.

In order to classify the different scenarios a sinuosity test has been conducted. For each scenario, a mesh of points has been defined. Then, the shortest path distance, accounting for obstacles, has been compared to the shortest path distance ignoring the obstacles, between any two of the selected points. Finally the average sinuosity of all this points has been computed. Equation 3.1 shows the formula used.

$$S_{scenario} = \frac{\sum\limits_{p_1,p_2 \in M} \frac{D_{path}(p_1,p_2)}{D_{straight}(p_1,p_2)}}{|M|} \tag{3.1}$$

, where $S_{scenario}$ is the sinuosity of the scenario, p_1,p_2 are different points in the selected mesh (M), $D_{path}(p_1,p_2)$ is the shortest path distance accounting for obstacles between p_1 and p_2, $D_{straight}(p_1,p_2)$ is the shortest path distance ignoring obstacles between p_1 and p_2 and $|M|$ is the number of selected points.

Table 3.1 shows the results of this calculation.

Table 3.1: Scenario sinuosity.

Scenario	Sinuosity
Office	1.4731
Randomized	1.0116
Labyrinth	1.3944
Empty	1

Besides the average sinuosity, it is also important to see the probability distribution of the sinuosity. Figure 3.2 shows the discrete, non-cumulative, probability distribution for the Office, Labyrinth and Randomized scenarios (the Empty scenario always has a sinuosity of 1).

3.2.3 Test

In each test, one robot was artificially selected as the victim, and each other robot was used to perform a rescue operation. These rescue operations were ranked by its performance, according to time taken and amount of battery left after the rescue operation on the rescuer (from now on, performance rank).

In addition, robots were grouped according to their ability to perform the rescue mission without generating additional rescue events (from now on, class).

Several tests were conducted for each combination of strategies and for each different scenario.

3.2.4 Results

In order to see if there is a clear relationship between the amount of spent battery at the beginning of the rescue mission, the distance to the victim and the performance

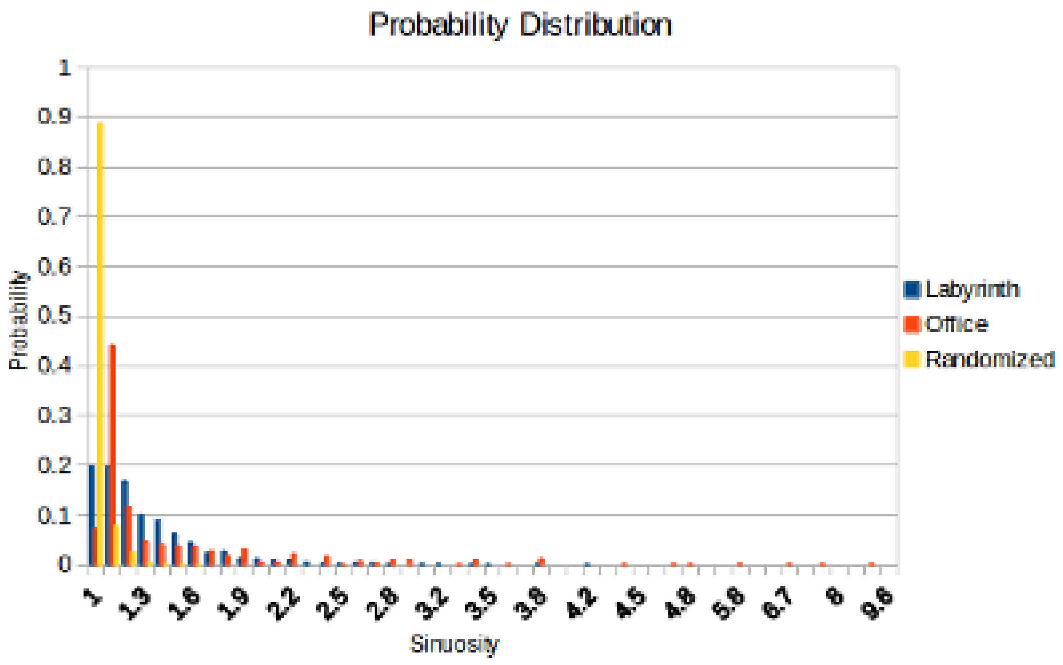


Figure 3.2: Probability distribution of sinuosity.

of the robot, it is necessary to apply some transformations to the obtained data.

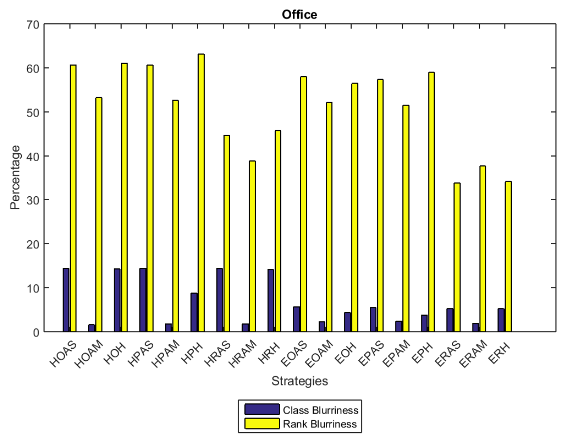
First, the battery and distance values need to be transformed to relative values with regard to the values of the other robots in the same test. In order to do that, for each value on each robot, a rank was assigned according to its relative position inside the swarm, that is, the robot with the smallest distance was assigned a distance rank of one, the robot with next smallest distance was assigned a distance rank of two, and so on.

Then, in order to avoid all data points being on top of each other (as all robots with distance rank 1 and battery rank 1 would be plotted in the same point) a certain jitter is applied (a random value between 0.1 and 0.9) so that the plots can be easily read.

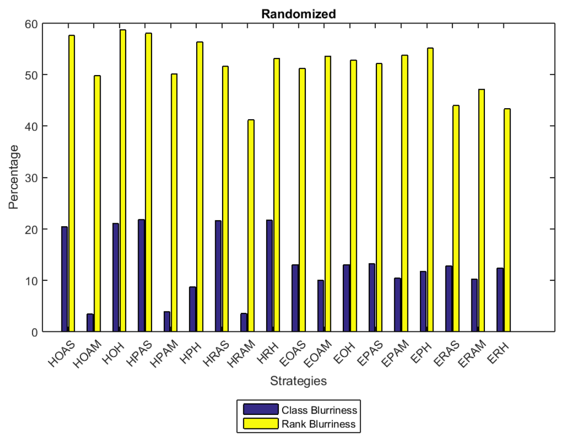
Finally, the average blurriness of each plot with regard to performance rank and class was computed. Blurriness measures the amount of error in each battery-distance rank combination, the higher the value the bigger the amount of different performance rank values present, which means that it becomes harder to separate the different rank performances.

Due to space constraints, the strategy descriptions have been encoded in the following way: First, the transfer type, with values of either H (for Half) or E (for Enough). Then the distance function, with values of either O (for Optimist), P (for Pessimist) or R (for Realist). And finally, the protocol used, with values of either AS (for Always Save), AM (for Always Mother) or H (for Hybrid). Thus, the strategy HPAM means transferring half the battery, using a pessimist distance function and always going to the Mother-bot before rescuing.

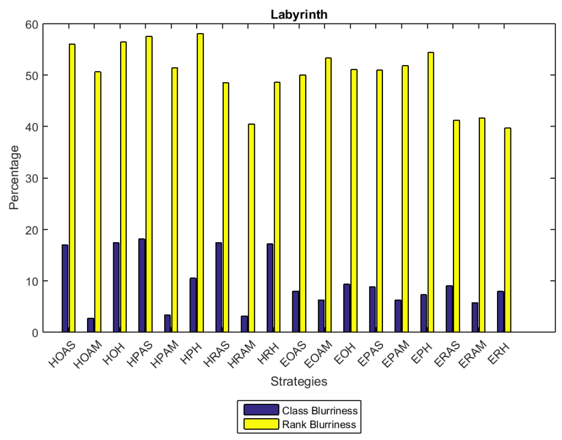
Figure 3.3 shows the blurriness values for each combination of strategies in each scenario.



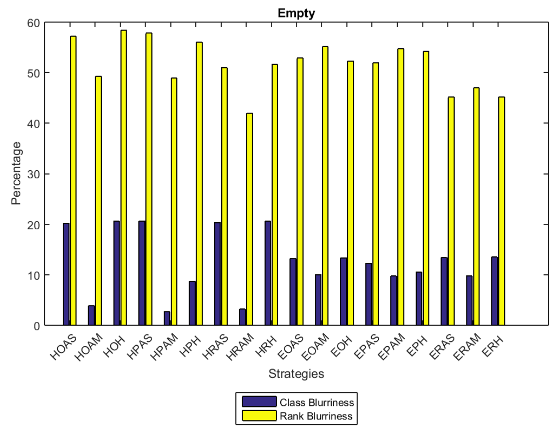
(a) Office.



(b) Randomized.



(c) Labyrinth.



(d) Empty.

Figure 3.3: Class and Rank Blurriness by scenario and strategy.

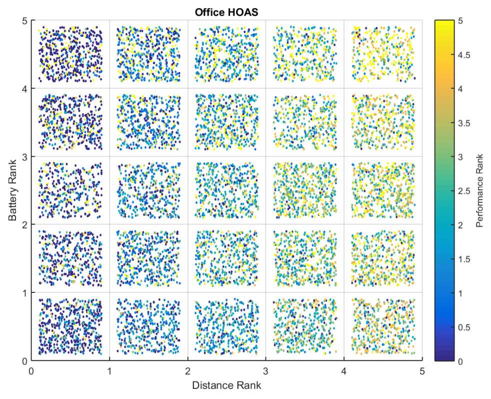
As can be seen, class blurriness values are, in most cases, below 20%, which means that it is easy to find a relationship between the distance-battery rank pair and the capacity of the Marsu-bot to complete the rescue operation successfully.

On the other hand, rank blurriness is quite significant. In all cases the average percentage of non-modal ranks for a given battery-distance pair is higher than 30%.

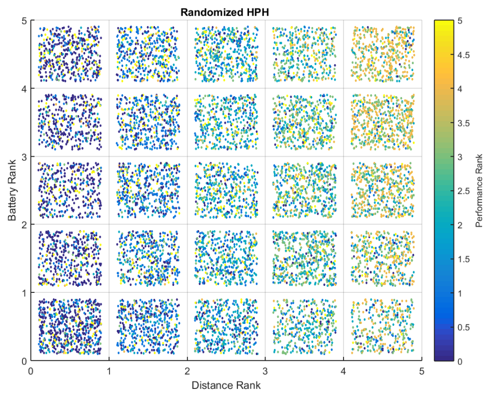
This high rank blurriness implies that, by using only the current factors (distance and battery) the performance of the rescue mission cannot be predicted. It is possible, however, to classify the Marsu-bots in such a way that the selected rescuer robot is able to complete the rescue mission without additional assistance.

Figure 3.4 shows the ranked data. As there are a total of 18 different strategies and 4 different scenarios, leading to a total of 72 plots, only a small subset of them will be shown in this section. All plots can be seen in Appendix C, section C.1.

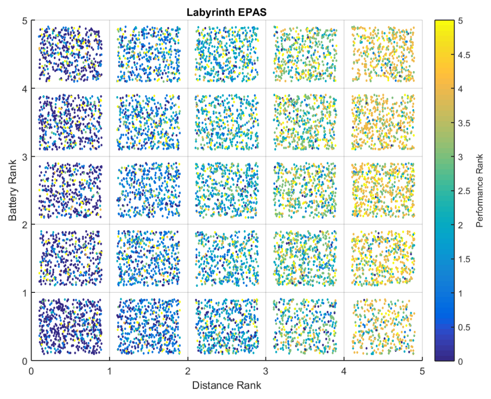
As expected, a relatively high variance may be easily noticed in each “cell”. It can be seen, however, than the mode of each cell (the most common colour) follows



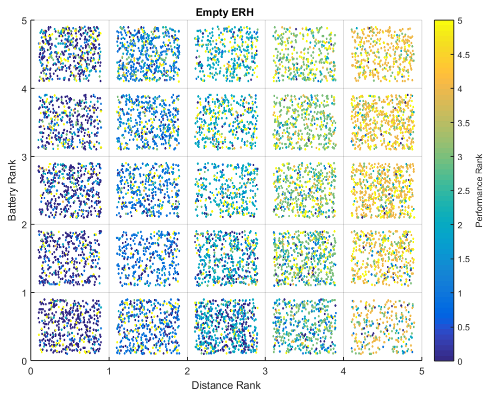
(a) Office HOAS.



(b) Randomized HPH.



(c) Labyrinth EPAS.



(d) Empty ERH.

Figure 3.4: Performance rank by Distance-Battery pair.

a quadric distribution. That leads to the hypothesis that the cost function can be approximated by a function of the form:

$$c_i = K_{d^2} D(r_v, r_i)^2 + K_{b^2} B(r_i)^2 + K_{db} D(r_v, r_i) B(r_i) + K_d D(r_v, r_i) + K_b B(r_i) \tag{3.2}$$

, where c_i is the cost of robot i, $D(r_v, r_i)$ is the distance between the victim and robot i, $B(r_i)$ is the percentage of depleted battery on robot i and K_{d^2}, K_{b^2}, K_{db}, K_d and K_b are the constant weights applied.

3.3 Cost Function Extraction

This section describes the specific implementation of the HNCA used to extract the function present in the computed data and the results obtained with this method. The aim of this algorithm is to obtain a function that maximizes the effectiveness of the selection made, that is, that results in the selection of the best possible robot or, at least, a robot able to perform the rescue without additional help.

When the algorithm is started, a population of individuals is created with random values assigned to their chromosomes. Then, each individual is tested and its fitness computed. After that, niching is applied to the population, in order to maintain a high genetic diversity. Once niching has been applied, a subset of the population is selected to influence the cultural knowledge of the problem. Finally, a new population is generated by using an Evolutionary Programming approach.

3.3.1 Individuals

Each individual is composed of five chromosomes:

$$\chi_i = \{K_{d^2}, K_{b^2}, K_{db}, K_d, K_b \quad | \quad K_{d^2}, K_{b^2}, K_{db}, K_d, K_b \in \mathbb{R}\} \tag{3.3}$$

where K_{d^2}, K_{b^2}, K_{db}, K_d and K_b correspond to those parameters in equation 3.2. Each chromosome is a continuous, unbounded, real number.

3.3.2 Fitness test

In order to find the fitness of a given solution, the cost function defined by its chromosomes is tested. To do that, the dataset generated during the pre-analysis of the problem is used.

For each set of Marsu-bots, the cost of each Marsu-bot (according to the cost function defined by the solution being tested) is computed and the Marsu-bot with a smaller cost is selected.

Finally, the four coefficients used in the fitness test are computed: robustness, correctness, complexity and size. Then, solutions are ranked by descending robustness, with ties broken by descending correctness, with further ties broken by ascending complexity and further ties broken by ascending size.

Robustness

Robustness is measured as the percentage of tests in which a robot able to successfully perform the rescue mission was selected by the cost function. This coefficient must be maximized. As this value is continuous, it is discretized by rounding it to closest multiple of 10. The reason behind this discretization is the fact that, when solutions are ranked, if the continuous value is used, there will be few collisions, and the other coefficients won't be used. Thus, a small amount of robustness is sacrificed by discretizing it, so correctness, complexity and size may have a bigger impact in the rank of the solution.

Correctness

Correctness is measured as the percentage of of tests in which the selected robot was the best available robot. This coefficient must be maximized. As this value is continuous, it is discretized by rounding it to closest multiple of 10, due to the same reasons explained above.

Complexity

Complexity refers to the quantity of error that may be introduced by the formula. Using squared values (such as squared distance) introduce greater error than using their linear counterparts. In addition integer values are preferred for the different constants as those are easier to represent. This coefficient must be minimized.

Size

Size is computed as the sum of the absolute values of all chromosomes. This coefficient must be minimized.

3.3.3 Niching

The niche clearing operation consists in clustering the individuals and then selecting only the fittest ones of each cluster. To cluster the solutions, a similarity function is applied to each chromosome, that is, for each chromosome, if the values of the two different solutions are close a value of 1 is returned (see equation 3.4).

$$S_{K_{d^2}}(\chi_i, \chi_j) = \begin{cases} 1 & \text{if } |K^i_{d^2} - K^j_{d^2}| \leq \frac{s(N_{K_{d^2}})}{D_c} \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

, where $S_{K_{d^2}}(\chi_i, \chi_j)$ is the similarity value for chromosome K_{d^2} between solutions χ_i and χ_j, $K^n_{d^2}$ is the value of chromosome K_{d^2} of solution χ_n, $s(N_{K_{d^2}})$ is the size of the interval defined by the normative knowledge for chromosome K_{d^2} (see section 3.3.4) and D_c is a constant used for all chromosome similarity functions.

If all five chromosomes result in a similarity of 1, then the solutions are clustered together and only the fittest ones are allowed to survive. The amount of solutions allowed to coexist in the same cluster is set dynamically.

In the beginning of the HCNA, the amount of solutions allowed to survive is set to one. This ensures that the resulting population after niching is sparsely spread, ensuring an exploratory behaviour. Every time the niching process results in more than half the population being removed, an additional solution is allowed to survive on each cluster. For example, after the first time more than half the population is removed, two solutions will be allowed to survive in the same cluster. This ensures that, as the algorithm progresses, more solutions are allow to coexist over a small space in the solution domain, which ensures an exploitation behaviour.

If, at any point, the niching operation removes less than a 10% of the population the amount of solutions allowed to survive is decreased by one.

3.3.4 Knowledge transfer

Knowledge transfer is applied in two steps: first, a subset of the individuals (that survived the niching phase) is selected, and then, knowledge is extracted from them. For this implementation of the algorithm, a simple average selection is used, that is, individuals with robustness and completeness better than average are selected to shape the belief space. Those individuals form the teaching set.

From the five types of knowledge defined by Reynolds [23] only three are used in this particular implementation: situational, normative and historical knowledge.

Situational Knowledge

Situational knowledge refers to the knowledge of the state of the algorithm, that is, the best solution found so far, without memory (only the absolute best is stored). To update this knowledge, the best solution from the teaching set is compared to the current knowledge, if the new solution is better it is stored and the old one discarded.

Normative Knowledge

Normative knowledge refers to a “set of promising variable ranges that provide standards for individual behavior and guidelines within which individual adjustments can be made” [27].

In this specific implementation normative knowledge is used to store the promising ranges of the different chromosomes. That is done by finding the boundary values of each chromosome and defining the “promising range” as the interval enclosed by this boundary values.

Normative knowledge is then used when computing the mutations for those chromosomes.

Historical Knowledge

Finally, historical knowledge refers to the knowledge of past events in the evolutionary process. In this particular implementation the fittest individual of each generation is stored.

3.3.5 Reproduction

This implementation uses and EP approach to create new solutions. In EP there is no usual reproduction, instead the parents are mutated and the new mutated values are compared to those of the parents, the fittest values are added to the population. Due to the complexity of computing the new fitness values, in this implementation both the unmodified parent and its mutated child are added to the new population.

The parents are selected using tournament selection. For this implementation, the number of solutions that participate in a tournament is dynamically selected, having, at least, a 1% of the population participating in a tournament.

Tournament Selection

Tournament selection consists in selecting n random individuals of the population, with all individuals having the same probability of being chosen and allowing for repetition. Then, the best individual of this set is chosen as the parent.

As shown in [21] the probability of an individual being selected is given by equation 3.5.

$$p_i = \frac{1}{\mu^q}((\mu - i + 1)^q - (\mu - i)^q) \tag{3.5}$$

, where q is the size of the tournament, μ is the size of the population and p_i is the probability of selecting the individual i.

Tournament selection has the advantage of being simple to implement, on the other hand is more biased towards fitter solutions than other options, which may lead to premature selection of a local minima instead of the absolute minima. This undesired result is, however, controlled due to niching technique applied earlier, as the population is more sparsely spread over the solution domain.

Mutation

Normative knowledge is used to direct chromosome mutation. The mutated chromosome value is computed using equation 3.6.

$$k_{i+1} = k_i + \mathcal{N}(0,1)s(N_K) \tag{3.6}$$

,where k_{i+1} is the mutated value, k_i is the original value, $\mathcal{N}(0,1)$ its a value taken from a normal distribution with mean 0 and variance 1 and $s(N_K)$ is the size of the interval defined by the normative knowledge associated to that chromosome.

The use of normative knowledge provides an exploration-explotation switch. In the firsts generations the interval defined by the normative knowledge is quite big, which leads to bigger mutations, which separate the different individuals through the solution domain (exploration). When the algorithm has ran for some time, however, this interval has been shrinking down towards the best solutions found, which leads to smaller mutations, concentrating the individuals around some specific values in the solution domain (explotation).

3.3.6 Termination conditions

A certain termination condition needs to be created to detect when the algorithm has converged on the absolute maximum. In this implementation of the algorithm, the execution is terminated when the best solution found has a robustness higher than 95% and a correctness higher than 90%.

In addition the total time taken for the algorithm is limited to 4 hours.

3.3.7 Tests

In order to ensure the correct performance of the algorithm two sets of data were created: one based on a linear function (K_b,K_d = 1, all other chromosomes 0), and a quadric one (all chromosomes 1).

Then the algorithm was used to extract the data. The correct function was found in both cases.

3.3.8 Results

Table 3.2 shows the Robustness and Correctness of the functions found.

Table 3.2: Function qualifiers by scenario and strategy.

	Office		Empty		Labyrinth		Randomized	
Strategy	R(%)	C(%)	R(%)	C(%)	R(%)	C(%)	R(%)	C(%)
HOAS	95.3	56.1	89.7	65.0	95.0	50.6	87.6	62.8
HOAM	98.5	79.4	96.2	82.4	97.4	85.0	96.6	85.0
HOH	95.0	58.6	87.0	62.2	95.0	56.8	86.3	61.9
HPAS	91.8	59.8	86.7	63.4	88.7	63.0	86.2	63.3
HPAM	98.3	75.0	97.5	85.2	96.6	80.1	96.4	82.6
HPH	95.2	55.0	95.0	50.2	95.0	45.5	95.0	47.8
HRAS	95.3	67.0	87.7	61.2	95.0	59.4	87.2	60.6
HRAM	98.3	88.1	96.7	86.6	97.0	86.7	96.6	85.9
HRH	95.6	66.6	88.9	65.3	95.1	57.3	87.0	58.9
EOAS	95.4	62.0	90.7	66.8	95.3	65.5	90.3	65.8
EOAM	98.1	80.5	90.6	77.2	94.3	80.4	90.8	75.7
EOH	96.1	65.1	89.9	65.0	92.9	67.1	90.4	65.3
EPAS	95.9	63.8	90.5	65.6	93.0	65.8	89.3	65.8
EPAM	97.7	75.1	90.2	76.5	93.9	80.7	89.9	77.1
EPH	96.2	55.5	89.4	59.7	92.3	60.8	88.5	58.1
ERAS	97.6	75.3	90.1	65.7	93.4	67.4	91.1	65.2
ERAM	98.1	88.9	90.6	79.2	94.8	85.1	90.1	78.7
ERH	97.6	75.1	90.1	66.2	95.1	66.9	91.3	66.4

As can be seen for all combinations of scenario and strategy it was possible to find a function which, at least 86% of the time, results in the selection of a rescuer able to perform the rescue operation successfully (Robustness higher or equal to 86%).

On the other hand Correctness values are quite variable, ranging from 50% in the worst case to 88% in the best case, which means that in the worst cases the best possible robot will be selected only half the times the algorithm is ran.

From this results it can be seen that, though the system is quite robust, optimality can not be assured.

Due to space constraints the specific functions found for each possible combination are shown in Appendix C, section C.2.

3.4 Strategy selection

The results found so far give no information about which combinations of strategies are better. In order to find the best combination of strategies a tournament system was implemented.

3.4.1 Tournament algorithm

Initially all strategies are considered. Using the MarSim simulation, a rescue event is generated. Then, two different sets of strategies are tested, and its performance compared according to time taken and amount of battery left in the rescuer robot. This test is repeated ten times for each pair of strategies. If a strategy obtains better results for seven or more tests, it is considered a winner in this round.

Once each pair of strategies have been tested, the results of the different matches are compared. If a strategy has not won any match, it is removed from the participant pool. If no such strategy is found, the different matches are separated into "team matches", for example, all matches in which an H strategy was used are compared with those in which an E strategy was used. If one of these "teams" won more than 20% of those matches, all its strategies are considered winners and the rest are removed.

Once the round results have been computed, if there is more than one strategy in the participant pool and at least one strategy has been removed from the participant pool in this round, a new round is started, testing all combinations of strategies still present in the participant pool.

3.4.2 Results

Table 3.3 shows the results of the tournament experiment.

Table 3.3: Tournament results per scenario.

Scenario	Winner	Second
Empty	EOAS, EOH, ERAS, ERH	EPAS, EPH
Randomized	EOH, ERAS, ERH	EOAS
Office	EOAS, EOH, ERAS, ERH	EPAS, EPH
Labyrinth	EOAS, EOH, ERAS, ERH	EPAS, EPH

As can be seen, for all four scenarios all winners and all second places use the E strategy, that is, transferring just enough energy for the Marsu-bot to reach the Mother-bot (plus some margin). In addition it can be seen that none of these strategies use the AM strategy, that is, always going to the Mother-bot and recharging before attempting the rescue.

It can also be seen that the H and AS strategies are, in most scenarios, equivalent when it comes to performance. The main reason behind that is the fact that the Hybrid strategy behaves like the Always Save strategy if the Marsu-bot has enough energy to reach the victim, which will usually be the case.

Finally, it can be seen that both Realist and Optimist distances have a good performance in all scenarios.

3.5 Analysis

Tests show that strategies using Optimist and Realist distance functions and transferring enough energy for the Marsu-bot to reach the Mother-bot have better performances (see in table 3.3). It has also been seen that strategies in which the Marsu-bot fully recharges its batteries before attempting the rescue mission have lower performances.

Table 3.4 shows the Robustness and Correctness values for the winner strategies shown in table 3.3.

Table 3.4: Function qualifiers by scenario and strategy for selected strategies.

	Office		Empty		Labyrinth		Randomized	
Strategy	$R(\%)$	$C(\%)$	$R(\%)$	$C(\%)$	$R(\%)$	$C(\%)$	$R(\%)$	$C(\%)$
EOAS	95.4	62.0	90.7	66.8	95.3	65.5	90.3	65.8
EOH	96.1	65.1	89.9	65.0	92.9	67.1	90.4	65.3
ERAS	97.6	75.3	90.1	65.7	93.4	67.4	91.1	65.2
ERH	97.6	75.1	90.1	66.2	95.1	66.9	91.3	66.4

It can be seen that for the Empty, Labyrinth and Randomized scenarios there is almost no difference on Robustness or Correctness (less than 1%) between different distance functions. For the Office scenario, however, there is a more significant difference between using an Optimist and a Realist distance.

This can be tied to the probability distribution of the sinuosity of each scenario. In figure 3.2, it can be seen that the probability distribution of the sinuosity has a longer tail for the Office scenario than for any other scenario, which means that there is a higher likelihood of any two arbitrary points having a bigger sinuosity.

In order to find a suitable cut-point that allows the differentiation of these two cases, the minimum value of sinuosity that has accumulated probability bigger than 0.9 for both scenarios was selected, that is, a value of sinuosity S such as $P(s \leqslant S) \geqslant 0.9$.

Table 3.5: S values such as $P(s \leqslant S) \geqslant 0.9$.

Scenario	S
Office	2.2
Labyrinth	1.8

Table 3.6 shows the probability of the sinuosity being bigger than 1.8 for each scenario, that is $P(s \geqslant 1.8)$.

This leads to the conclusion that a cut-point can be established in order to group the different scenarios. Two groups of scenarios are then defined using this cut point. On one hand, “simple” scenarios, that is, scenarios in which the probability of obtaining a sinuosity bigger or equal to 1.8 is smaller than 0.142, in this study, the Empty, Randomized and Labyrinth scenarios. On the other hand, “complex”

Table 3.6: $P(s \geqslant 1.8)$ by scenario.

Scenario	$P(s \geqslant 1.8)$
Office	0.1781
Labyrinth	0.1058
Randomized	0.0
Empty	0.0

scenarios, that is, scenarios in which the probability of obtaining a sinuosity bigger or equal to 1.8 is bigger than 0.142, in this study, the Office scenario.

3.5.1 Simple scenarios

In simple scenarios, the difference between Optimist and Realist distance is, usually, quite small. As the computation cost of a Realist distance is always bigger than the computational cost of an Optimist distance, due to the fact that a Realist distance is based on a path-planning algorithm, it is recommended to use Optimist distances, specifically, the euclidean distance between the victim and the rescuer.

Tables 3.7, 3.8, 3.9 show the functions obtained for the EOAS and the EOH strategies for the Randomized, Labyrinth and Empty scenarios respectively.

Table 3.7: Function parameters by strategy in an Randomized scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
EOAS	0.0	0.0	0.0	10.0	0.0
EOH	0.0	0.0	0.0	10.0	10.0

Table 3.8: Function parameters by strategy in an Labyrinth scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
EOAS	0.0	0.0	10.0	40.0	0.0
EOH	0.0	0.0	0.0	10.0	0.0

Table 3.9: Function parameters by strategy in an Empty scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
EOAS	0.0	0.0	0.0	10.0	10.0
EOH	0.0	0.0	0.0	10.0	10.0

As can be seen, there are three different cost functions. As the results of the cost function are compared between different Marsu-bots, it is possible to divide the functions by the maximum common divider without modifying its results (selection-wise). Those are equations 3.7, 3.8 and 3.9.

$$c_i^{F_1} = D(r_v, r_i) \tag{3.7}$$

$$c_i^{F_2} = D(r_v, r_i) + B(r_i) \tag{3.8}$$

$$c_i^{F_3} = D(r_v, r_i)B(r_i) + 4D(r_v, r_i) \tag{3.9}$$

In order to try to find a single function that yields good results for each scenario, the three functions were tested in the different combinations of scenario and strategy. As the three functions have different complexities, it is possible that a function with higher complexity may bring a small increase in performance (smaller than 10%).

Then, the decrease in Robustness and Correctness was computed, subtracting the new values from the original ones. Table 3.10 shows the results of this calculation.

Table 3.10: Robustness and Correctness decrease for different formulas.

	F_1		F_2		F_3	
Test Set	$\Delta R(\%)$	$\Delta C(\%)$	$\Delta R(\%)$	$\Delta C(\%)$	$\Delta R(\%)$	$\Delta C(\%)$
Empty EOH	0.9	1.8	0	0	−0.2	2.8
Empty EOAS	0.8	2.1	0	0	−0.2	3.3
Labyrinth EOH	0	0	−0.8	−0.1	−1.3	3
Labyrinth EOAS	1.3	−3.3	0.4	−2.2	0	0
Randomized EOH	0.9	2.9	0	0	−0.4	2.6
Randomized EOAS	0	0	−0.5	−0.2	−1.1	2.9
Average	0.65	0.583	−0.15	−0.416	−0.533	2.433

It can be seen that using function 3.8 brings an average improvement in both Robustness and Correctness. Furthermore, it can be seen that only the Correctness of using the EOAS strategy in the Labyrinth scenario is reduced, and only by a 0.4%.

Thus, for Simple scenarios, the cost function used should be:

$$c_i^{simple} = D(r_v, r_i) + B(r_i) \tag{3.10}$$

3.5.2 Complex scenarios

In complex scenarios, the difference between using a Realist and an Optimist distance function is highly relevant, thus, a Realist function should be used.

Table 3.11: Function parameters by strategy in an Office scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
ERAS	0.0	0.0	0.0	10.0	10.0
ERH	0.0	0.0	0.0	10.0	10.0

As can be seen, both for the Hybrid and Always Save strategies, the function has the same parameters. As the results of the cost function are compared between different Marsu-bots, it is possible to divide the function by the maximum common divider without modifying its results (selection-wise).

Thus, for Complex scenarios the cost function used should be:

$$c_i^{complex} = D(r_v, r_i) + B(r_i) \tag{3.11}$$

3.6 Conclusion

At the beginning of the chapter, the aim was to find a cost function which used the distance from victim to robot and the amount of battery left in the robot to predict the performance of a rescue mission performed by that robot. This function had to have a minimum on the most suitable robot to perform the rescue, in terms of time and battery left at the end of the rescue.

After generating several thousands of simulations using different scenarios and conditions, it was seen that it was impossible to find a unique function that conformed to those parameters, but it was possible to find a function to classify the Marsu-bots in such a way that the selected rescuer was able to complete the rescue mission without additional assistance. In addition, it was possible to build the function in such a way that, more than 50% of the selections resulted in the selected rescuer being also the best rescuer.

Once those functions were found, different possible strategies to complete the rescue mission were considered. Transferring half the available battery or just enough battery to reach the Mother-bot (plus some margin); using an Optimist (euclidean), Pessimist (euclidean squared) or a Realist (path) distance; and always recharging the rescuer battery before a rescue mission, always going directly to the victim or going to the victim unless there wasn't enough battery to reach it.

By comparing the performance of the different strategies, it was possible to see that transferring just enough battery to reach the Mother-bot (plus some margin) had better results independently of the scenario in which the rescue was performed. It could also be seen that there wasn't a big difference in performance between always going directly to the victim (Always Save) or going to the victim unless there wasn't enough battery to reach it (Hybrid).

On the other hand, it was possible to see that there was a difference in the impact on the performance of using an Optimist or a Realist distance that was dependant on the scenario. By analysing the properties of the scenario, more specifically, the probability distribution of the sinuosity between two arbitrary points, it was seen that scenarios could be categorized in two groups, Simple and Complex.

Finally, the functions for the different scenarios (in each group) were considered, and a function that performed correctly in both cases was found. This function is not scenario-dependant.

That lead to the conclusion that a taxonomy can be built to help decide which distance type and cost function to use according to the characteristics of the swarm and the scenario in which this swarm is. This taxonomy is shown in figure 3.5.

The taxonomy shown in figure 3.5 is used in the following way: each circle represents a question about the swarm or the scenario. Each square represents a conclusion reached. For example, in the current case the swarm is homogeneous

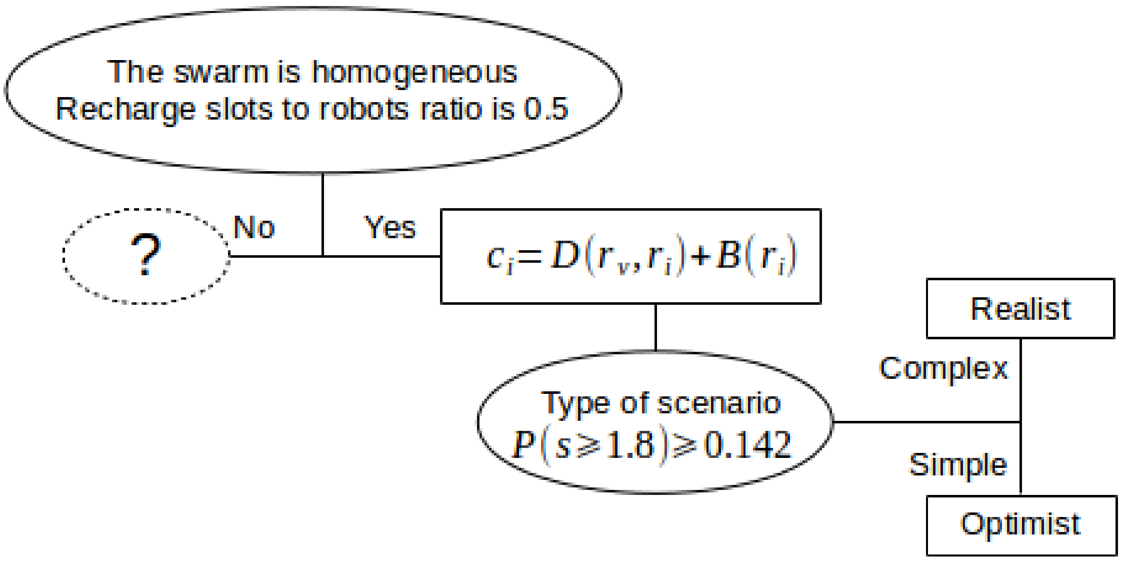

The swarm is homogeneous
Recharge slots to robots ratio is 0.5
?
No
Yes
$c_i = D(r_v, r_i) + B(r_i)$
Type of scenario
$P(s \geqslant 1.8) \geqslant 0.142$
Complex
Simple
Realist
Optimist

Figure 3.5: Rescue taxonomy.

and the ratio of recharge slots to robots is 0.5, so the cost function that should be used is $c_i = D(r_v, r_i) + B(r_i)$. If the swarm is in the Office scenario (Complex, $P(s \geqslant 1.8) > 0.142$) then a Realist distance function should be used to compute $D(r_v, r_i)$.

CHAPTER 4

Conclusions

The goal of this thesis was to design a protocol to allow the Marsu-bots to exchange battery in case of need. Figure 4.1 shows the complete protocol for a rescue operation.

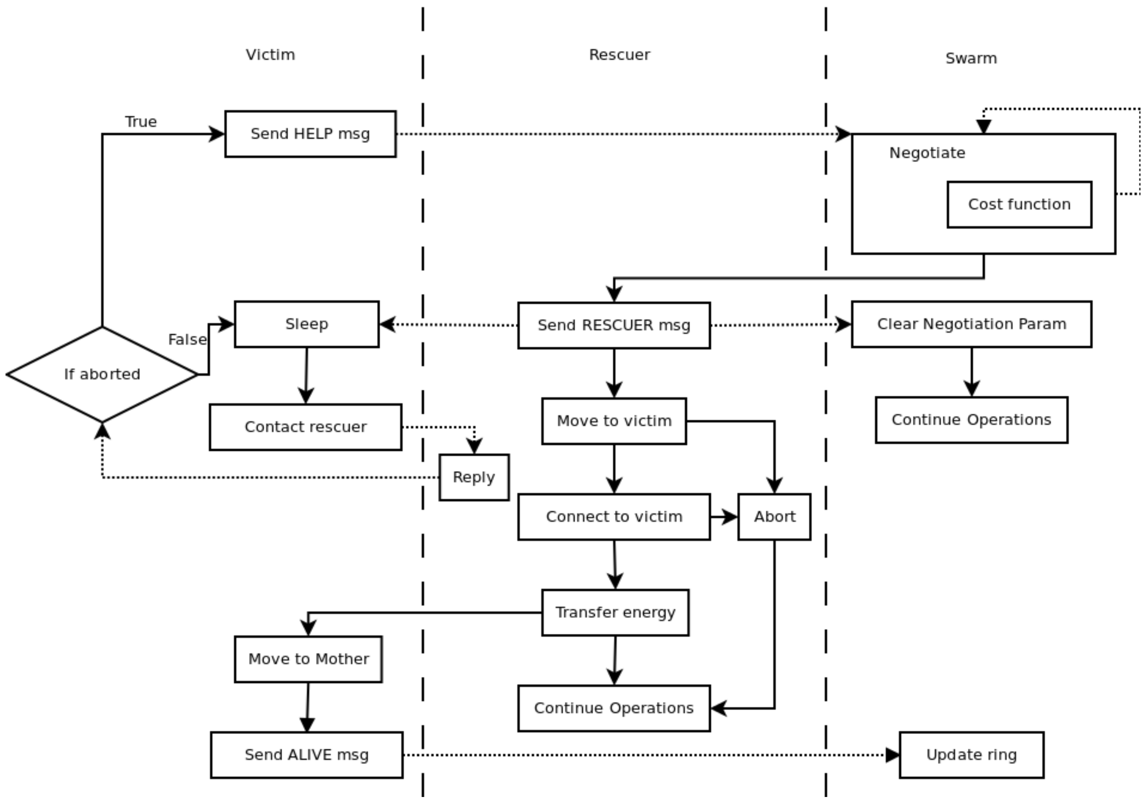


Figure 4.1: Flowchart of the rescue protocol. Dotted arrows represent messages.

When the battery of one of the Marsu-bots falls below the critical threshold that Marsu-bot becomes a victim. At this point, it will send a HELP message through the network with its current coordinates (position and orientation) and Marsu-bot id. Then, the negotiation algorithm will be engaged by the swarm to autonomously select a rescuer.

Once the rescuer Marsu-bot has been selected the rest of the swarm will resume its normal operation. The rescuer, on the other hand, will move towards the position of the victim, connect to it and transfer energy to it.Once this is done, it will resume its normal operations.

Once the battery transfer has been completed the victim will move to the Mother-bot and fully recharge its batteries.

This study focused on finding the answers to the following questions:

- Which negotiation algorithm should be used?
- How can the performance of each robot be predicted?

4.1 Negotiation algorithm

The first key component that needed to be researched was the negotiation algorithm to use. That algorithm needs to be decentralized, finite and result in the selection of a single Marsu-bot. It was hypothesized that a leader election algorithm could be used for this purpose.

Different leader election algorithms where tested in a simulated network in order to find an algorithm that was fast, used few messages and specifically few simultaneous messages. The algorithms were tested for scalability and computation flexibility, that is, the effect that longer computational times for the cost calculation has on the algorithm performance.

It was found that the "leader election algorithm for complete networks without sense of direction" [15], written by Villadangos *et al.*, started at two different nodes across the ring, provides the best results in terms of compromise between time and amount of simultaneous messages on the network.

4.2 Cost function

The negotiation algorithm requires a cost function to minimize. This cost function should predict the performance of a a certain robot executing the rescue mission based on its distance to the victim and the amount of available battery. In addition, different strategies were considered as those may have an effect on the cost function.

Three sets of different strategies were considered: transferring half the available energy or just enough energy to reach the Mother-bot (transfer type); using an Optimist (euclidean), Pessimist (euclidean squared) or Realist (path) distance function to compute the distance between victim and robot and engaging the rescue mission directly, recharging to full battery before attempting the rescue mission or a hybrid method (engaging the rescue mission directly unless the robot doesn't have enough energy to reach the victim).

As it was unknown which shape such a cost function may take, or even if it existed, a series of simulations were performed to obtain a relevant set of data that could point towards the right answers. For each possible combination of strategies several tests were performed in four different scenarios.

It was seen that, though it is nearly impossible to fully predict the performance of a rescue mission carried out by a specific robot using only its distance to the victim and the amount of battery this robot has already spent, it was possible to find a cost function which has its minimum at either the best robot, or a robot that is able to perform the rescue operation without additional help.

By using the generated simulations as a training set for a Cultural algorithm, the different functions were extracted from the data. This resulted in a total of 72 different functions, one for each combination of strategies and scenario.

Finally, the different strategies were compared to each other in order to select the best strategy possible. It was seen that transferring just enough energy to reach the Mother-bot (plus a small margin) provided the best general performance. It was also seen that fully recharging the batteries of the rescuer prior to the rescue attempt resulted in lower performances.

Furthermore, it was seen that the cost function used is not scenario-dependant, but the function used to compute the distance is. A taxonomy was devised to help select the distance function that suits the scenario better. This taxonomy is shown in figure 4.2

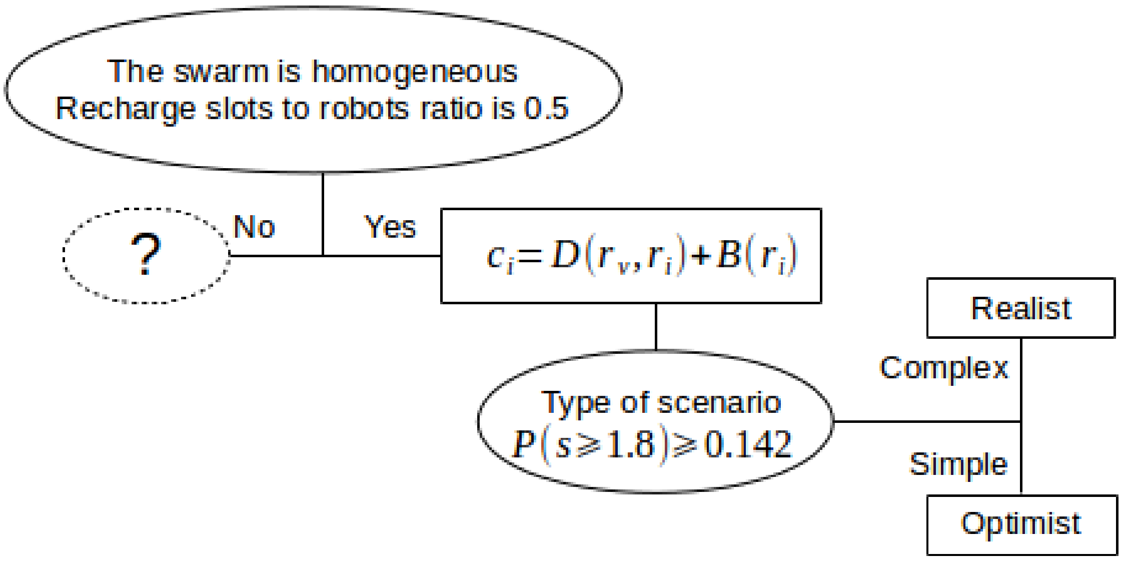

The swarm is homogeneous
Recharge slots to robots ratio is 0.5
?
No
Yes
$c_i = D(r_v, r_i) + B(r_i)$
Type of scenario
$P(s \geqslant 1.8) \geqslant 0.142$
Complex
Simple
Realist
Optimist

Figure 4.2: Rescue taxonomy.

The cut-point between Simple and Complex scenarios is determined by the probability distribution of the sinuosity in that scenario. Complex scenarios are those in which this probability distribution has a long tail, that is, the probability that the sinuosity between two arbitrary points is high is also high.

4.3 Further work

A rescue protocol has been designed, yet much work remains to be done. On one hand, practical tests need to be conducted with the physical robots. On the other hand, further study about the cost function needs to be done.

4.3.1 Practical tests

The protocol needs to be implemented and tested on the actual Marsu-bots, to ensure that the performance of the protocol is as desired. Special care needs to be given to the cascade prevention systems.

A system could be designed to identify the the different parameters required (for example, the type of distance function) autonomously based on the current knowledge of the map.

4.3.2 Theoretical research

The taxonomy built in this study covers a very small subset of possibilities, and further studies should be made:

The current taxonomy gives no information on the effects of different ratios between number of recharge slots available and number of robots on the swarm.

Scalability of the current protocol should be tested, as the effects of increasing the number of robots (though keeping the recharge slots to robots ratio) is unknown.

The effects of using a non-homogeneous swarm are unknown, that is, which changes would appear on the cost function if the robots have different battery capacities or different energy consumption rates.

More scenarios should be tested to optimize the cut-point between Simple and Complex scenarios.

Bibliography

[1] S. Ramalingam, S. Siddarthan, R. Srinivasan, and V. Prabakaran, "Optimal Battery Charging and Solar Tracking System for Robots," *Automation and Autonomous System*, vol. 7, no. 1, pp. 7–11, 2015.

[2] M. S. El-Genk and H. H. Saber, "Performance analysis of cascaded thermoelectric converters for advanced radioisotope power systems," *Energy Conversion and Management*, vol. 46, no. 7-8, pp. 1083–1105, 2005.

[3] I. Ieropoulos, J. Greenman, C. Melhuish, and I. Horsfield, "EcoBot-III: a robot with guts," pp. 733–740, 2010. [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=12433

[4] M. Rubenstein, C. Ahler, and R. Nagpal, "Kilobot: A low cost scalable robot system for collective behaviors," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 3293–3298, 2012.

[5] C. Skjetne, P. C. Haddow, A. Rye, H. v. Schei, and J.-M. Montanier, "The ChIRP Robot: A Versatile Swarm Robot Platform," in *Robot Intelligence Technology and Applications*. Springer, 2014, vol. 2, pp. 71–82.

[6] A. Spröwitz, R. Moeckel, M. Vespignani, S. Bonardi, and A. J. Ijspeert, "Roombots: A hardware perspective on 3D self-reconfiguration and locomotion with a homogeneous modular robot," *Robotics and Autonomous Systems*, vol. 62, pp. 1016–1033, 2014.

[7] H. Schioler and T. D. Ngo, "Trophallaxis in robotic swarms - beyond energy autonomy," *2008 10th International Conference on Control, Automation, Robotics and Vision, ICARCV 2008*, pp. 1526–1533, 2008.

[8] R. O'Grady, C. Pinciroli, R. Groß, A. L. Christensen, F. Mondada, M. Bonani, and M. Dorigo, "Swarm-bots to the rescue," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5777 LNAI, 2011, pp. 165–172.

[9] M. Traub, G. a. Kaminka, and N. Agmon, "Who goes there ?: selecting a robot to reach a goal using social regret," *The 10th International Conference on Autonomous Agents and Multiagent Systems*, pp. 91–98, 2011.

[10] H. A. V. Marek Matusiak, Janne Paanajärvi, Pekka Appelqvist, Mikko Elomaa and Mika Ylikorpi T., “A Novel Marsupial Robot Society: Towards Long-Term Autonomy,” in *Proceedings of the 9th International Symposium on Distributed Autonomous Robotic Systems (DARS 2008)*, 2008.

[11] R. R. Murphy, M. Ausmus, M. Bugajska, T. Ellis, T. Johnson, N. Kelley, J. Kiefer, and L. Pollock, “Marsupial-like Mobile Robot Societies Computer Science and Engineering,” in *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS 1999)*, Seattle, 1999.

[12] D. L. Martinez and A. Halme, “MarSim , a simulation of the MarsuBots fleet using NetLogo,” in *The 12th International Symposium on Distributed Autonomous Robotic Systems*, no. DARS 2014, 2014, pp. 262–270.

[13] U. Wilensky, “NetLogo,” 1999. [Online]. Available: http://ccl.northwestern.edu/netlogo/

[14] N. Santoro, *Design and Analysis of Distributed Algorithms*. John Wiley & Sons, 2006.

[15] J. Villadangos, a. Córdoba, F. Fariña, and M. Prieto, “Efficient leader election in complete networks,” *Proceedings - 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing 2005, PDP 2005*, vol. 2005, pp. 136–143, 2005.

[16] D. S. Hirschberg and J. B. Sinclair, “Decentralized Extrema- Finding in Circular Configurations of Processors,” *Communications of the ACM*, vol. 23, no. 11, pp. 627–628, 1980.

[17] M. C. Loui, T. A. Matsushita, and D. B. West, “Election in a Complete Network with a Sense of Direction,” *Information Processing Letters*, vol. 22, no. April, pp. 185–187, 1986.

[18] H. Garcia-Molina, “Elections in a distributed computing system,” *IEEE Trans. Comp.*, vol. 31, pp. 48–59, 1982.

[19] J. Baca, B. Woosley, P. Dasgupta, A. Dutta, and C. Nelson, “Coordination of Modular Robots by means of Topology Discovery and Leader Election : Improvement of the Locomotion Case,” in *The 12th International Symposium on Distributed Autonomous Robotic Systems*, no. DARS 2014, 2014, pp. 271–282.

[20] A. P. Engelbrecht, *Computational intelligence: An introduction*. John Wiley & Sons, 2008.

[21] T. Bäck, *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[22] R. G. Reynolds, “An introduction to cultural algorithms,” *Proceedings of the Third Annual Conference on Evolutionary Programming*, pp. 131–139, 1994. [Online]. Available: http://www.danielcondurachi.ro/podpress_trac/feed/10/0/cult94.pdf

[23] R. G. Reynolds and B. Peng, “Cultural Algorithms: Computational Modeling of How Cultures Learn To Solve Problems: an Engineering Example,” *Cybernetics & Systems*, vol. 36, pp. 753–771, 2005. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01969720500306147

[24] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[25] M. Z. Ali, N. H. Awad, and R. G. Reynolds, “Hybrid niche Cultural Algorithm for numerical global optimization,” *2013 IEEE Congress on Evolutionary Computation, CEC 2013*, pp. 309–316, 2013.

[26] G. W. Greewood, G. B. Fogel, and M. Ciobanu, “Emphasizing extinction in evolutionary programming,” *Proceedings of the 1999 Congress on Evolutionary Computation, CEC 1999*, vol. 1, pp. 666–671, 1999.

[27] C.-J. Chung and R. G. Reynolds, “CAEP: an evolution-based tool for real-valued function optimization using cultural algorithms,” *International Journal on Artificial Intelligence Tools*, vol. 7, no. 3, pp. 239—-291, 1998.

APPENDIX A

NetSim Developer Manual

A.1 Introduction

NetSim is a multi-platform tool for distributed algorithm performance analysis built in Java. NetSim was originally designed to simulate leader election algorithms, but many different kinds of algorithms may be implemented as well.

At the core of the simulation, a directed graph is used to simulate a network topology, in which each node is a single processor (multitasking is not supported) that executes a certain algorithm. After the different experiments have been executed NetSim generates plots for different parameters of interest, such as number of messages sent, time taken to completion or number of simultaneous messages.

This manual describes the architecture of NetSim and how can it be extended to suit the needs of each specific user.

A.2 Software Architecture

NetSim is divided in 4 building blocks. In the following sections the relationships between classes will be explored. For more information on the attributes and methods of each class please consult the JavaDoc files.

A.2.1 NetSim class

The NetSim class is the host to the `main` method of NetSim. It also contains the methods for generating the experiments from .ae files and to run those experiments.

If NetSim has been added as a library in your project you will need to invoke the static methods `NetSim.runFile` or `NetSim.runFolder` to run either an experiment file or a folder containing experiments.

A.2.2 The Core

The core classes of NetSim are the classes used to represent the network and its interactions. Figure A.21 shows the UML diagram of those.

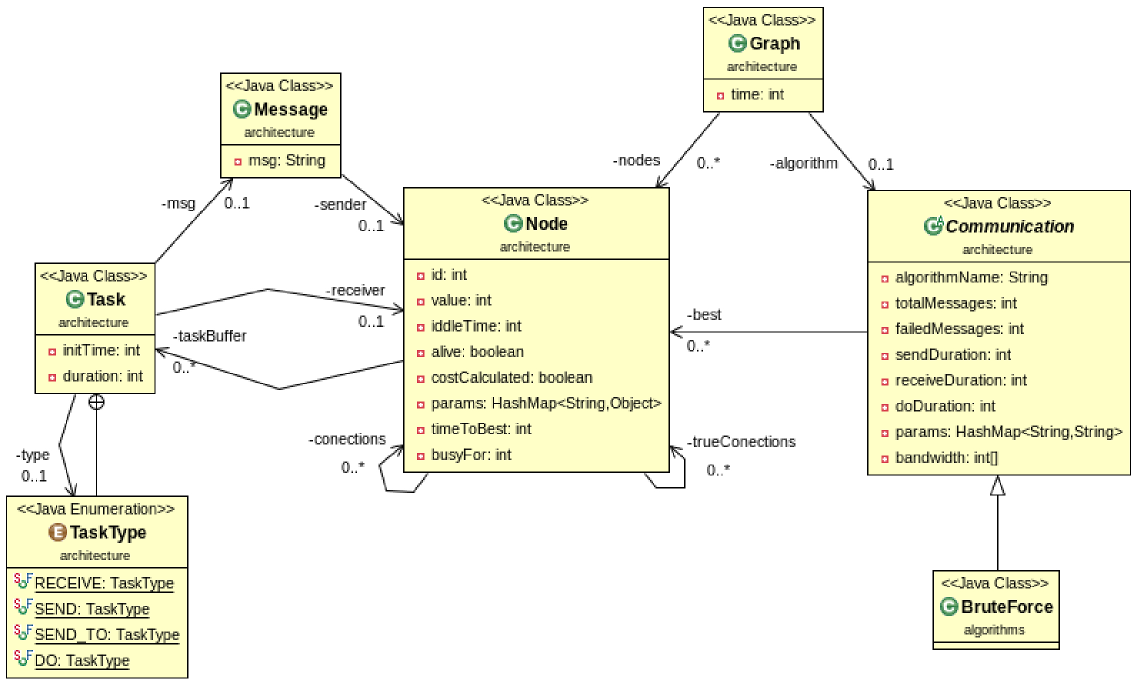


Figure A.21: UML Diagram of the core classes in NetSim

For any experiment a new `Graph` will be created. This graph contains a list of `Node` instances. Each node contains two lists of nodes, one with the nodes it “thinks” it can communicate to, and another one with the nodes it can actually communicate to (this double list was built to handle nodes disappearing from the network unexpectedly, but this feature has not been implemented yet). Each node contains also a list of `Task` instances with the tasks this node will execute at some point in the future. Each task has a `TaskType` and may contain one `Message`. This message contains a string with the specific message and the node who sent the message. Finally the graph object contains the `Communication` instance representing the algorithm to be tested (actually it will always be a class that extends the `Communication` class). The communication object contains a list of nodes containing the nodes that have been selected by the algorithm.

A.2.3 Analysis Results

Once an experiment has been run its results are stored to generate the plots at the end of the program execution. Figure A.22 shows the UML diagram corresponding to the result containers.

There are two kinds of containers. On one hand the `ResultSet` container stores the experiment results of non-timing experiments (those experiments in which the

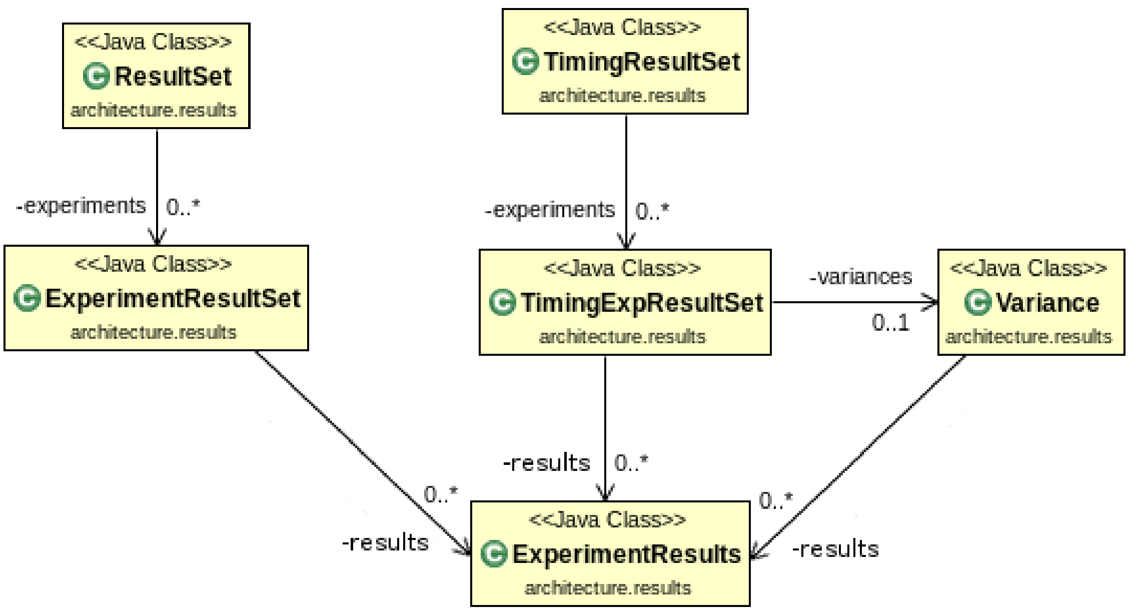


Figure A.22: UML Diagram of the result container classes in NetSim

send, receive and compute durations are constant through the experiment), while `TimingResultSet` contains the results of the timings experiments.

A `ResultSet` instance will contain a list of `ExperimentResultSet` instances, grouped by algorithm. Similarly, a `TimingResultSet` instance contains a list of `TimingExpResultSet` instances, grouped by algorithm as well.

An `ExperimentResultSet` instance contains, in turn, a list of `ExperimentResults` instances, in this case grouped by number of nodes in the experiment.

A `TimingExpResultSet` instance contains a list of `ExperimentResults`, in this case grouped by send plus receive time, and then grouped by computing time. It also contains a `Variance` object, which contains a list of `ExperimentResults` instances grouped by send, receive and compute times.

The `ExperimentResults` object contains all the related information to the experiment, such as time taken, selected nodes, number of messages, and so on.

A.2.4 Factories

There are two classes in NetSim used as factories. Both classes implement the singleton pattern (static classes).

The `PlotFactory` class contains the methods to generate the expected plots. The `PermutationFactory` contains the methods to generate permutations of numbers (more information can be found in the User Manual and in the JavaDoc).

A.3 Program flow

This section details the intended flow for a NetSim execution.

A.3.1 Main flow

Figure A.31 shows the sequence diagram for a NetSim execution.

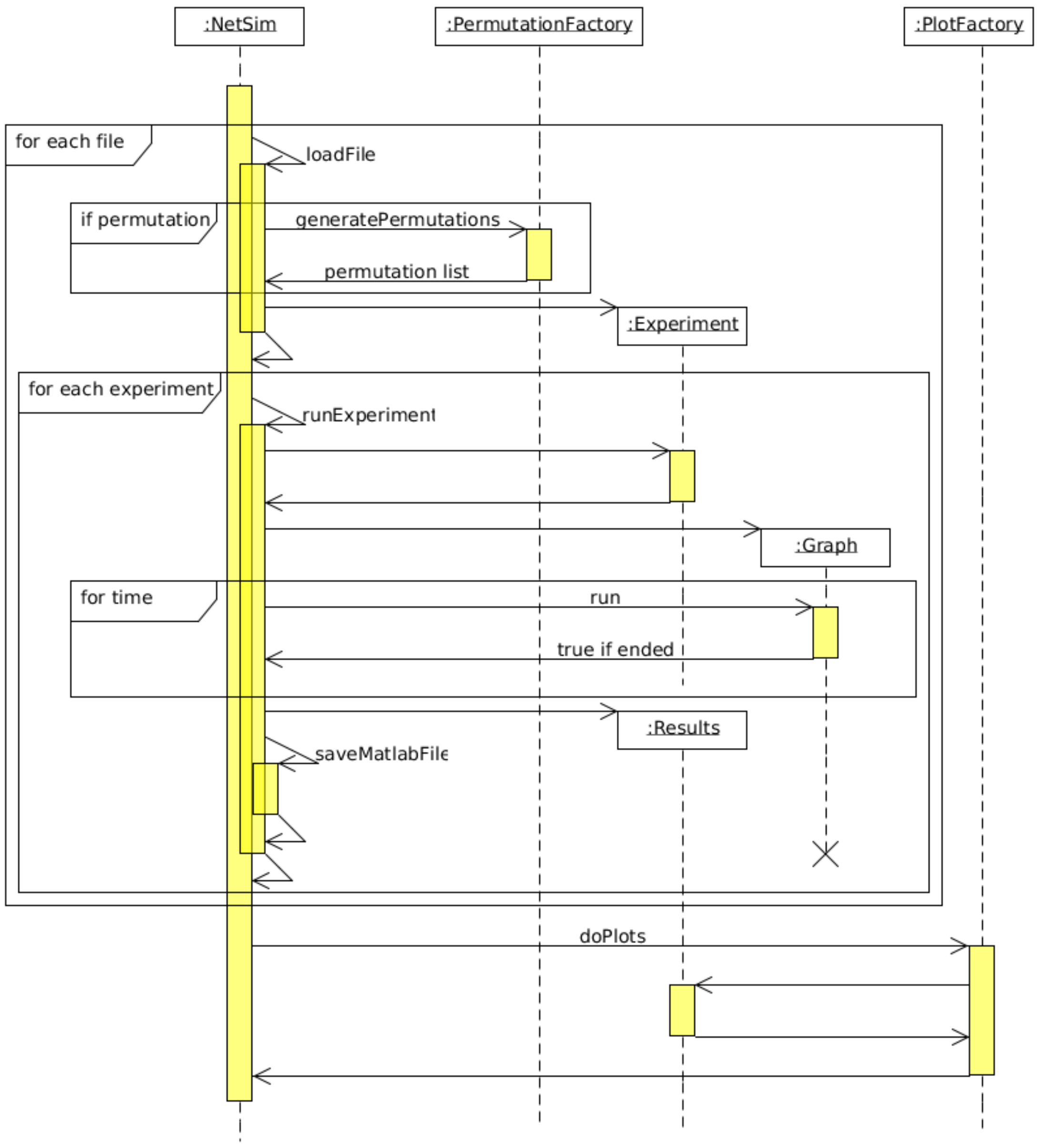


Figure A.31: Sequence diagram for the main flow

On initialization the main method will collect a list of `.ae` files to execute. Then, for each file, the file will be parsed generating a list of `Experiment` instances with the corresponding parameters. During the parsing of the file, if permutations are required, a call to the `PermutationFactory` will be issued.

Once the list of experiments has been created, for each experiment, the corresponding parameters will be set, and the `Graph` instance will be created (see section A.3.2 for more details on graph initialization).

Then, for each time step up to the defined maximum time the algorithm will be run (see section A.3.3 for more details on step execution). The graph algorithm returns a `true` value if the algorithm has already selected a leader, but the algorithm will still be run for the remaining time steps. Finally, once the algorithm has been run a new `ExperimentResults` instance is created (and stored) and the corresponding results are added to the matlab file.

Once all the files have been processed this way the plots will be generated (unless the `-noplots` option has been selected), by issuing a call to the `PlotFactory` class, which will access the results stored to generate the plots.

A.3.2 Graph initialization

Figure A.32 shows the sequence diagram of a graph initialization.

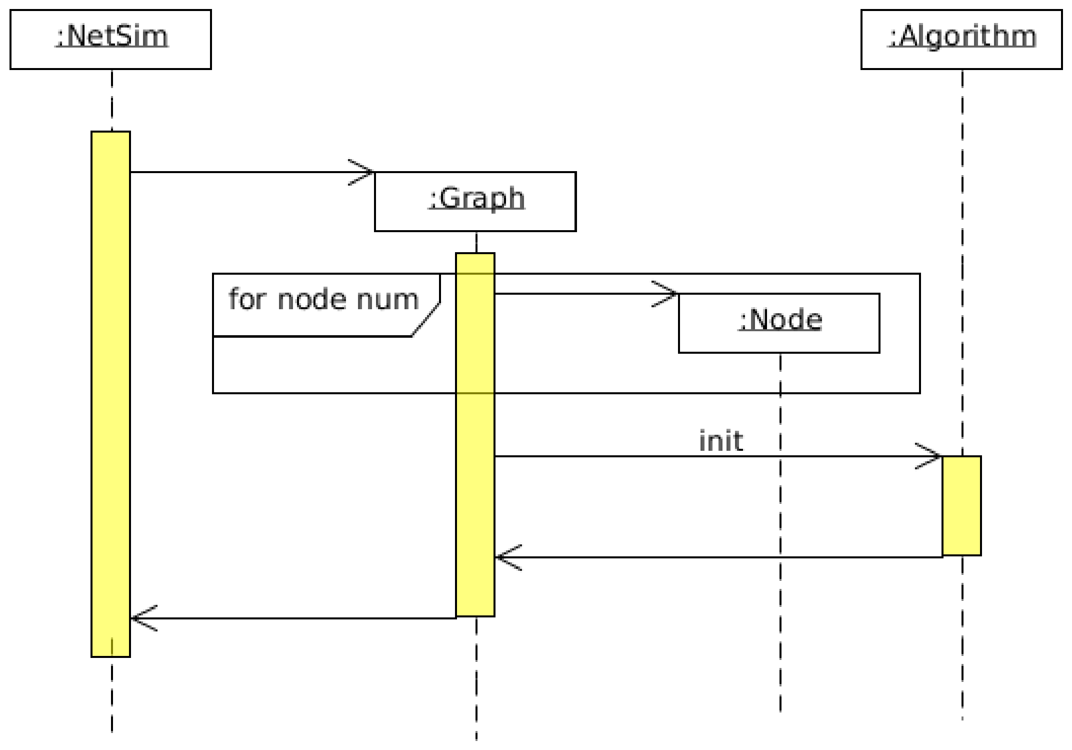


Figure A.32: Sequence diagram for the graph initialization

For each experiment to be performed a new Graph object will be created. On construction the graph object will create and initialize the values of the necessary Node objects to represent the intended network. Once all the nodes have been created a call to the init method in the corresponding algorithm will be issued (see the User Manual for more information on the Communication.init method).

A.3.3 Experiment step

Figure A.33 shows the sequence diagram of a one step in the execution of the experiment.

For each time step in the experiment the run method of the corresponding Graph object is executed. In this method, for each Node instance in the graph, the procesTasks method is called. This method will check if the node is not busy at the time, if there are tasks to be executed at the time and, if there are, it will pick one of them and execute it. Then, depending on the task, the corresponding

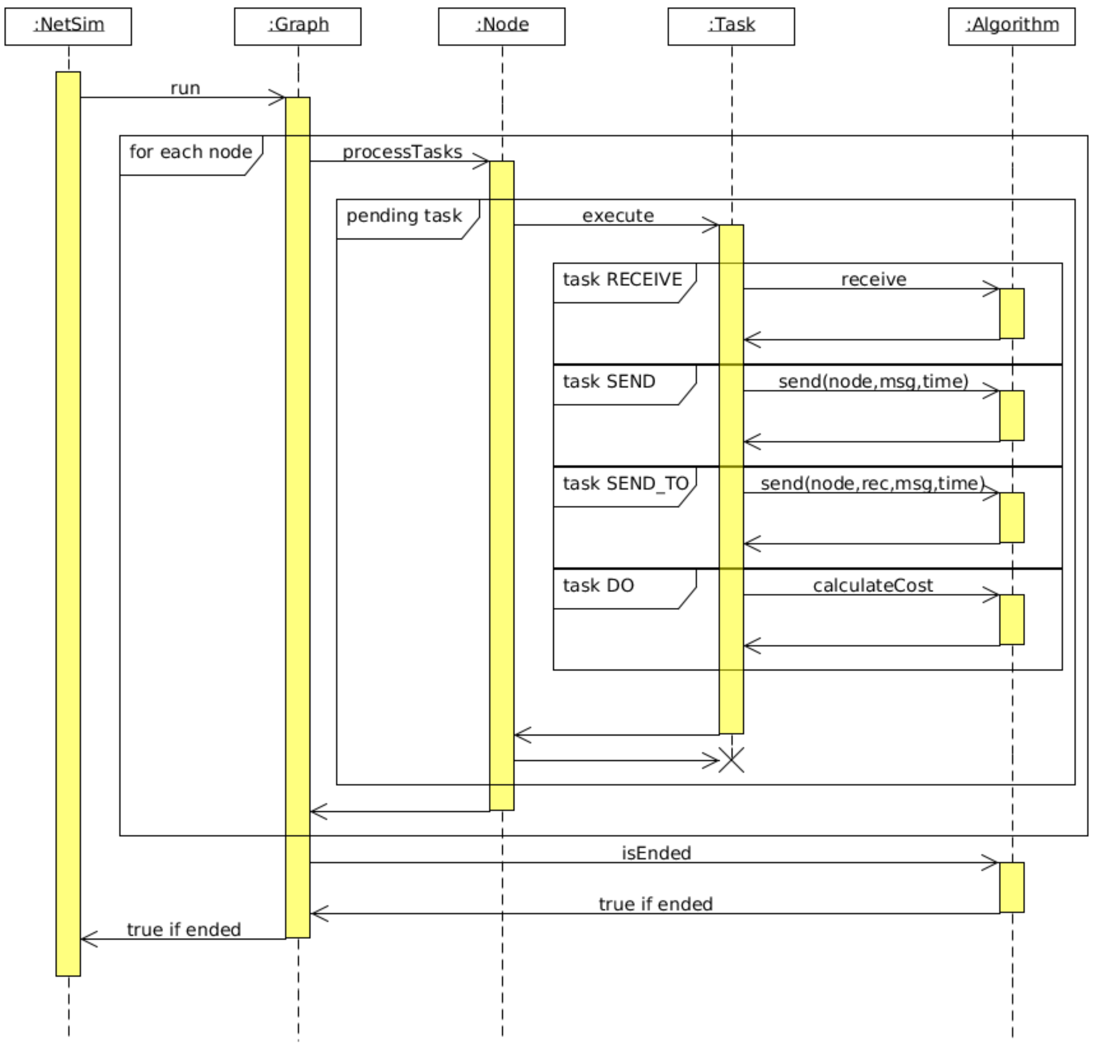


Figure A.33: Sequence diagram for an experiment step

method will be executed (see the User Manual and the JavaDoc for more details on the `Communication` class methods).

Once the task has been executed it will be removed from the task list of the node and destroyed. New tasks are created during the execution of the algorithm methods if necessary and added to the task list.

It is important to notice than the nodes represent single processor systems, and as such can only perform one task at a time.

Finally, once all nodes have executed their tasks for the current time step the `isEnded` method of the `Communication` class associated with the experiment will be called to check if the algorithm has finished. Once again it is important to notice that, even if the algorithm ends before the maximum time defined in the `.ae` file, it will be ran for the remaining steps anyway.

A.4 Further work

As NetSim was created as a tool for use during my Master thesis, and it only addressed a small part of the work to be done, the tool is quite uncompleted, and there are several parts of the code intended to provide support for other features that could not be implemented due to time constraints and lack of personnel. Some of those features, and the intended way of implementing them are described in this section.

A.4.1 Multi-task nodes

Currently nodes can only handle one task at a time. It would be possible to implement a multi-task ability by adding a second (or more) task buffer and adding the task to the corresponding buffer, for example, separating send/receive tasks on one buffer and compute tasks on another, or send and receive tasks in different buffers, then executing one task for each buffer.

A.4.2 Dynamic networks (node death)

Another interesting feature would be to simulate errors in the network by switching the node state (death or alive). For that a new parameter should be added to the `.ae` file, as a time-node pair to indicate which node should switch its state at which time (this way different algorithms would face identical scenarios providing meaningful results).

In the current implementation if a node is dead it will execute no further tasks, though it will remember which tasks it had pending. If a message is sent to a dead node it will not be added to its pending tasks, and it will be counted as a failed message (though the number of failed messages is not being used for anything at the moment).

A.4.3 Ad-hoc networks (dynamic topologies)

Another interesting feature would be to add dynamism to the current topology. By providing different network matrices at different times in the `.ae` files this could be simulated. Then, at each time step the network topology should be checked and, if required, changed, by updating the `trueConections` list of a node. This is intended to simulate the fact that a node will not be aware of the topology change.

In the current implementation a message is only sent if the recipient of the message exists in the `trueConections` list, otherwise the message is lost. Unfortunately, as the `trueConections` list is initialized as a copy of the `connections` when the `Graph` object is constructed and it is never modified it serves no purpose.

APPENDIX B

CAFE Developer Manual

B.1 Introduction

The CAFE library (Cultural Algorithm Framework Extension) is a Java built library that provides the framework to implement a “Hybrid niched cultural algorithm” (HNCA), as proposed by Ali *et al.* [25].

Cultural Algorithms (CA) are an evolution to Genetic Algorithms (GA) presented by Reynolds in 1994 [22] that use both a solution set (called population) and a knowledge set (called belief space). The population is then evolved used standard evolutionary computation techniques and at each generation the knowledge obtained is extracted and used to modify the following population.

The belief space consists in five different knowledge sources: situational, normative, topographical, domain and historic knowledge [23]. A CA then defines a set of communication strategies between the population and the belief space to extract the knowledge gained by each generation. Afterwards the gained knowledge is used to influence the new population.

In addition the algorithm presented by Ali *et al.* uses the Niche Clearing technique [24], in which each niche contains part of the population and only a few individuals (the fittest) of each niche are allowed to survive. In addition a “tabu list” is used to avoid unusable solutions.

B.1.1 Jars and Requirements

There are four different versions of the CAFE library, designed to suit different needs. All version requires Java 1.7.

CAFE bare

CAFE bare (CAFE_bare.jar) provides the core functionality of the CAFE library without any additional support or required libraries present in the classpath.

CAFE NetLogo

The CAFE NetLogo (CAFE_NL.jar) version of the library provides support for using NetLogo simulations to test the fitness of the solutions. The Netlogo jar and library folder need to be present in the classpath when using this version of CAFE.

CAFE openMPI

The CAFE openMPI (CAFE_oMPI.jar) library provides support for the parallelization of the HNCA using openMPI. The openMPI Java library needs to be added to the classpath when using this version.

CAFE full

The CAFE full library (CAFE.jar) provides support for both openMPI and NetLogo usage. This, of course, means that both openMPI and NetLogo need to be present in the classpath.

B.2 Class Architecture

Figure B.21 shows the UML class diagram of the CAFE library in all its possible versions.

Class `CulturalAlgorithmRunner` is the entry point of the library. By creating an instance of this class and calling the `run` method the algorithm is loaded and started. To load the parameters of the algorithm a `.ca` file needs to be provided to this method, as well as the maximum number of generations and the `verbose` parameter (a boolean indicating if the execution of the algorithm should generate full output).

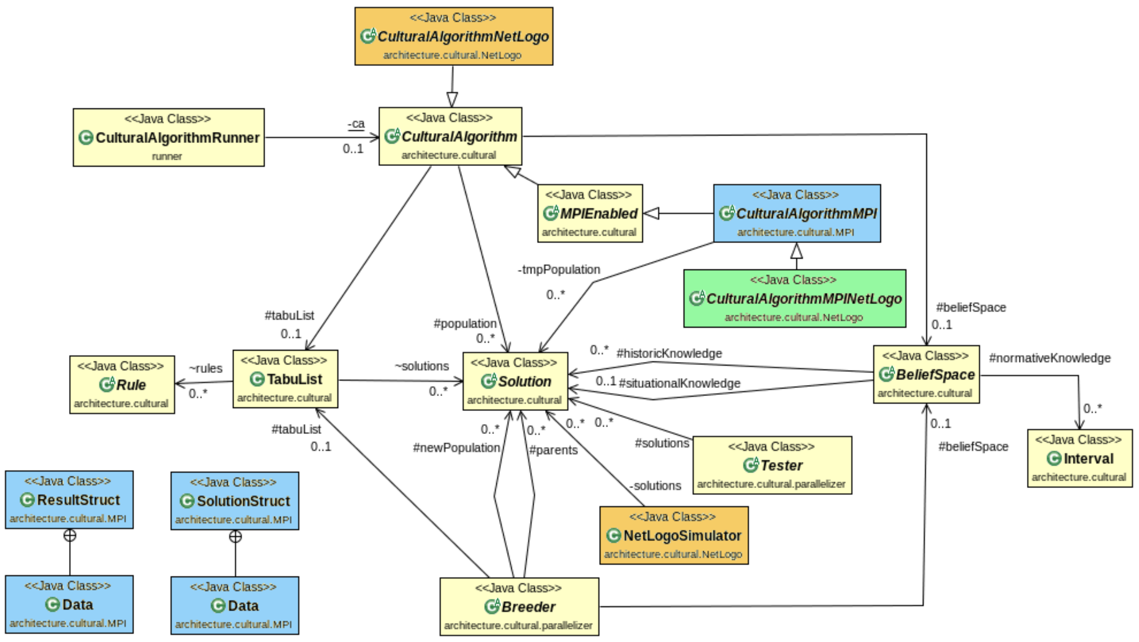
When an algorithm is run an instance of a class extending either the `CulturalAlgorithm` abstract class, the `CulturalAlgorithmNetLogo` abstract class, the `CulturalAlgorithmMPI` abstract class or the `CulturalAlgorithmMPINetLogo` abstract class(depending on the version and type used) is creaetd and evolved. This class uses a specific class extending `BeliefSpace` to store the accumulated knowledge. More information on this classes and their interactions can be found in the following sections.

B.3 Execution Flow

The execution flow of CAFE differs depending on the version of the library used. When using an openMPI-enabled version there are some minor differences to how the algorithm is executed, and especially to how the algorithm is terminated.

B.3.1 Standard Execution Flow

The CAFE library implements an HNCA. Figure B.31 shows the execution flow of the algorithm.



<<Java Class>> CulturalAlgorithmNetLogo architecture.cultural.NetLogo
<<Java Class>> CulturalAlgorithmRunner runner
-ca 0..1
<<Java Class>> CulturalAlgorithm architecture.cultural
<<Java Class>> MPIEnabled architecture.cultural
<<Java Class>> CulturalAlgorithmMPI architecture.cultural.MPI
<<Java Class>> CulturalAlgorithmMPINetLogo architecture.cultural.NetLogo
-tmpPopulation 0..*
#tabuList 0..1
#population 0..*
#beliefSpace 0..1
<<Java Class>> Rule architecture.cultural
~rules 0..*
<<Java Class>> TabuList architecture.cultural
~solutions 0..*
<<Java Class>> Solution architecture.cultural
0..* #historicKnowledge
0..1 #situationalKnowledge
<<Java Class>> BeliefSpace architecture.cultural
#normativeKnowledge 0..*
<<Java Class>> Interval architecture.cultural
#tabuList 0..1
#newPopulation 0..*
#parents 0..*
0..* 0..*
#solutions
-solutions
<<Java Class>> Tester architecture.cultural.parallelizer
<<Java Class>> NetLogoSimulator architecture.cultural.NetLogo
0..1 #beliefSpace
<<Java Class>> ResultStruct architecture.cultural.MPI
<<Java Class>> SolutionStruct architecture.cultural.MPI
<<Java Class>> Data architecture.cultural.MPI
<<Java Class>> Data architecture.cultural.MPI
<<Java Class>> Breeder architecture.cultural.parallelizer

Figure B.21: UML Diagram of the CAFE library. The classes with yellow background are present in all versions of the library. Those with orange backgrounds are present in the NetLogo and full versions. Those with blue background are present in the openMPI and full versions. Finally, the class with green background is only present in the full version.

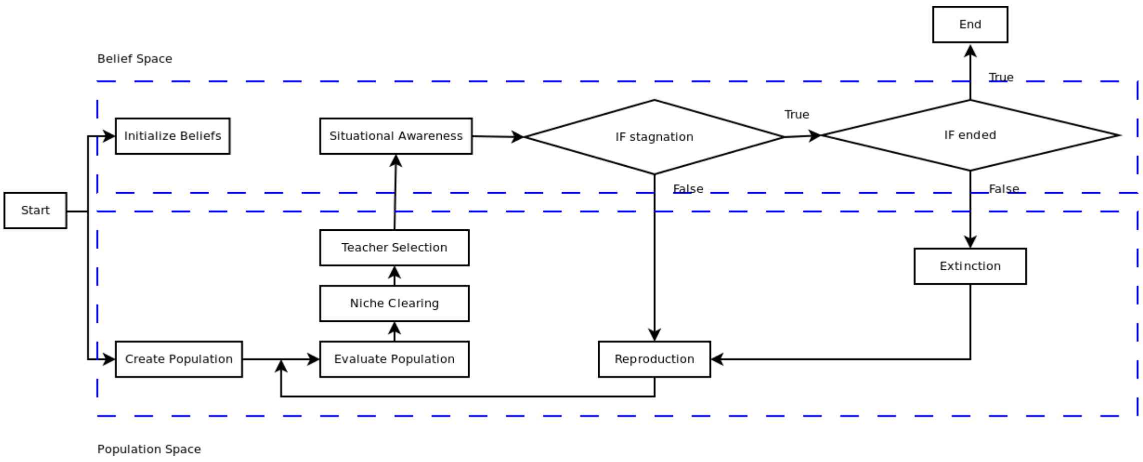


Figure B.31: Flow diagram for the CAFE algorithm in its standard configuration.

Initially both the population space and the belief space are initialized. Then, each element of the population is tested to compute its fitness (this operation may be parallelized). After that niche clearing is applied to ensure the sparseness of the population. Once the niching has been applied a subset of the population is selected and used to influence the belief space. The methods provided by default on the CAFE library allow the belief space to contain situational awareness (the best solution found so far) and normative awareness (intervals of interest).

Once the belief space has been updated the population is tested for stagnation (this operation may be defined by the user). If there is no stagnation the population is evolved generating new individuals and repeating the process (the generation of new individuals can also be paralellized). If a stagnation condition is detected and the algorithm hasn't yet meet the termination conditions an extinction event is applied and a new generation is created, continuing the loop.

If the ending condition has been met the algorithm then finishes.

B.3.2 openMPI-enabled Execution Flow

The open-MPI enabled version of the CAFE library operates in the same way as the standard version with a few minor modifications. The most important one is that the termination condition in this version is purely a certain amount of generations, which means the algorithm will run for that many generations and then stop.

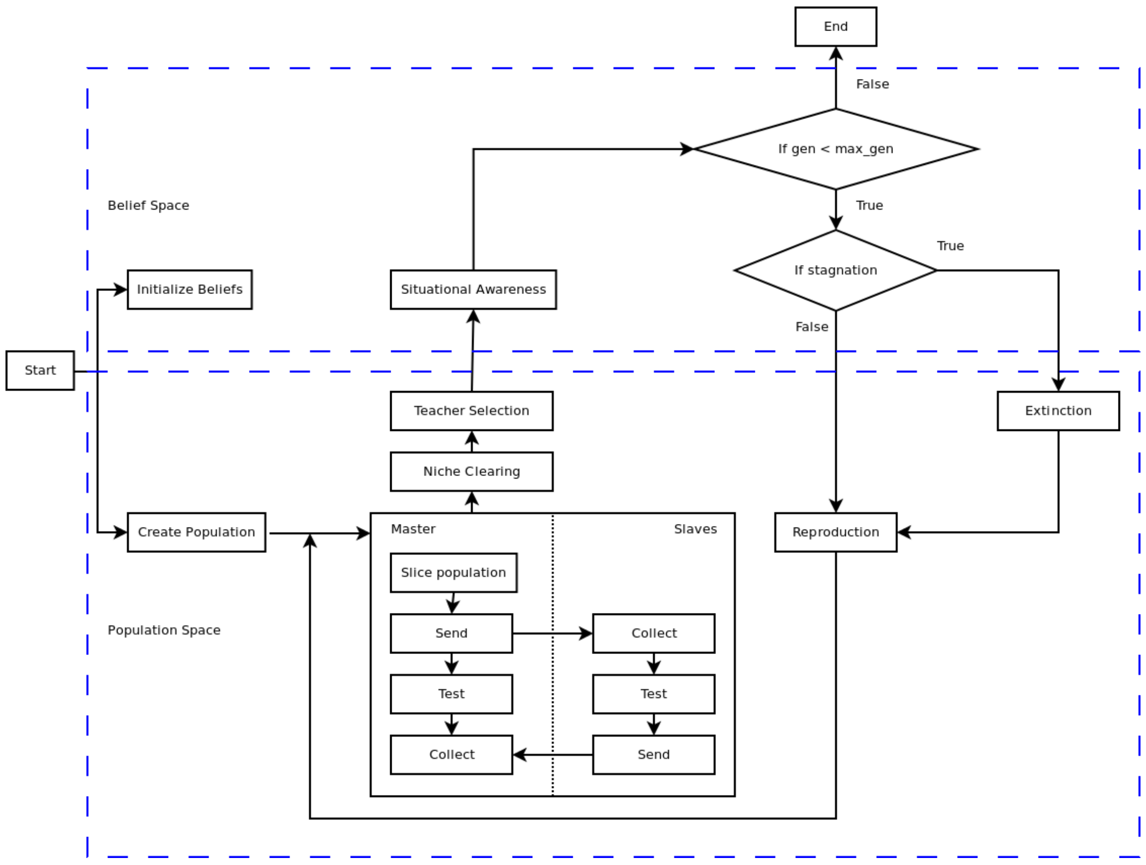


Figure B.32: Flow diagram for the CAFE algorithm in its openMPI-enabled configuration.

When the openMPI-enabled version of the algorithm is used several instances of the program will be created and run. One of this instances is the Master instance, while the rest are Slave instances. The Master will then run the bulk of the algorithm as described in the previous section. When the testing phase is reached, however, the algorithm differs. In this case the Master will break down the population in as many chunks as processes exist (including itself) and send each Slave process its chunk of the population. Then all the processes will compute the fitnesses of their assigned individuals. Once this is done each Slave process will send its chunk back to the master. Finally the master process will collect all the chunks and continue the algorithm.

B.4 Building a Cultural Algorithm

The CAFE library provides the framework for the HNCA. Most of the required methods and parameters, however are problem-dependant, and as such need to be defined for each specific experiment. In order to do so a series of class need to be extended to provide the desired functionality.

B.4.1 The Solution class

A class extending the `Solution` class needs to be implemented. This class contains all the information on a specific solution (individual). The following methods need to be implemented:

distanceTo method

The `distanceTo(Solution solution)` method is designed for the Niching phase of the algorithm. This method should return the distance between the current solution and the passed solution. This will be used to group the solutions during the niching phase.

toString method

The `toString()` method generates a String version of this object. It is used to create the result files of the algorithm execution, and it should return a string sequence of the values of the different parameters in the solution.

testString method

The `testString()` method is used in the NetLogo version of the library. It should return the different attributes of the solution in a NetLogo-friendly way, so they can be used as the parameters of the NetLogo reported called to evaluate the solution.

toStringGroup method

The `toStringGroup()` method is a method that should generate a string that identifies the niche to which this solution belongs. This is used when printing the historical solution for later analysis.

toMPIString method

The `toMPIString`() method is used in the openMPI-enabled version of the library. It should return a string with the parameters of this solution that can be then parsed by this solution object, so the values can be sent through MPI.

parse method

The `parse`(`String string`) method is used in the openMPI-enabled version of the library. It should load the parameter values from the string generated by the `toMPIString`() method.

B.4.2 The BeliefSpace class

A class extending the `BeliefSpace` class needs to be implemented. The `BeliefSpace` class contains the methods to use and manipulate the belief space. In the abstract class support for situational and normative knowledge are implemented.

accept method

The `accept`(`ArrayList<Solution>` teachers) method is the method in charge of modifying the belief space. The method receives a list of solutions that have been selected for this task. It should then iterate over this solutions extracting the required knowledge (store the best solution if its better than the current situational knowledge, modify the intervals of the normative knowledge, and so on).

ended method

The `ended`() method should return a boolean value of “true” if the algorithm has reached the termination condition. This method is used in the standard flow to determine if the algorithm is finished.

stagnation method

The `stagnation`() method should return a boolean value of “true” if stagnation is detected and an extinction event needs to be triggered (the most common way to detect it is if more than n generations have passed without change in the situational knowledge). This method is used to determine if an extinction event is needed.

B.4.3 The Rule class

If the “tabu list” is used then one or more classes extending the `Rule` class need to be implemented. Those classes contain the rules against which each solution will be tested to determine if it is tabu or not.

isTabu method

The `isTabu(Solution solution)` method will return a “true” value if the solution breaks the rule and should not be tested or used.

B.4.4 The CulturalAlgorithm class

A class extending the `CulturalAlgorithm` class must be implemented. This class is the main class of the algorithm and contains all the methods for its execution.

init method

The `init()` method is the first method that will be called. This method creates the initial population and the initial belief space as well as setting up any necessary parameters for the algorithm.

loadParams method

The `loadParams(String[] params)` can be used to pass aditional parameters to the algorithm. Those parameters are passed to the `CulturalAlgorithmRunner.run` as a `String` array. This method is called after the `init` method but before the the algorithm is started.

test method

The `test()` method is the method responsible for performing the tests on the population to compute their fitnesses.

niching method

The `niching()` method is the method responsible of performing the niching on the population. The specific implementation of the niching strategy is up to the programmer, but undesired solutions should be removed from the population before other methods are called.

accept method

The `accept()` method selects the members of the population that are used to influence the belief space. It should then call the `BeliefSpace.accept` method for that purpose.

reproduce method

The `reproduce()` method is responsible for creating a new population by reproduction and or mutation. The specific strategies used are left to the programmers.

extinction method

The extinction() method is called whenever stagnation is detected. The specific strategy is left to the programmers.

close method

The close() method is called at the end of the algorithm.

B.4.5 Parallelization

Both reproduction and testing can be parallelized in every version of the library. This parallelization, however, is done on a single computer, and bounded by the number of cores this computer possesses.

To implement this parallelization one of the Tester or Breeder classes have to be extended. Then it is enough to instantiate the class and use fjPool.invoke(Class) to launch the parallelized version of the algorithm.

The Tester Class

The Tester class is designed to parallelize solution tests. Three methods have to be implemented. The initTest() method will be called once before the testing is started. The testSolution(Solution solution) method will be called once for each solution in the population (this method is parallelized). The endTest() method will be called once after all solutions have been tested.

The Breeder Class

The Breeder class is designed to parallelize solution generation (reproduction). The breed() method must be implemented. This method is responsible to generate a new solution (or multiple solutions) and deposit it in the newPopulation list.

B.5 Building a NetLogo Cultural Algorithm

To use a NetLogo simulation the CAFE_NL.jar or the CAFE.jar version of the library have to be used. Instead of extending the CulturalAlgorithm class the CulturalAlgorithmNetLogo class should be extended. This class have the same methods than the CulturalAlgorithm, but the test() method is already implemented to use a parellelized NetLogo tester, implemented in the NetLogoSimulator class. The corresponding parameters have to be used in the .ca file.

B.6 Building an openMPI-enabled Algorithm

To use an openMPI-enabled simulation the CAFE_oMPI.jar or the CAFE.jar version of the library have to be used. Instead of extending the CulturalAlgorithm class the `CulturalAlgorithmMPI` class should be extended. This class have the same methods than the `CulturalAlgorithm`. The corresponding parameters have to be used in the `.ca` file.

B.7 Building an openMPI and NetLogo supported algorithm

To use a NetLogo simulation the CAFE.jar version of the library has to be used. Instead of extending the `CulturalAlgorithmMPI` class the `CulturalAlgorithmMPINetLogo` class should be extended. This class have the same methods than the `CulturalAlgorithmMPI`, but the `test`() method is already implemented to use a parellelized NetLogo tester, implemented in the `NetLogoSimulator` class. The corresponding parameters have to be used in the `.ca` file.

B.8 The `.ca` file

The parameters necessary for a CAFE execution are given in a `.ca` file. Tables B.81, B.82 and B.83 show the specific parameters that may be used grouped by version of the CAFE library.

B.9 Implemented classes and methods

The CAFE library provides a series of already implemented classes and methods that simplify the design and implementation of a HNCA. Those methods, their utilities and their intended uses are described in this section.

B.9.1 Solution class

The `Solution` class contains two attributes (and their respective getters and setters) that are commonly used in any implementation of an HNCA. Those are the `fitness` and `niched` attributes. The first is intended to hold the fitness of the specific solution after testing. The `niched` attribute is used to reduce the time taken by the niching operation. When an individual has been included in a niche this attribute should be set to `true`, so it would not be checked again for niching.

B.9.2 BeliefSpace class

The `BeliefSpace` class contains a few useful methods. Three of them are designed for printing information: `toString`(), `historyToString`() and `getHeaders`(`String solutionHeader`). The first generates a string with the current state of the BeliefSpace (normative and situational knowledge), the second generates a string with the contents of the history knowledge as a `.csv` table (each individual in a different line) and the last generates the header for the belief space file.

Table B.81: CAFE .ca general file parameters

Parameter	Value	Description
%	Comment line	Any line beginning with a % symbol will be ignored
Class	Class name	Complete class path to the class extending the CulturalAlgorithm class (or any of its versions). This must be the first non-commented line of the file.
-ofolder	Path	Folder were the results of the execution will be stored.
-header	String	This string will be used as the header for the solutions in the generated .csv files.
-slices	Parallel threads	Number of parallel NetLogo simulations that will be run.
-breedDepth	Number of splits	Number of times the population will be sliced when using a Breeder implementation.
-breedSliceSize	Number of slices	Number of blocks in which the population will be spliced at each split when using a Breeder implementation. In total $breedSliceSize^{breedDepth}$ threads will be generated.
-rules	Class name list	Comma-separated list of classes implementing the Rule class that will be used for the tabu list.

Table B.82: CAFE .ca file NetLogo parameters

Parameter	Value	Description
-simFile	Path to the NetLogo file	Path to the NetLogo file that will be used to test the solutions.
-simMethod	NetLogo method	Name of the NetLogo reporter that will be used to test the solutions.
-extractor	Netlogo method	Name of the NetLogo reporter that will be called after the tests.
-extractions	Number	Number of times the extractor method has to be called.

In addition the BeliefSpace class contains two methods that generate mutation driven by the normative knowledge. Those are mutate(double value, String chromosome) and getRandom(String chromosome). The first method uses equation B.91 to generate a new value for a chromosome with a continuous value. The second uses equation B.92 to generate a completely new value. Both receive the name of the mutated chromosome as a parameter. In addition the mutate method requires the

Table B.83: CAFE .ca file openMPI parameters

Parameter	Value	Description
-mpiGen	Number of generations	Number of generations that this algorithm should be ran for.
-mpiPopSize	Population size	Number of solutions in the population.
-mpiArgs	Arguments	Arguments passed to the MPI.Init(String[] arguments) method.
-solutionClass	Class name	Complete class path to the class extending Solution.

current value of the chromosome as a parameter.

$$k_{i+1} = k_i + \mathcal{N}(0,1)s(N_K) \tag{B.91}$$

Where k_{i+1} is the mutated value of the chromosome, k_i is the current value, $\mathcal{N}(0,1)$ is a random value drawn from a normal distribution with mean 0 and variance 1 and $s(N_K)$ is the size of the interval for that chromosome stored in the normative knowledge.

$$k_{i+1} = N_K^l + \mathcal{N}(0,1)s(N_K) \tag{B.92}$$

Where k_{i+1} is the mutated value of the chromosome, N_K^l is the lower bound of the interval for that chromosome stored in the normative knowledge, $\mathcal{N}(0,1)$ is a random value drawn from a normal distribution with mean 0 and variance 1 and $s(N_K)$ is the size of the interval for that chromosome stored in the normative knowledge.

B.9.3 Interval class

The Interval class is a class designed to contain the normative knowledge associated to a specific chromosome. It is intended to be used in the different methods of the BeliefSpace class. It should be stored in the BeliefSpace.normativeKnowledge attribute, which is a HashMap that maps intervals to strings (chromosome names).

The Interval class contains the minimum and maximum bounds of the interval, as well as the fitnesses associated with those values. It has getters and setters for each attribute. It contains the methods upgradeMin(double value, double fitness) and upgradeMax(double value, double fitness) and size(). The first two methods upgrade the minimum and maximum values of the interval if the new values increase its size, or if the fitness is better than the stored fitness. The size() computes the size of the interval.

The Interval class contains two methods for printing. The toString() method generates a semicolon-separated string with the values of all four attributes. The getHeaders() method generates the headers used in the BeliefSpace class.

B.9.4 TabuList class

The `TabuList` class contains the logic for a tabu list. It is designed to be used during the breeding step to remove unwanted individuals from the population. It contains two attributes, `rules` and `solutions`. The `rules` attribute is a list containing all the `Rule` subclasses that will be used by the tabu list. The `solutions` attribute contains a list of solutions.

The `init(String rules)` method takes a comma-separated list of class paths to the rules that should be used as a parameter. This method loads the required tabu rules, and it is called during initialization if the `-rules` parameter is used in the `.ca` file.

The tabu list contains two methods for tabu control: `isTabu(Solution solution)` and `isTabuStore(Solution solution)`. The first method simply calls the `isTabu(Solution solution)` of each `Rule` in the rules list, and returns `true` if any of them does. The second method does the same computation, but afterwards checks the `solutions` list. If the individual is already present in the `solutions` list then it returns `true`, otherwise it adds this solution to the `solutions` list and returns `false`.

B.9.5 CulturalAlgorithm class

The `CulturalAlgorithm` class, as well as its subclasses `CulturalAlgorithmNetLogo`, `CulturalAlgorithmMPI` and `CulturalAlgorithmMPINetLogo`, contain several methods that should not be overwritten or called. Those methods control the flow of the algorithm, the initialization of the different parameters, the handling of messages (if the openMPI-enabled version is used) or the parallel test implementation (if the NetLogo version is used). More information can be found on this methods can be found on the corresponding Javadocs.

APPENDIX C

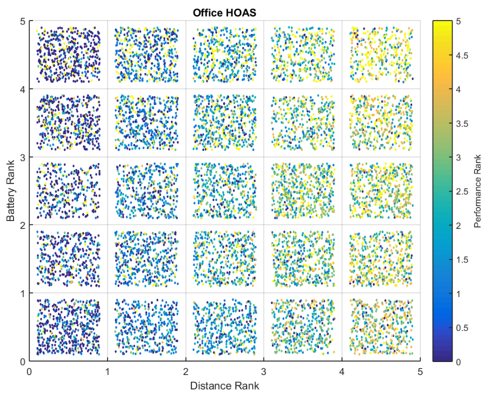
Cost Function Test Results

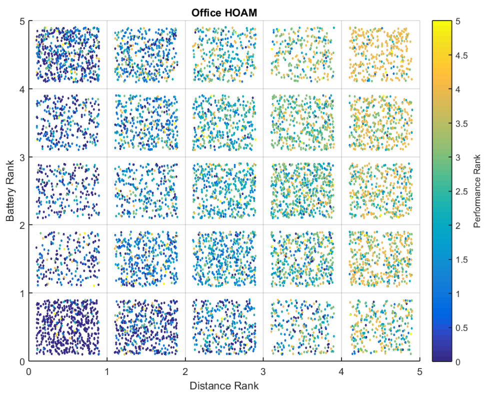
This appendix presents the results obtained in the different tests performed to obtain the cost function.

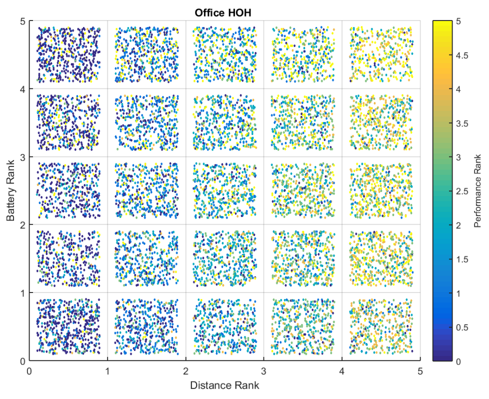
C.1 Pre-analysis Results

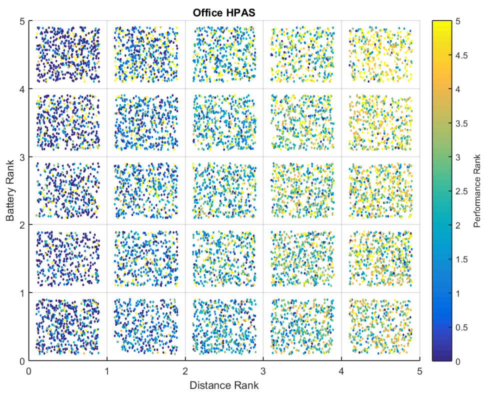
The following pages presents the plots showing the performance rank versus distance-battery rank pair for each combination of strategy.

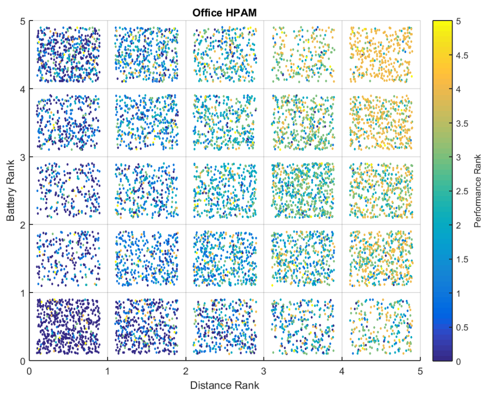
C.1.1 Office scenario

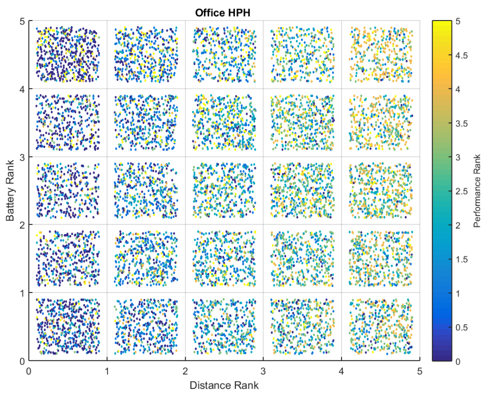


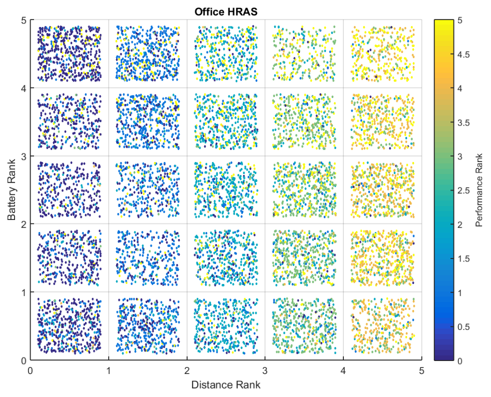


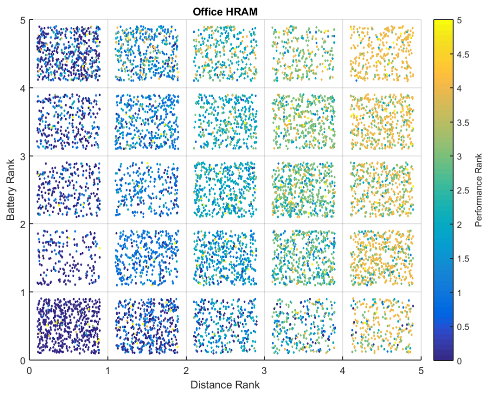


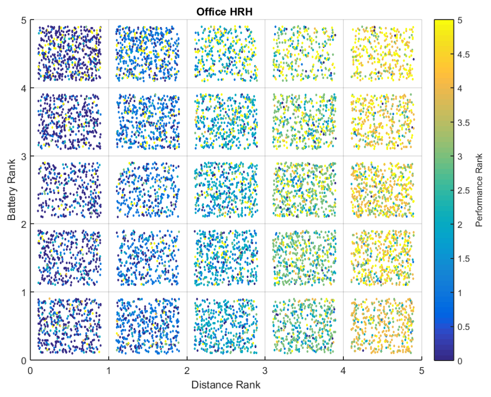


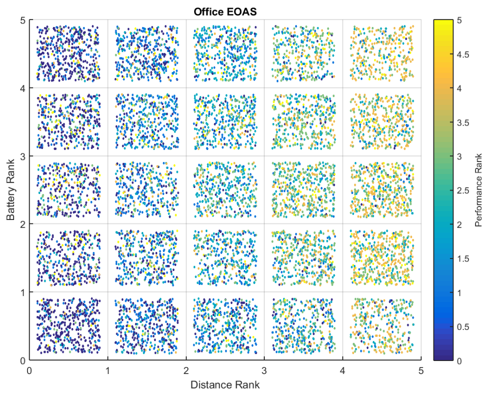


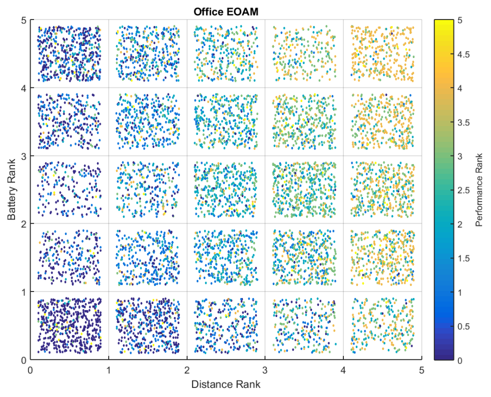


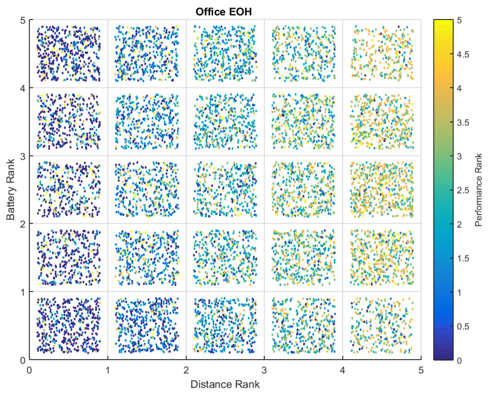

Office HRAS
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

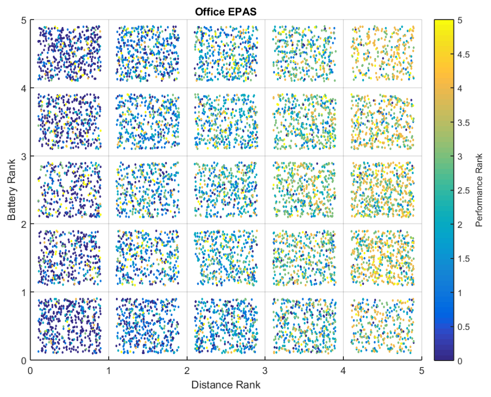

Office HRAM
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

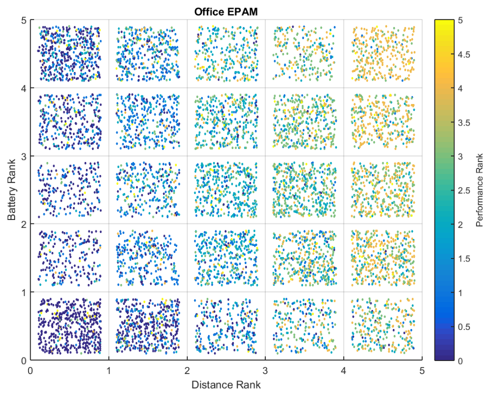

Office HRH
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

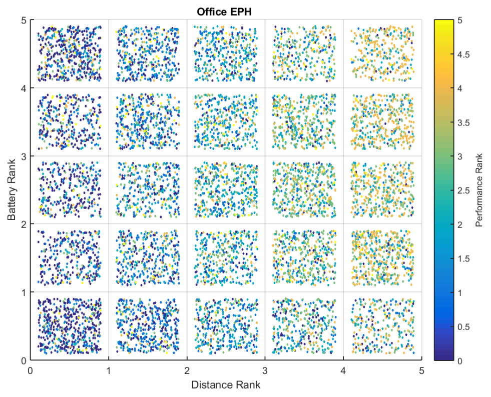

Office EOAS
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

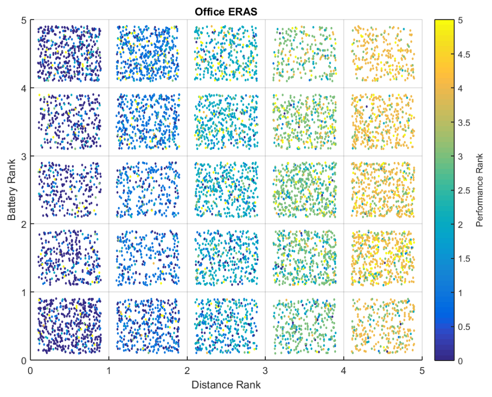

Office EOAM
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

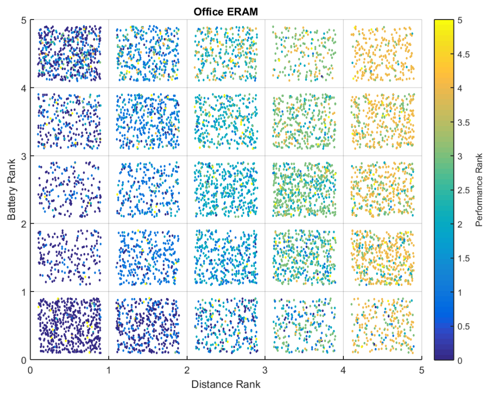

Office EOH
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

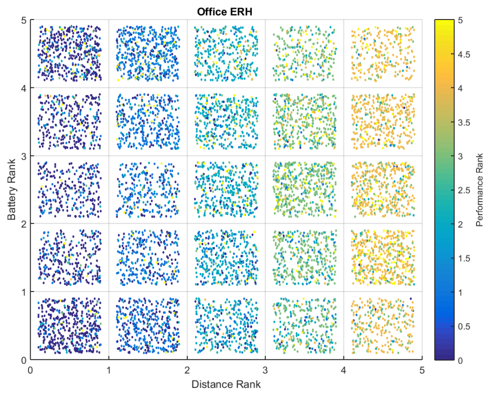

Office EPAS
Battery Rank
Distance Rank
Performance Rank


Office EPAM
Battery Rank
Distance Rank
Performance Rank

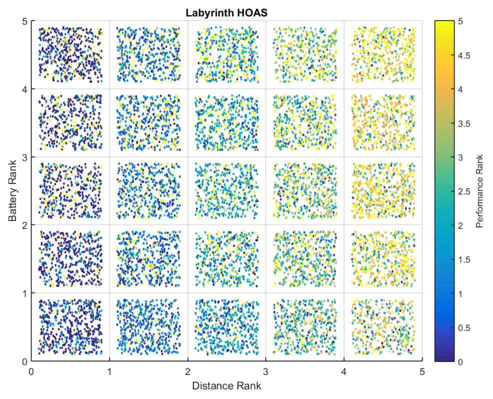

Office EPH
Battery Rank
Distance Rank
Performance Rank

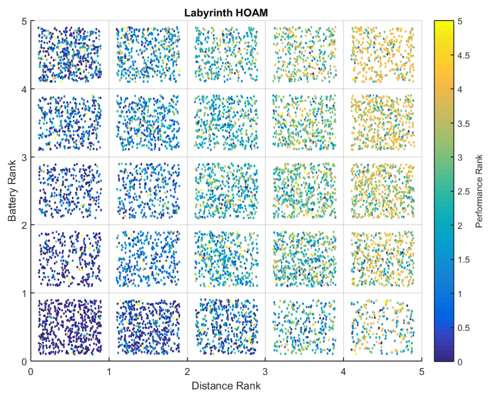

Office ERAS
Battery Rank
Distance Rank
Performance Rank

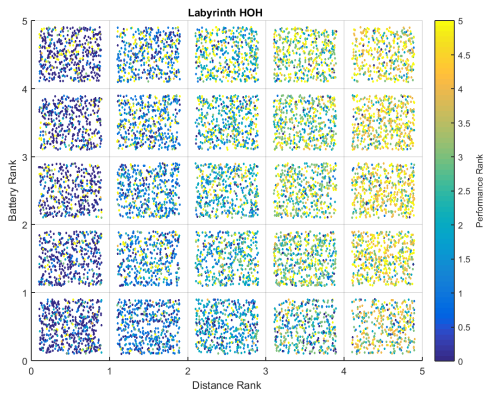

Office ERAM
Battery Rank
Distance Rank
Performance Rank

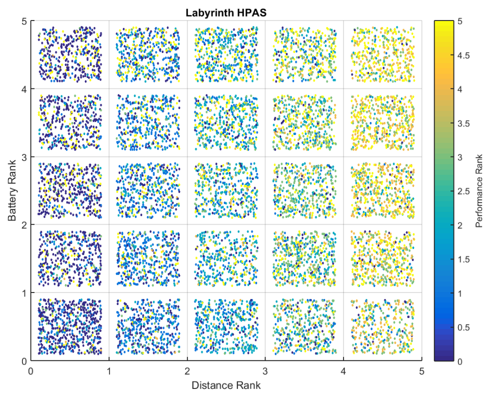

Office ERH
Battery Rank
Distance Rank
Performance Rank

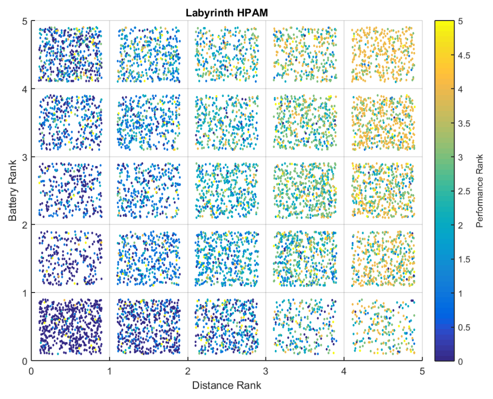
C.1.2 Labyrinth scenario

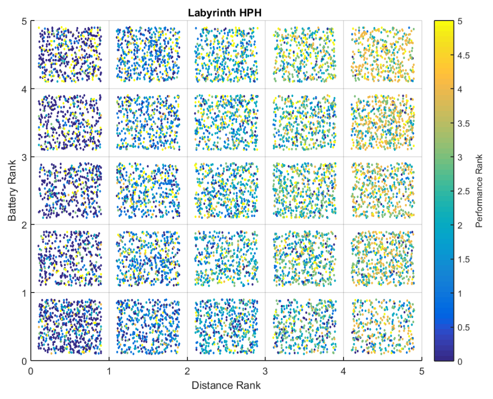


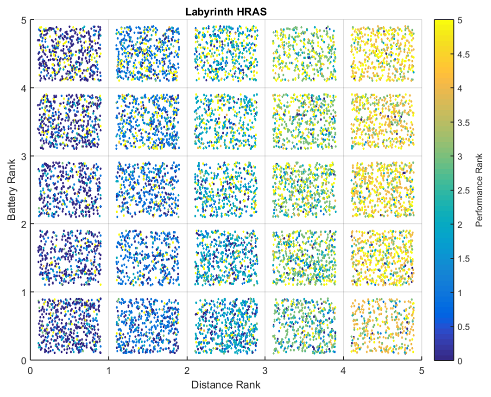


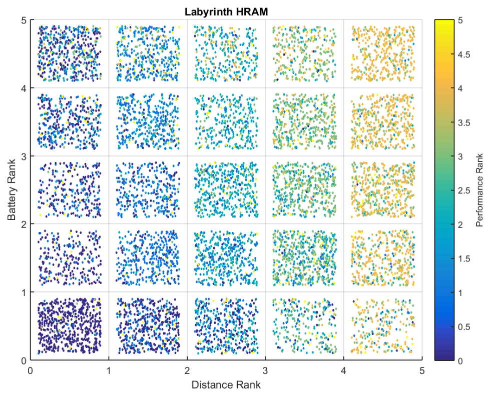


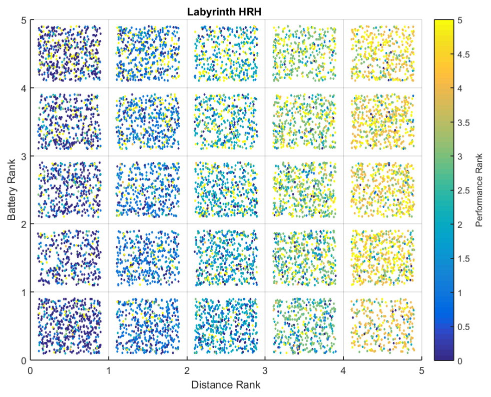


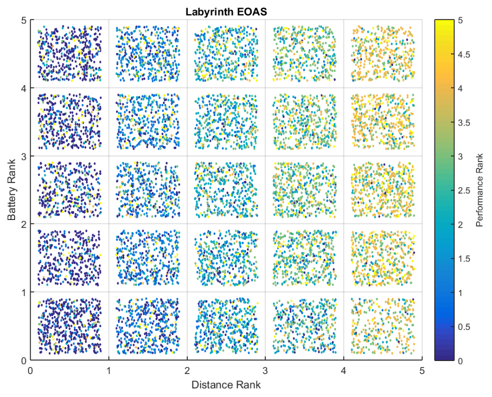


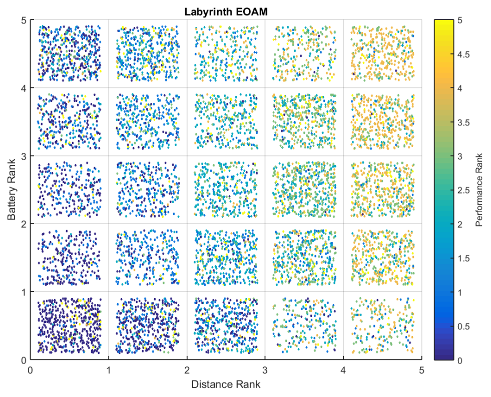


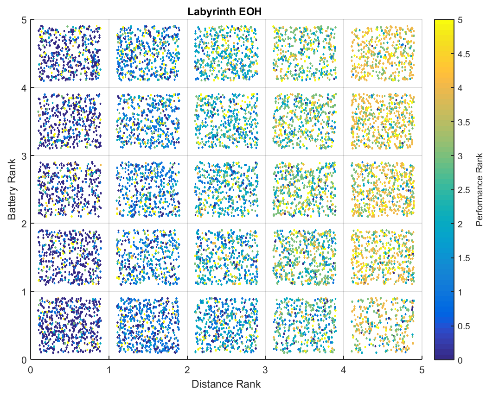

Labyrinth HRAS
Battery Rank
Distance Rank
Performance Rank

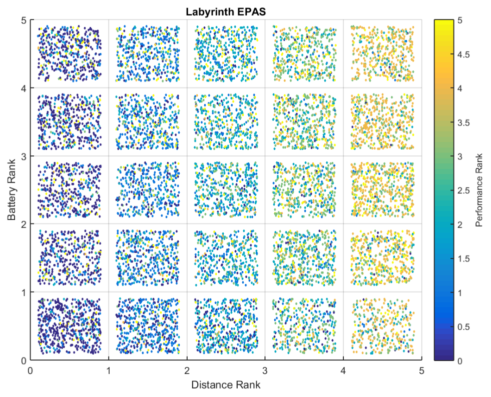

Labyrinth HRAM
Battery Rank
Distance Rank
Performance Rank

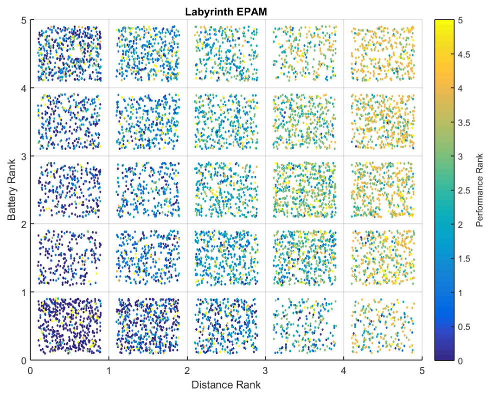

Labyrinth HRH
Battery Rank
Distance Rank
Performance Rank

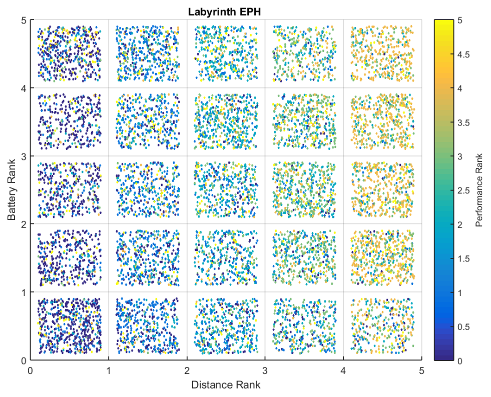

Labyrinth EOAS
Battery Rank
Distance Rank
Performance Rank

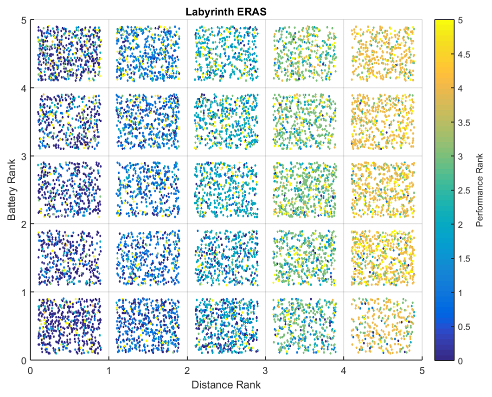

Labyrinth EOAM
Battery Rank
Distance Rank
Performance Rank

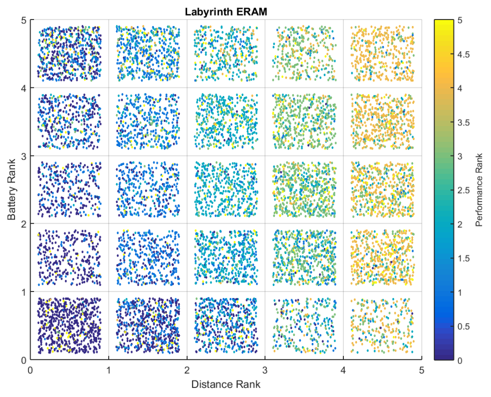

Labyrinth EOH
Battery Rank
Distance Rank
Performance Rank

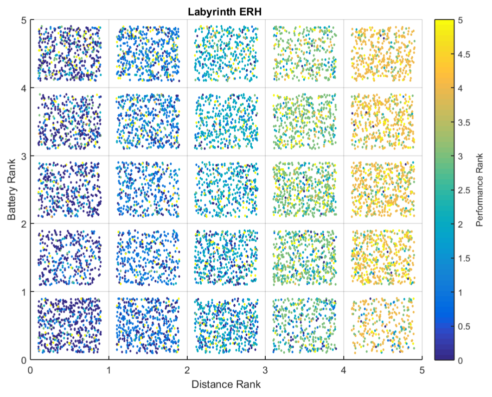

Labyrinth EPAS
Battery Rank
Distance Rank
Performance Rank


Labyrinth EPAM
Battery Rank
Distance Rank
Performance Rank

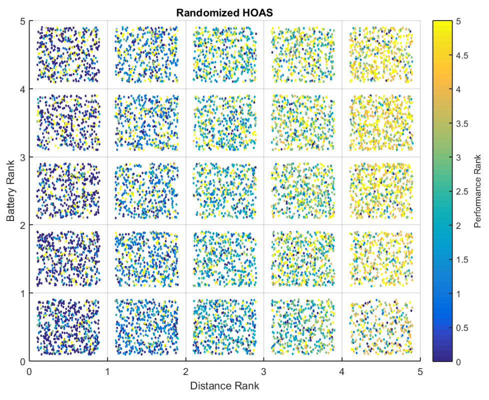

Labyrinth EPH
Battery Rank
Distance Rank
Performance Rank

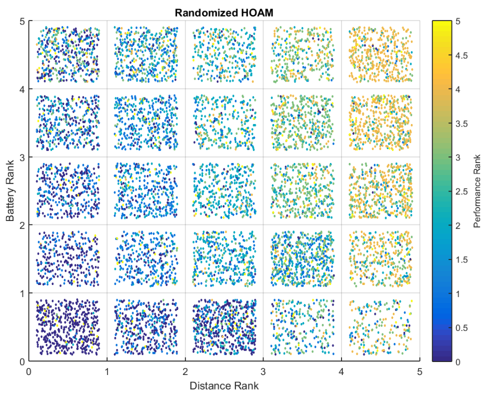

Labyrinth ERAS
Battery Rank
Distance Rank
Performance Rank

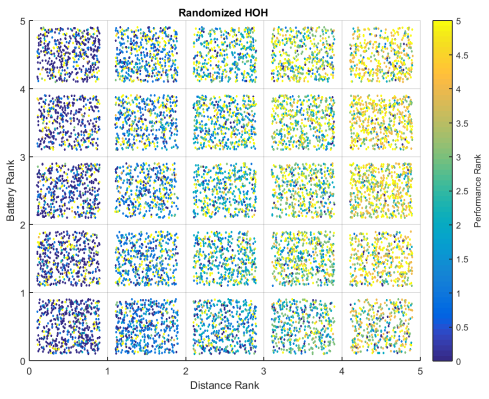

Labyrinth ERAM
Battery Rank
Distance Rank
Performance Rank

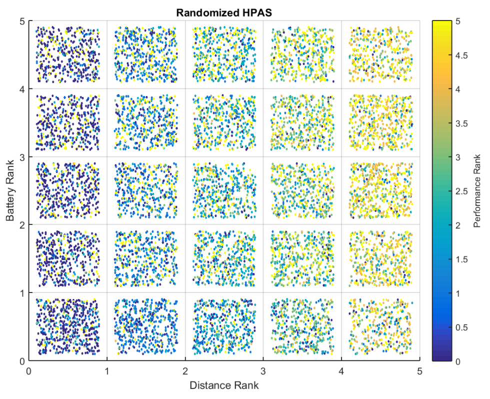

Labyrinth ERH
Battery Rank
Distance Rank
Performance Rank

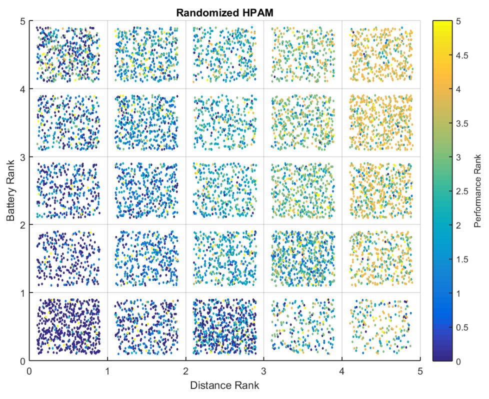
C.1.3 Randomized scenario

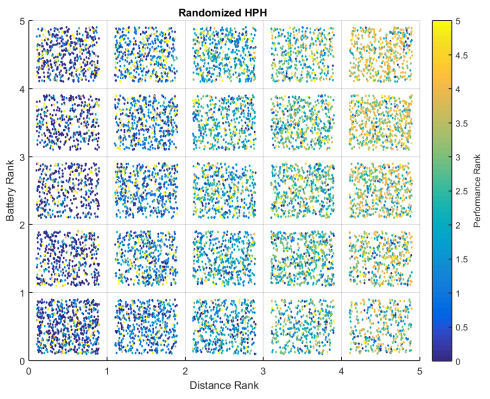


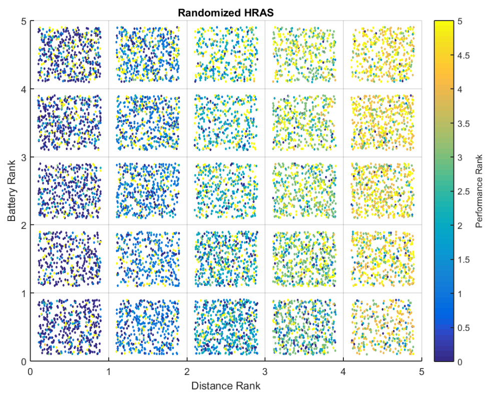


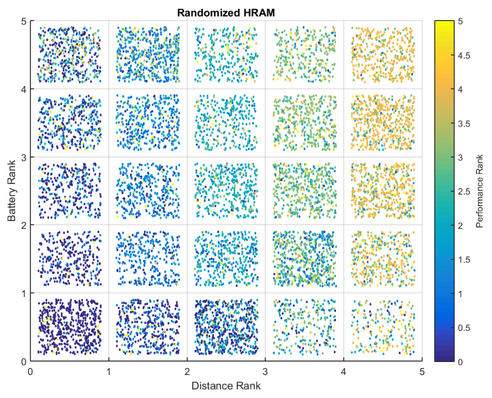


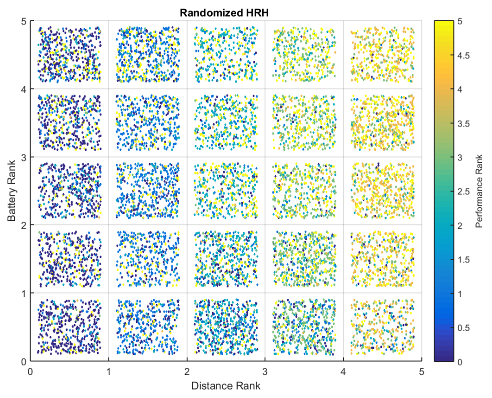


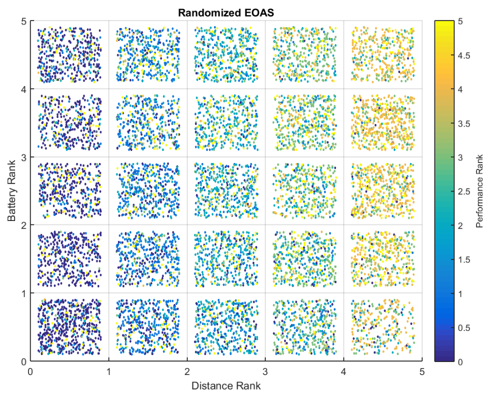


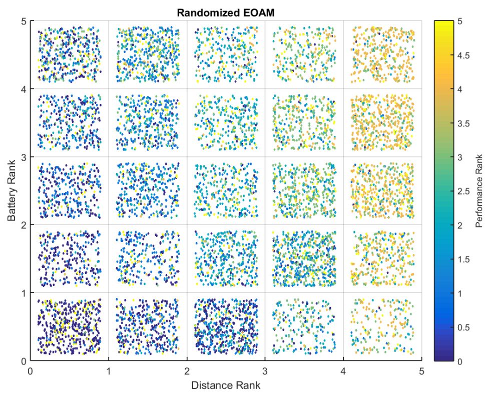


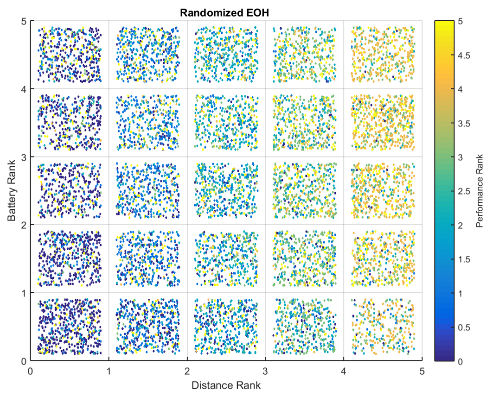

Randomized HRAS
Battery Rank
Distance Rank
Performance Rank

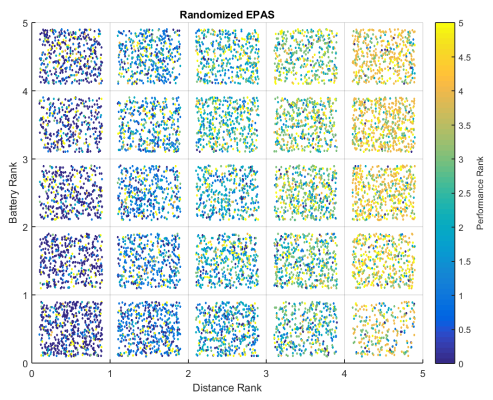

Randomized HRAM
Battery Rank
Distance Rank
Performance Rank

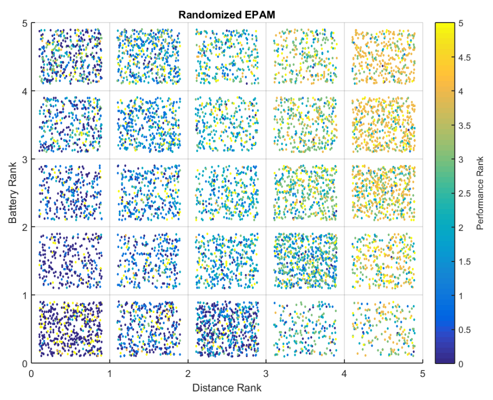

Randomized HRH
Battery Rank
Distance Rank
Performance Rank

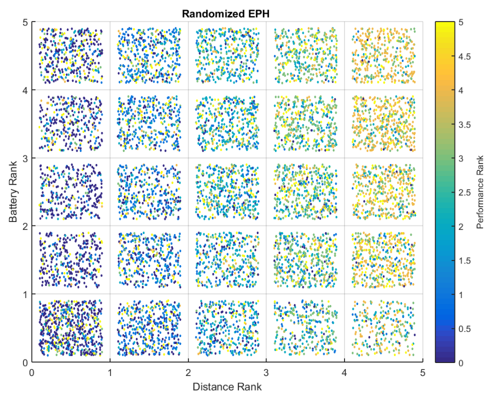

Randomized EOAS
Battery Rank
Distance Rank
Performance Rank

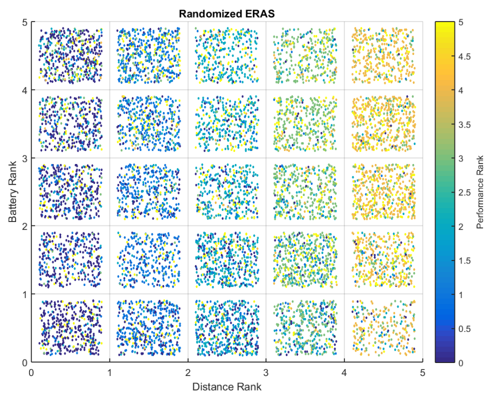

Randomized EOAM
Battery Rank
Distance Rank
Performance Rank

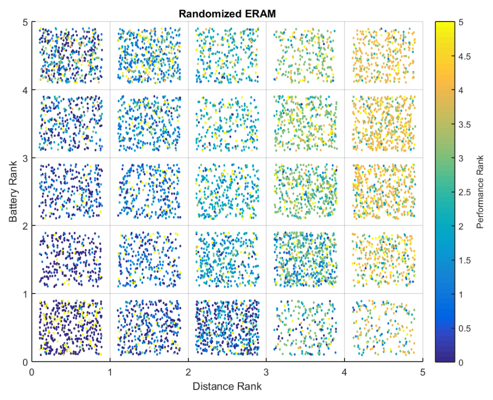

Randomized EOH
Battery Rank
Distance Rank
Performance Rank

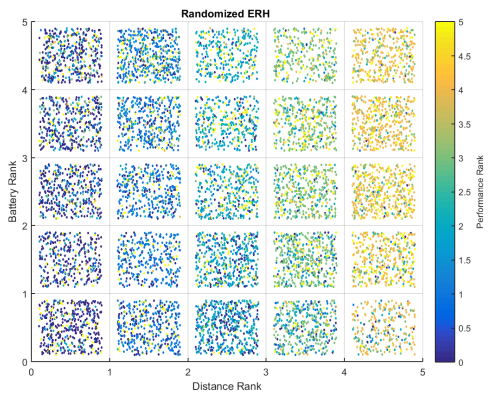

Randomized EPAS
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5


Randomized EPAM
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

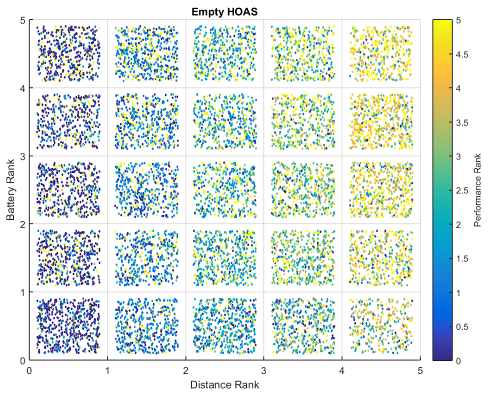

Randomized EPH
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

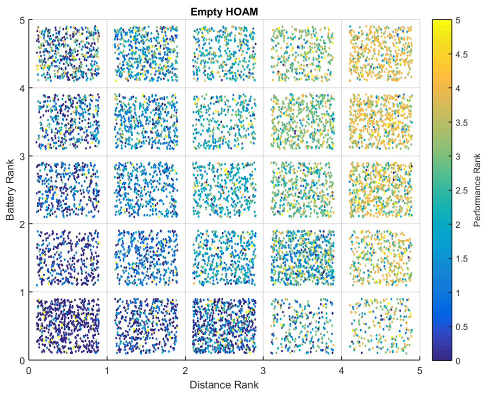

Randomized ERAS
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

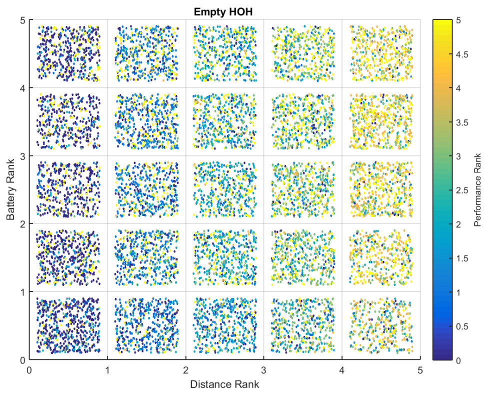

Randomized ERAM
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

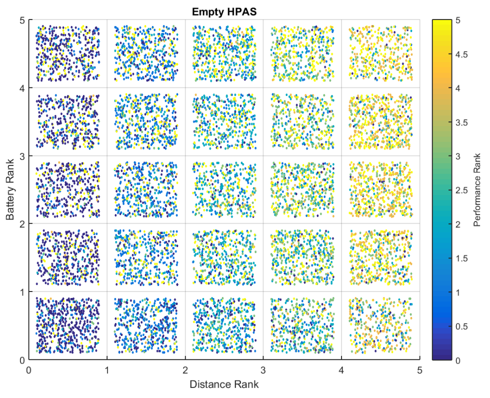

Randomized ERH
Battery Rank
Distance Rank
Performance Rank
0
1
2
3
4
5
0.5
1.5
2.5
3.5
4.5

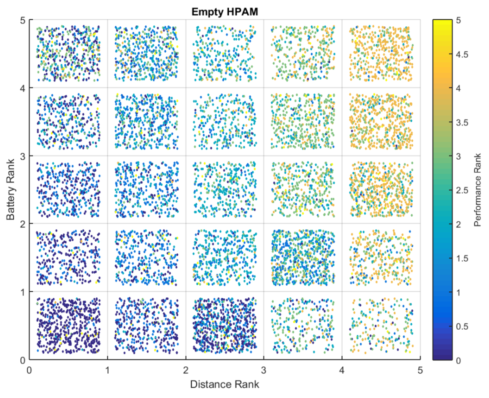
C.1.4 Empty scenario

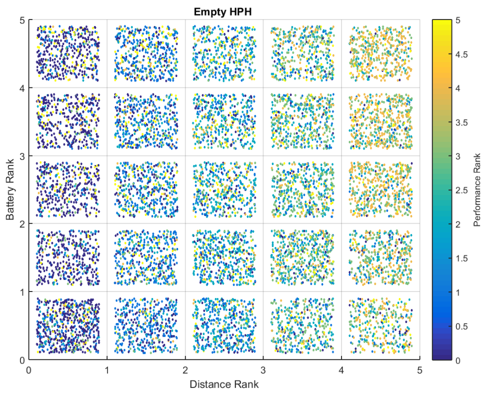


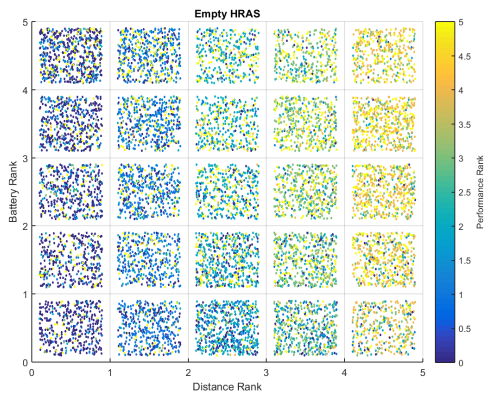


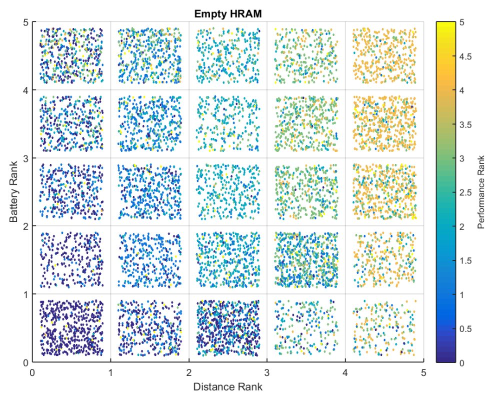


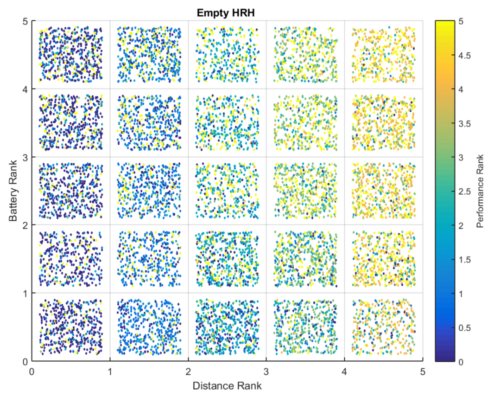


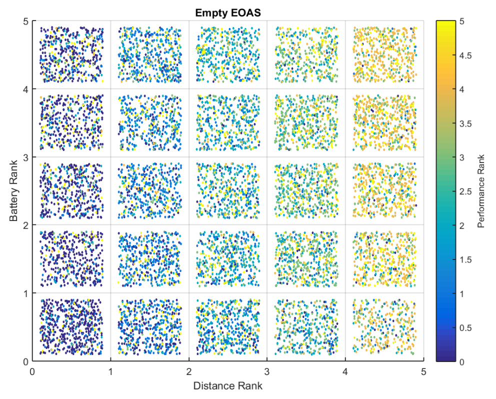


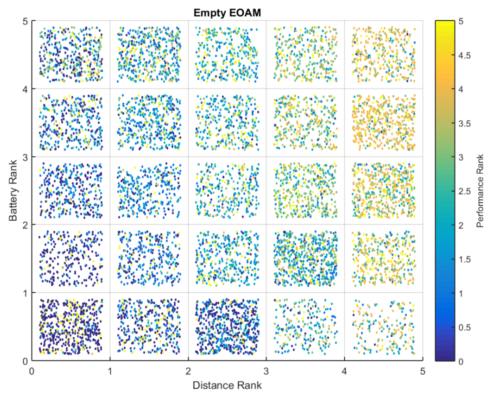


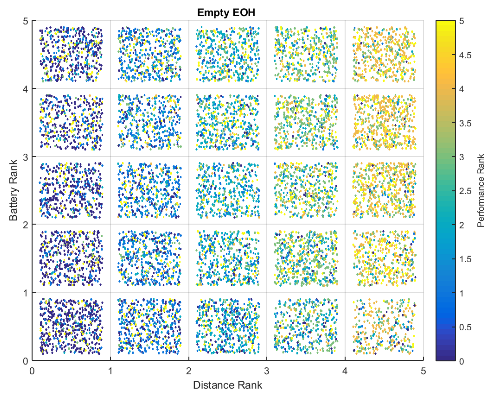

Empty HRAS
Battery Rank
Distance Rank
Performance Rank

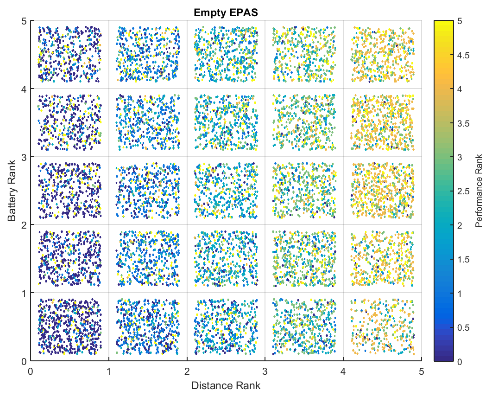

Empty HRAM
Battery Rank
Distance Rank
Performance Rank

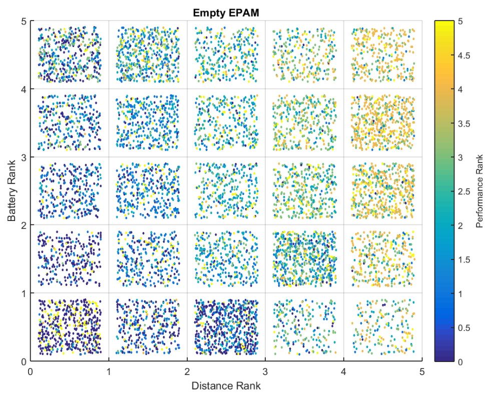

Empty HRH
Battery Rank
Distance Rank
Performance Rank

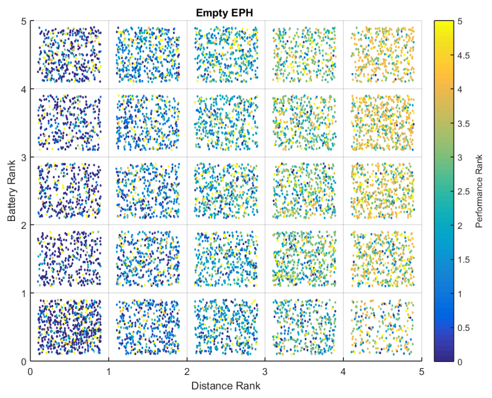

Empty EOAS
Battery Rank
Distance Rank
Performance Rank

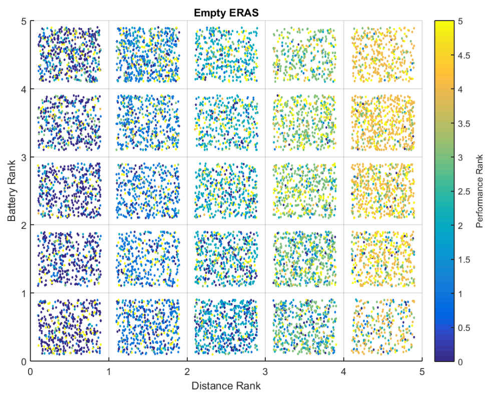

Empty EOAM
Battery Rank
Distance Rank
Performance Rank

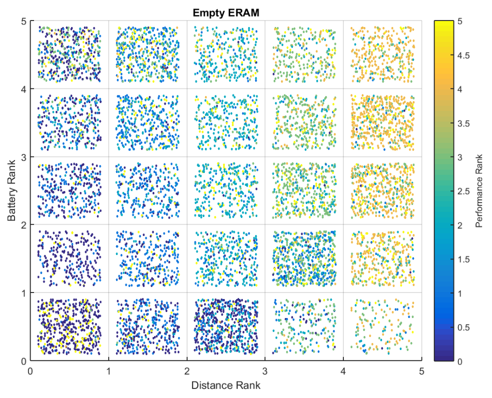

Empty EOH
Battery Rank
Distance Rank
Performance Rank

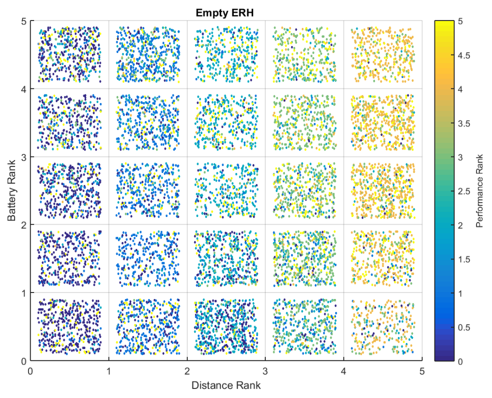

Empty EPAS
Battery Rank
Distance Rank
Performance Rank


Empty EPAM
Battery Rank
Distance Rank
Performance Rank


Empty EPH
Battery Rank
Distance Rank
Performance Rank


Empty ERAS
Battery Rank
Distance Rank
Performance Rank


Empty ERAM
Battery Rank
Distance Rank
Performance Rank


Empty ERH
Battery Rank
Distance Rank
Performance Rank

C.2 Cost Function Extraction Results

This section presents the cost functions found for each scenario and strategy pair.

Table C.21: Function parameters by strategy in an Empty scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
HOAS	0.0	−46.0	145.0	3324.0	16.0
HOAM	0.0	0.0	0.0	10.0	10.0
HOH	0.0	0.0	0.0	10.0	0.0
HPAS	0.0	0.0	0.0	10.0	0.0
HPAM	0.0	1011.0	80.0	587.0	−411.0
HPH	0.0	5466.0	−96.0	2791.0	298.0
HRAS	0.0	0.0	0.0	10.0	0.0
HRAM	0.0	0.0	0.0	10.0	10.0
HRH	351.0	0.0	−122.0	−4734.0	1254.0
EOAS	0.0	0.0	0.0	10.0	10.0
EOAM	0.0	0.0	0.0	10.0	10.0
EOH	0.0	0.0	0.0	10.0	10.0
EPAS	0.0	0.0	0.0	10.0	0.0
EPAM	0.0	10.0	0.0	10.0	0.0
EPH	0.0	0.0	0.0	1.0	0.0
ERAS	0.0	0.0	0.0	10.0	10.0
ERAM	0.0	0.0	0.0	10.0	10.0
ERH	0.0	0.0	0.0	10.0	10.0

Table C.22: Function parameters by strategy in an Office scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
HOAS	0.0	0.0	0.0	30.0	40.0
HOAM	0.0	0.0	0.0	10.0	10.0
HOH	0.0	0.0	0.0	10.0	10.0
HPAS	0.0	0.0	0.0	30.0	0.0
HPAM	0.0	0.0	10.0	0.0	30.0
HPH	−0.010	10.040	4.2012	59.557	−11.851
HRAS	0.0	0.0	0.0	10.0	10.0
HRAM	0.0	0.0	0.0	10.0	10.0
HRH	0.0	0.0	0.0	10.0	10.0
EOAS	0.0	0.0	0.0	10.0	10.0
EOAM	0.0	0.0	0.0	10.0	10.0
EOH	0.0	0.0	0.0	20.0	10.0
EPAS	0.0	0.0	10.0	30.0	0.0
EPAM	0.0	0.0	10.0	0.0	20.0
EPH	0.0	0.0	0.0	1.0	0.0
ERAS	0.0	0.0	0.0	10.0	10.0
ERAM	0.0	0.0	0.0	10.0	10.0
ERH	0.0	0.0	0.0	10.0	10.0

Table C.23: Function parameters by strategy in an Labyrinth scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
HOAS	0.0	30.0	50.0	0.0	0.0
HOAM	0.0	0.0	0.0	10.0	10.0
HOH	0.0	0.0	10.0	0.0	0.0
HPAS	0.0	0.0	0.0	10.0	0.0
HPAM	0.0	0.0	0.0	1.0	15.0
HPH	0.0	1271.0	−34.0	837.0	222.0
HRAS	44.0	648.0	191.0	−386.0	−3784.0
HRAM	0.0	0.0	0.0	10.0	10.0
HRH	0.0	0.0	0.0	10.0	50.0
EOAS	0.0	0.0	10.0	40.0	0.0
EOAM	0.0	0.0	0.0	10.0	10.0
EOH	0.0	0.0	0.0	10.0	0.0
EPAS	0.0	0.0	0.0	10.0	0.0
EPAM	0.0	10.0	0.0	10.0	0.0
EPH	0.0	0.0	0.0	1.0	0.0
ERAS	0.0	0.0	0.0	10.0	10.0
ERAM	0.0	0.0	0.0	10.0	10.0
ERH	0.0	0.0	0.0	10.0	20.0

Table C.24: Function parameters by strategy in an Randomized scenario.

Strategy	K_{d^2}	K_{b^2}	K_{db}	K_d	K_b
HOAS	0.0	0.0	0.0	10.0	0.0
HOAM	360.0	534.0	946.0	183.0	−33.0
HOH	0.0	0.0	0.0	10.0	0.0
HPAS	0.0	0.0	0.0	10.0	0.0
HPAM	0.0	10.0	0.0	10.0	0.0
HPH	−1.0	306.0	97.0	2521.0	244.0
HRAS	0.0	0.0	0.0	10.0	0.0
HRAM	0.0	0.0	0.0	10.0	10.0
HRH	0.0	0.0	0.0	10.0	0.0
EOAS	0.0	0.0	0.0	10.0	0.0
EOAM	0.0	0.0	0.0	10.0	10.0
EOH	0.0	0.0	0.0	10.0	10.0
EPAS	0.0	0.0	0.0	10.0	10.0
EPAM	0.0	10.0	0.0	10.0	0.0
EPH	0.0	0.0	0.0	10.0	0.0
ERAS	0.0	0.0	0.0	10.0	10.0
ERAM	0.0	0.0	0.0	10.0	10.0
ERH	0.0	0.0	0.0	10.0	10.0