Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2008

Pedro Alves

RoboCup Rescue: Development of Inteligent Cooperative Agents
RoboCup Rescue: Desenvolvimento de Agentes Cooperativos Inteligentes

"Sit down before fact like a little child, and be prepared to give up every preconceived notion, follow humbly wherever and to whatever abyss nature leads, or you shall learn nothing."

— T. H. Huxley

Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática,
2008

Pedro Alves

RoboCup Rescue: Development of Inteligent Cooperative Agents
RoboCup Rescue: Desenvolvimento de Agentes Cooperativos Inteligentes

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor José Nuno Panelas Nunes Lau, Professor Auxiliar da Universidade de Aveiro, e do Doutor Luís Paulo Gonçalves dos Reis, Professor Auxiliar do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto

o júri / the jury

presidente / president

Doutor António Rui de Oliveira e Silva Borges
Professor Associado da Universidade de Aveiro

vogais / examiners committee

Doutor Luis Miguel Parreira e Correia
Professor Associado do Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa

Doutor José Nuno Panelas Nunes Lau (Orientador)
Professor Auxiliar da Universidade de Aveiro

Doutor Luís Paulo Gonçalves dos Reis (Co-Orientador)
Professor Auxiliar do Departamento de Engenharia Informática da Faculdade de Engenharia da Universidade do Porto

agradecimentos / acknowledgements

English poet John Donne said 'No man is an island'. During the time it took to complete this thesis I've had the opportunity to see how true that statement is. Many people played a part in the path that took me here. Be it family and friends. Colleagues and professors. Even the rotten apples taught me valuable lessons. To all I thank you.

Nevertheless I would like to point out a few. To Professors Luís Paulo Reis and Nuno Lau, in particular to the last for his guidance, teaching, knowledge, and patience. To three friends of long working hours, and some good times in between, David, Luís and Picado. And last, but by no means the least, to Anabela. The first one to pick me up when I was down. My inexhaustible source of will power when I was running low.

Resumo

O trabalho desenvolvido nesta dissertação tem como tema o desenvolvimento de um agente inteligente com coordenação e comunicação no ambiente RoboCup Rescue. No RoboCup Rescue existem seis tipos de agentes, no entanto nesta tese só dois agentes foram desenvolvidos, especificamente o tipo de agentes Ambulâncias e Centros de Ambulâncias.

O tipo de agente Ambulância é o elemento responsável pelo salvamento de civis na cidade virtual que constitui o ambiente RoboCup Rescue. Para cumprir essa tarefa da forma mais eficiente possível conta com coordenação e comunicação com outros agentes do mesmo tipo, e com os Centros de Ambulâncias.

O comportamento da ambulância é modelado tanto para situações em que o Centro de Ambulâncias está presente durante a simulação, podendo, portanto, delegar funções para o Centro; como em situações em que o Centro não está presente, e, por isso, as ambulâncias estão encarregues de todo o processamento dos dados e de todas as tomadas de decisões.

As actividades desenvolvidas pelas ambulâncias podem ser resumidas a duas: pesquisa e salvamento. Para a primeira as soluções passam muito pelo uso de algoritmos estudados em Teoria de Grafos, já que a cidade virtual é, na sua essência, um grafo, e são necessárias soluções para problemas como visitar o mapa completamente e determinar o caminho mais rápido entre dois nós.

Na parte de salvamento a coordenação tem um grande papel a desempenhar. É necessário determinar que ambulâncias devem ir socorrer que civil, e quantas ambulâncias devem ajudar; ou que ambulâncias que devem continuar com a pesquisa do mapa. Ou seja, a coordenação é vital para uma utilização eficiente dos recursos disponíveis, e, consequentemente, uma boa pontuação.

Abstract

The work developed in this thesis has as background the development of an intelligent agent with coordination and communication in the environment of the RoboCup Rescue. RoboCup Rescue has six types of agents, however only two were developed in this thesis, specifically Ambulances and Ambulance Centers.

The type of agent Ambulance is the element responsable for the rescuing of civilians in the virtual city which comprises the environment of RoboCup Rescue. To fulfill this task in the most efficient way possible it relies on coordination and communication with other agents of the same type, as well as Ambulance Centers.

The behavior of an ambulance is modeled for situations when an Ambulance Center is available during the simulation, thus allowing the ambulances the possibility of dividing some of the processing and decision making; or, for situations when a center is not available and it is up to the ambulances to do make all of the decisions, and do all of the processing.

The activities performed by the ambulances can be summarized in two: search, and rescue. For the first, many of the solutions may be provided by algorithms studied in Graph Theory, since the virtual city is, in its essence, a graph, and its necessary solutions to problems such as visit the city entirely, and determine the shortest path between two locations, or nodes.

In the rescuing part, the coordination has a very big part to play. It is necessary to choose which ambulances should rescue a civilian, and how many should help doing it; or, which ambulances should continue searching the city for more civilians. In other words, coordination is vital for an efficient allocation of available resources, and, ultimately, a good score.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Objectives . . . 2
- 1.3 Thesis Outline . . . 3

2 RoboCup Rescue - Agent Simulation **5**
- 2.1 Rescue Simulator . . . 5
  - 2.1.1 Simulator Architecture . . . 5
  - 2.1.2 Kernel . . . 6
  - 2.1.3 World Objects . . . 7
  - 2.1.4 Viewer . . . 8
  - 2.1.5 Agents . . . 14
  - 2.1.6 GIS (Graphical Information System) . . . 18
  - 2.1.7 Simulator Modules . . . 18
  - 2.1.8 Communication . . . 20
- 2.2 Competition Rules . . . 20
- 2.3 Final Considerations . . . 21

3 Graph Theory **23**
- 3.1 Path Planning . . . 23
  - 3.1.1 Dijkstra Algorithm . . . 23
  - 3.1.2 A* Algorithm . . . 24
  - 3.1.3 D* Algorithm . . . 26
  - 3.1.4 Floyd-Warshall Algorithm . . . 28
- 3.2 Graph Traversing . . . 30
  - 3.2.1 Breadth First . . . 30
  - 3.2.2 Depth First . . . 32
  - 3.2.3 Related Algorithms . . . 33
- 3.3 Graph Partitioning . . . 34
- 3.4 Graph Partition Refinement . . . 35
  - 3.4.1 Kernighan-Lin . . . 35
  - 3.4.2 Ants Algorithm . . . 36
  - 3.4.3 Heuristic Method for Balanced Graph Partitioning . . . 38
- 3.5 Final Considerations . . . 40

4 Agent Interaction **41**
4.1 Cooperation . . . 41
4.1.1 Locker Room Agreement . . . 42
4.1.2 Situation Based Strategic Positioning . . . 45
4.2 Communication . . . 49
4.2.1 Language Components . . . 50
4.2.2 Types of Messages . . . 51
4.2.3 Communication Protocol Structure . . . 51
4.2.4 Speech Act Theory . . . 52
4.2.5 KQML - Knowledge and Query Manipulation Language . . . 52
4.2.6 KIF - Knowledge Interchange Format . . . 53
4.2.7 FIPA ACL - Agent Communication Language . . . 54
4.3 Final Considerations . . . 57

5 Rescue Agent Strategies **59**
5.1 Communication . . . 59
5.2 World Model . . . 63
5.3 Cooperation . . . 65
5.3.1 Search . . . 66
5.3.2 Rescue Planning . . . 68
5.4 Final Considerations . . . 73

6 Agent Development **75**
6.1 Agent Development Tools . . . 75
6.1.1 Agent Viewer . . . 75
6.2 Robocup Rescue Ambulance Team . . . 77
6.3 Robocup Rescue Ambulance Center . . . 78
6.4 Overall Strategy . . . 78
6.5 Single Agent Scenario . . . 79
6.5.1 World Model . . . 80
6.5.2 Search . . . 83
6.5.3 Rescue . . . 90
6.6 Multi Agent Scenario . . . 93
6.6.1 Subgraphs . . . 93
6.6.2 Communication . . . 107

7 Conclusion **109**
7.1 Future Work . . . 110

List of Figures

2.1 Simulator Architecture [14] . . . 6
2.2 Initial communication between simulator kernel and other components [14] . 7
2.3 Morimoto Viewer [37] . . . 8
2.4 Different States of a Civilian [12] . . . 9
2.5 RoboCup Rescue Simulator . . . 9
2.6 Freiburg Viewer [31] . . . 10
2.7 Freiburg Viewer showing statistics [31] . . . 11
2.8 FCPx chart comparing FCPortugal team with the Impossibles team regarding discovered civilians [12] . . . 12
2.9 FCPx table showing Firefighter related data [12] . . . 12
2.10 Facelifted Viewer [51] . . . 13
2.11 Overlapping signs showing number and status of the agents [51] . . . 13
2.12 Building border colored acording to its type [51] . . . 14
2.13 Ambulances during simulation [12] . . . 16
2.14 Policemen during simulation [12] . . . 16
2.15 Firefighters during simulation [12] . . . 17
2.16 Communication between agents [38] . . . 20

3.1 Shortest path from i to j where k is the highest intermediate vertex [15] . . . 29
3.2 The evolution of breadth-first search [55] . . . 31
3.3 The evolution of deapth-first search [56] . . . 32
3.4 A possible solution to partition a graph with the Kernighan-Lin heuristic [10] 36
3.5 Movement of an ant towards the worst local node [13] . . . 37

4.1 Agent architecture featuring locker-room agreements [48] . . . 43
4.2 Agent architecture featuring locker-room agreements [48] . . . 44
4.3 An example team strategy [44] . . . 46
4.4 Agent Architecture with SBSP mechanism [44] . . . 49

5.1 Communication method employed by the IUST team of 2006 . . . 60
5.2 Poseidon Agent Architecture [19] . . . 61
5.3 Poseidon Communication Structure [19] . . . 62
5.4 Sources of information for the World Model . . . 64
5.5 Centralised strategy used by the DAMAS team . . . 66
5.6 Ambulance agents rescue priority queue . . . 69
5.7 Behaviour of a GoldenKnight's Ambulance [28] . . . 71

5.8 Results comparison between greedy method and method using genetic algorithm [32] 73

6.1 Image of the Agent Viewer 76
6.2 Agent Viewer window identification 76
6.3 Image of a RCR map with its alleys in shades of grey 83
6.4 Example of traveling to a node with unvisited neighbors 85
6.5 Image of an agent with buildings in his viewing range 86
6.6 Area with unvisited nodes 90
6.7 States of an Ambulance agent 91
6.8 Location of Refuges in the Kobe map 92
6.9 Location of the Refuges in the Foligno map 93
6.10 Initial cuts for original ESP method 95
6.11 Total area calculated for Foligno map 96
6.12 Traditional calculation of Hough parameters 96
6.13 Example of calculating main directions 97
6.14 Example of a map divided into 9 partitions 97
6.15 Example of a division with no nodes 98
6.16 Example of partition expansion 99
6.17 Malformed partition 100
6.18 Foligno map divided in 4 partitions 101
6.19 Buildings close to each other 102
6.20 Buildings far from each other 102
6.21 Observation Points for the Kobe map 103

List of Tables

2.1 List of actions that the agents can perform [38] 15

4.1 Agent types vs Messaging abilities [26] 51
4.2 FIPA ACL Performatives [3] 55
4.3 FIPA ACL Performatives (cont.) [3] 56
4.4 FIPA ACL Message Parameters [2] 57

6.1 Number of agents available in a simulation [42] 80
6.2 World Model data and purpose 81
6.3 Results comparing average time required to visit each partition completely with each method 88
6.4 Results comparing average number of visited buildings during a simulation with neighbour building lookup and without 88
6.5 Results comparing average time required to visit each partition completely with direct entrance to buildings and without 89
6.6 Results evaluating the performance of Ambulances rescuing civilians 92
6.7 Results comparing average buildings visited with ESP and BOP 102
6.8 Results comparing average buildings visited with all partitioning methods . . 105
6.9 Number of buildings in each partition created with the different partitioning methods 106
6.10 Results comparing average search time between partitioned maps and unpartitioned maps 106

Chapter 1

Introduction

In this thesis we have a study of several strategies that can be applied to RoboCup Rescue agents, in particular Ambulance Teams, and Ambulance Centers. The task of an Ambulance Team is to search and rescue injured and/or trapped Civilians. The task of an Ambulance Center is to help coordinate Ambulance Team efforts, and communicate with other centers of other types of agents.

RoboCup Rescue is a division of the RoboCup initiative. "RoboCup offers an integrated research task covering the broad areas of AI and robotics, including real-time sensor fusion, reactive behavior, strategy acquisition, learning, real-time planning, multiagent systems, context recognition, vision, strategic decision making, motor control, and intelligent robot control." [29].

RoboCup has three competitions: Soccer, Rescue and Home. The first two divide themselves into simulation and robotics leagues. In the Soccer competition we have two teams of robots, or simulation agents, playing a game of soccer, with the obvious objective of scoring as many goals as possible. With Rescue, developers have a simulated environment of a city where a natural disaster has just occurred. The objective here is to save civilian lives, and extinguish fires that are raging in the virtual city. The ultimate aim of RoboCup Rescue "is to develop a series of technologies that can actually save people in the case of large-scale disasters and to actually operate such systems worldwide" [30].The last competition aims at developing autonomous robotic solutions that help taking care of everyday situations.

In both these competitions we have a Multi Agent System, or MAS, where coordination between teams of agents is essential to achieve the desired goal, since the teams of agents must work together if they are to be successful. An agent "is something that perceives and acts in an environment" [45]. In a MAS we have "autonomous agents who, while operating on local knowledge and possessing only limited abilities, are nonetheless capable of enacting the desired global behaviors." [53].

Although the soccer competition gathers more attention, Rescue also provides researchers with several challenging problems. The main concerns in the Rescue environment are: extinguish fires; rescue civilians and clear blocked roads. To each of these concerns we have a type of agent, Fire fighter, Ambulance, and Police, respectively. Each type has two subtypes, team and center. Teams of agents are field agents who do the actions required to fulfill the objectives of their respective type. Centers are auxiliary agents, responsible for aiding the field agents in what way they can. Whether its by doing some of the processing required, or by assigning tasks to field agents, or, as is imposed by RoboCup Rescue rules, to allow

communication between different types of agents. They cannot perform any other kind of action, and are immovable. Therefore, researchers, while developing Rescue agents, must take in consideration several aspects, such as decision making, team coordination, agent communication, task assignments, real-time planning, heterogeneity, logistic planning and incomplete information.

Even though a viable solution capable of deployment in a real life scenario may still be a long way from coming, developments in both the fields of AI and robotics give researchers confidence that in the long term, human rescue teams will have at their disposal intelligent technological resources which will help minimize human losses and property damage in the events of natural disasters, such has earthquakes, tornados, hurricanes and tsunamis. "Once accomplished, it [RoboCup Rescue] will be the one of the largest contributions that the AI and robotics communities can make for mankind." [30].

1.1 Motivation

On a personal note, Artificial Intelligence is, in the opinion of the author, one of the most interesting areas in Computer Sciences. That, coupled with the social aspects of RoboCup Rescue, and its growing success, made this thesis very much appealing.

Computers have made a huge impact in peoples lives. They allow the average user great working conditions, facilitate communication between different parts of the globe, allow for services to be quicker, cheaper, more accessible and requiring less paperwork, provide entertainment, and much more.

Nevertheless, computers can give us even more. With the ongoing research in the field of AI more and more uses arise for a computer. From an internet auction, to an opponent in a computer game, from a robot in a factory, to a robot used in space exploration, or, in the specific case of this thesis, a team of agents for search and rescue.

The work done in this thesis has the goal of helping to achieve a set of strategies that will one day give us several tools to aid humans in emergency situations such as earthquakes, tornados, tsunamis and volcanoes.

1.2 Objectives

The goal of this thesis is to further develop RoboCup Rescue agents, and develop new strategies that will lead to an efficient search and rescue plan, and that can be applied to agents other than ambulances.

Specifically, what is intended with this thesis is:

- Creation of an efficient method for map searching;
- Development of cooperative strategies for:

  Map searching using a team of undifferentiated agents;

  Saving civilians using several ambulance agents and undifferentiated agents.

1.3 Thesis Outline

The following chapter will describe the Robocup Rescue Project with some detail, including the Simulator System, and help illustrate the environment in which this work was done. The third chapter describes some of the algorithms and fields of study related with the development of strategies for the RCR agents, including the work developed for this thesis. Chapter 4, like chapter 3, will describe some work developed in the field of agent interaction, namely agent coordination and agent language. Moving on to chapter 5, several strategies of other teams of developers are presented. Chapter 6 will present the work implemented. The last chapter will give the reader some results, conclusions and future work.

Chapter 2

RoboCup Rescue - Agent Simulation

The work developed in this thesis is included in the RoboCup Rescue Agent Simulation Competition. This competition gathers participants from different parts of the world. Every year the teams develop new strategies, or upgrade previous implemented ones with the goal of having the most efficient agents that will perform the top scoring simulations.

2.1 Rescue Simulator

To create the conditions required for the development of intelligent agents capable of acting in disaster situations a simulator was necessary. The simulator provides a city that has just suffered an earthquake. The city's buildings will crumble, and its citizens will perish unless swift, decisive and efficient action takes place.

2.1.1 Simulator Architecture

The architecture of the simulation system is modular. It is based in separate components, each with its function, and they all communicate using UDP messages [14]. Figure 2.1 shows the different components and how they are connected.

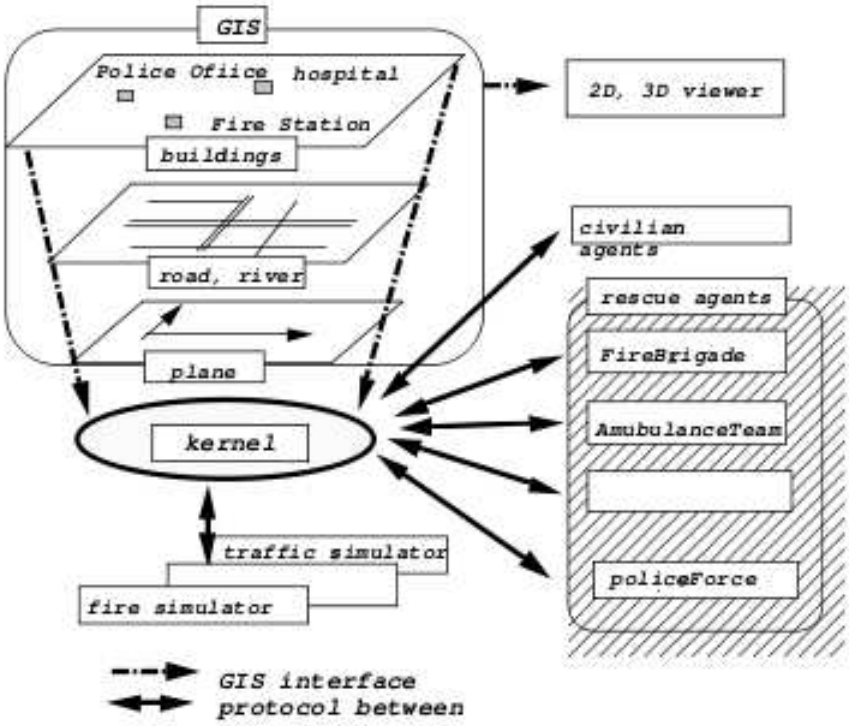

Figure 2.1: Simulator Architecture [14]

The modules present in the simulator architecture are: kernel, GIS (Geographical Information System), agents, component simulators. This architecture presents a great flexibility. By default, a module should represent a process. However, it is possible to run several modules in only one process. The communication between modules also follows certain guidelines. With exception for the agent modules, other modules may communicate directly, although it is preferable that the communication passes through the simulator kernel.

2.1.2 Kernel

Being the central point in the simulator architecture, the kernel plays a great part in the simulation process. It controls the simulation and acts as a proxy for remaining modules. Currently the number of modules is limited, but in the future this will change. The number of agents and modules will increase as the demand for more realistic simulations also increases. Obviously, the robustness of the kernel will have to accompany this growth.

Figure 2.2 displays the communication that occurs at the start of a simulation. Notice that GIS is able to communicate with the viewer without passing through the kernel as was referenced in section 2.1.1.

Figure 2.2 demonstrates that when the simulation begins the kernel receives data from the GIS regarding the initial world configuration. During the rest of the simulation other exchanges of information happen. For example, at each cycle the kernel sends sensorial data to the agents and receives the agents' actions. After validating them (ex.: with a MOVE command check if a path is correctly formed) they will be executed [49], and the consequences of those actions will be evaluated by the respective module simulators.

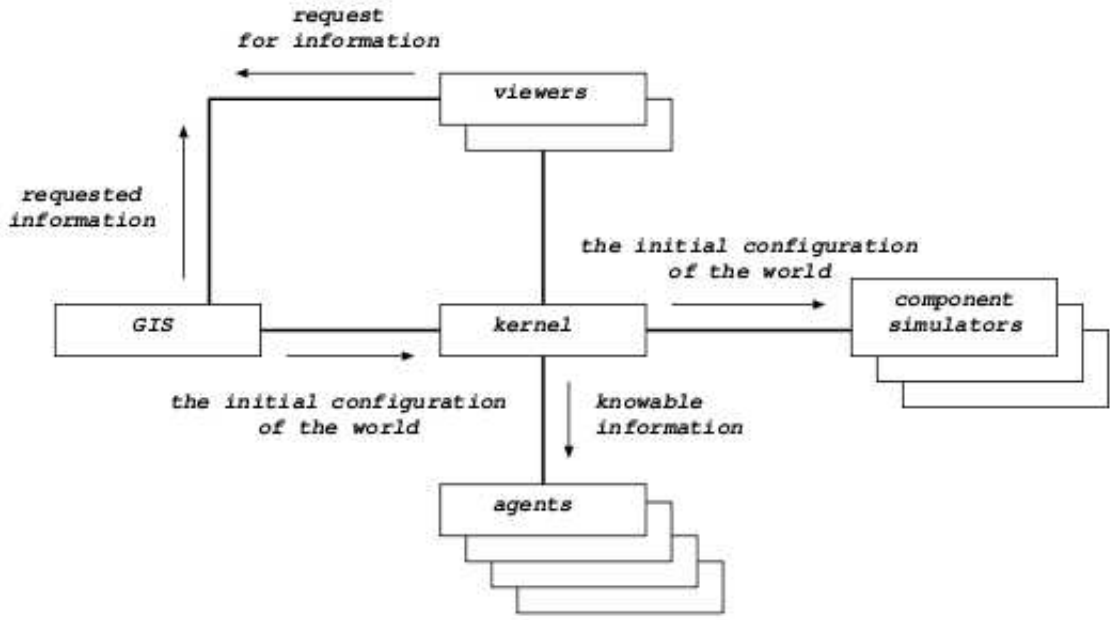

Figure 2.2: Initial communication between simulator kernel and other components [14]

2.1.3 World Objects

The scenario for the simulation, a virtual city, is composed of several objects. All buildings, roads, civilians and remaining agents are mapped to distinct objects. And to the kernel the world is just composed of a set of objects with different properties. To identify clearly each object, a unique ID is assigned to each of them.

2.1.3.1 Road

The map of the city is represented as a graph where the roads are the edges of that graph. Besides the characteristic properties of edges of a graph, a Road has all the necessary properties that allow to determine the size of the road (width, length) and others that limit in some way the crossing of the road by the agents (number of lanes in each direction, lane width for walkers). These last properties, and the number of agents crossing the road at a given time will tell the simulator if there is a traffic jam at that road.

2.1.3.2 Node

The nodes are crossings on the map, and vertexes in the graph. Like the Roads, this object also has the usual properties of a graph vertex, plus properties that resemble that of an actual road crossing (existence of signals, and others connected to turns).

2.1.3.3 River and River Node

The river system would be another representation of a graph. However, the current version of the simulator (0.49+) still does not implement these objects.

2.1.3.4 Building

A building is exactly what its name implies. It is where civilians will be trapped, and where fires will begin to burn. Its properties define the shape, size, number of floors, position, and type of material that composes the building (wood, steel, concrete). A building may also have several entrances. Those will be the map's nodes which have a connection to the building.

2.1.3.5 Refuge

A Refuge is a special building where civilians go to be safe and firefighters go to refill their water tanks. It has all the properties of a building, but it is indestructible.

2.1.4 Viewer

2.1.4.1 Morimoto Viewer

For the RCR developers to be able to get a empyric feedback several viewers were devoloped. With them it was possible to see the agents' behavior and represent the objects listed in section 2.1.3. There are three viewers that rank amongst the most used ones. The most widely used is the Morimoto Viewer. Figure 2.3 displays an example of a simulation using the Morimoto Viewer.

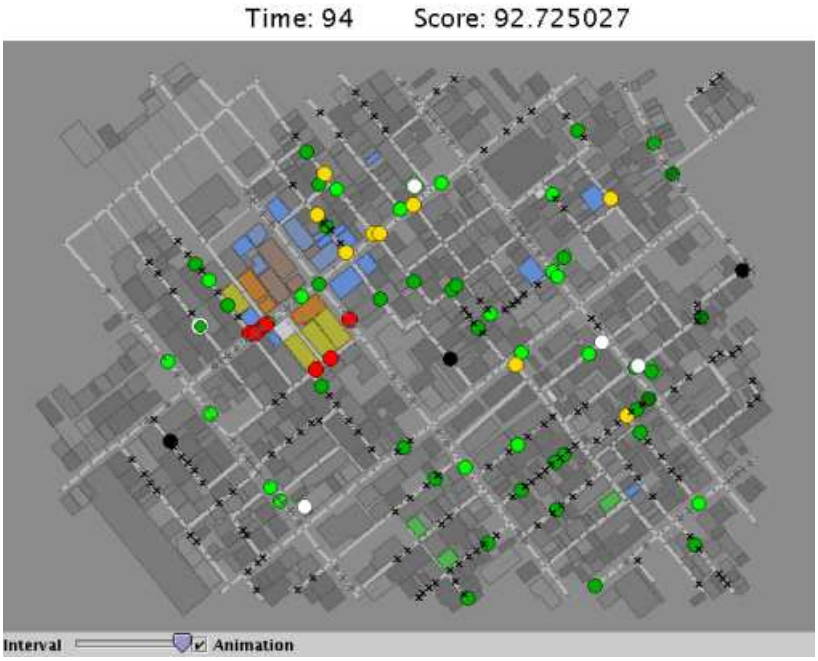

Figure 2.3: Morimoto Viewer [37]

This viewer draws the agents with different colors for them to be distinguishable. Ambulances are painted in white, Policemen in blue, Firefighters in red and civilian in green. To reflect the agents' Health Points (HP) the color in which the agent is painted becomes darker,

as its HP decreases. Figure 2.4 shows the several colours that a civilian assumes, according with its remaining HP.

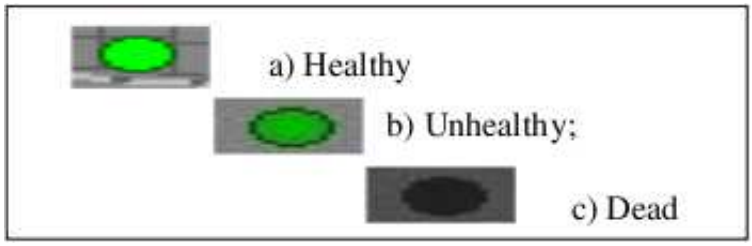

Figure 2.4: Different States of a Civilian [12]

Buildings also have different colors to tell them appart and to know their current status. A Refuge is painted green, and Centers are painted in white. All remaining buildings are painted in shades of grey, according to how much they collapsed at the start of the simulation. The darker the building, the more collapsed it is. In case there is a fire, a building will see its color evolve from yellow, to orange, and red. If a building is completely burned down it will be painted in dark grey. Should the building be extinguished or flooded it will be painted in tones of blue.

Roadblocks are represented by crosses on the roads. Like the agents and the buildings, the darker the cross, the greater the obstacle.

2.1.4.2 Extended Viewer

Figure 2.5 depicts another RCR viewer. This one has the particularity of providing detailed information regarding a selected object, and also some added features. Those features made this viewer the most used while working on this thesis as they simplified the correction of bugs.

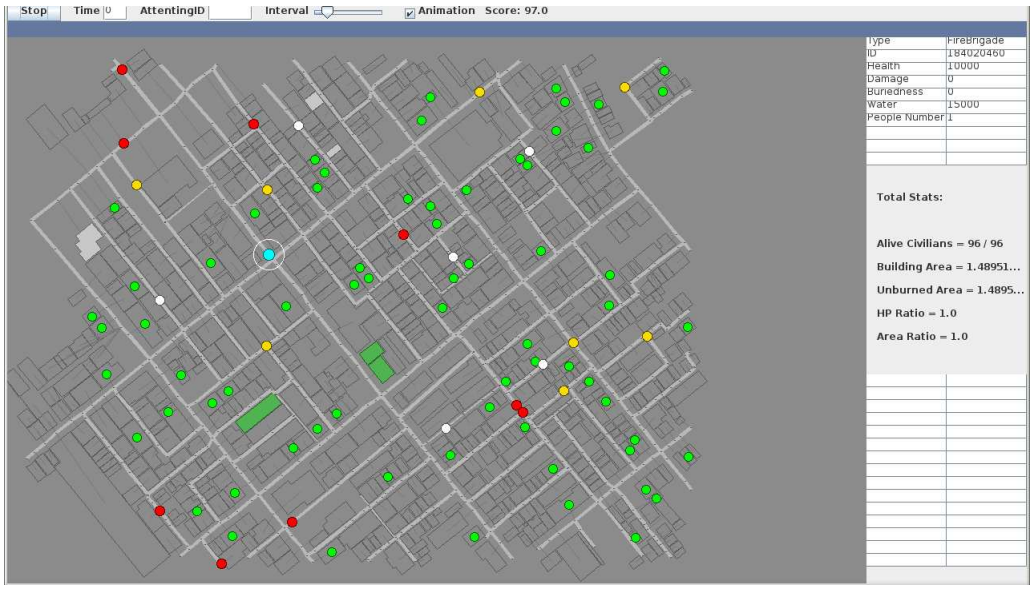

Figure 2.5: RoboCup Rescue Simulator

The map itself is the same as the Morimoto Viewer, however, on the right of the viewer there is a table that is filled with data of the selected object, such as its type, ID, HP, Damage and Buriedness (for humanoid agents), Area and Number of floors (for buildings), among others. When an agent is selected, a circle is drawn around it. That circle represents the agent's viewing range. Selecting an agent also causes the bottom right table to be filled with data of other agents that are at the same location as the selected agent. There are also some commands that in the upper bar like the Stop button, that freezes the image of the viewer, while the simulation continues. And also the AttendingId textbox that allows the user to write an object Id which will be selected, and its data presented in the aforementioned table. Finaly there is a checkbox that allows the user to stop the animations, making the simulation lighter, processing wise.

2.1.4.3 Freiburg 3D Viewer

Four years after the RCR was added to the RoboCup competition, Alexander Kleiner and Moritz Gobelbecker of the Freiburg University, decided to create a viewer that would try to diminish two limitations identified by the authors. Those limitations where the absence of more detailed results regarding a simulation, and lack of appeal to the general public [31].

This viewer presents an improved viewing experience, where even the agents' representation met a more recognizable form. Firefighters are now represented by firetrucks while on roads, and by fire fighter helmets while inside buildings, Ambulances by actual ambulances, and Policemen by police cars. Another improvement was to the buildings on fire. The Freiburg Viewer displays a burning building with flames and smoke coming out of it. Figure 2.6 is an example of the Freiburg Viewer with the Kobe map.

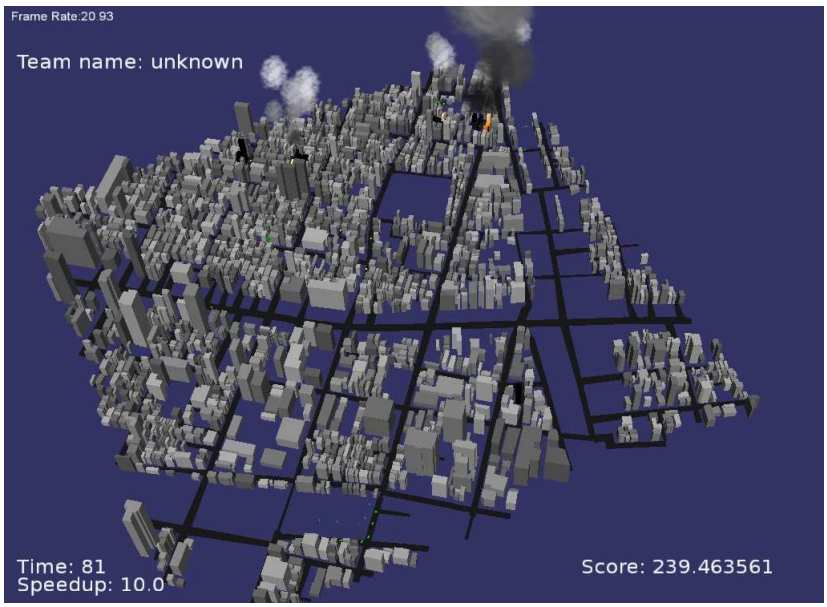

Figure 2.6: Freiburg Viewer [31]

To address the simulation results issue the viewer is able to present statistics that allow developers to know where should they improve. The statistics displayed by the viewer include: percentage of blocked agents over time (for police forces); percentage of dead agents over time

(for ambulance teams); percentage of destroyed buildings over time (for firefighter teams). And also more generic statistics: percentage of found civilians over time; how many times did agents run into blockades [31].

Figure 2.7 shows an example of statistics provided by the viewer during a simulation.

Figure 2.7: Freiburg Viewer showing statistics [31]

FCPx - A Tool for Agent Evaluation and Comparison

The portuguese team FC Portugal developed a tool to help evaluate the agent performance and compare Rescue teams, the FCPortugal eXtended Freiburg 3D viewer (FCPx) [36]. This tool extends the analysis features of the Freiburg 3D viewer to help objectively measure agent performance and allow team comparison. This tool extracts as much data as possible from the log file and extrapolates further information from the log file. It then saves all data into another file where the information is kept in a format that allows easy importation into a spreadsheet where it becomes simple to create tables and graphics. Figures 2.8 and 2.9 show two different representations that can be built from the data collected by FCPx.

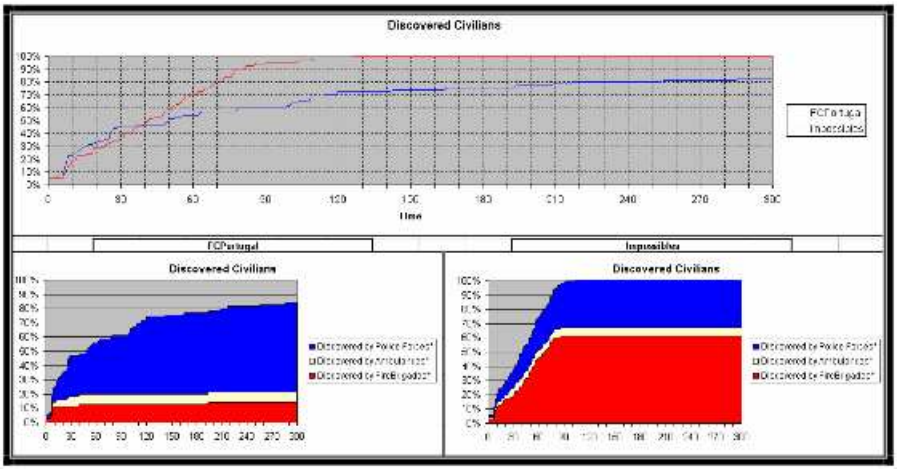

Figure 2.8: FCPx chart comparing FCPortugal team with the Impossibles team regarding discovered civilians [12]

	Agent Related Data				
	Legend				
	Buried Police Forces Killed by Fire			FireBrigades Killed by Fire (Not Buried)	
	Buried Ambulances Killed by Fire			Number of Buried FireBrigades (Alive)	Total Number of FireBrigades
	Buried FireBrigades Killed by Fire				13
	Buried Agents Killed by Fire				
Time	Police	Ambulance	FireBrigade	FireBrigades Killed by Fire (Not Buried)	No. Buried FireBrigades (Alive)
0	0	0	0	0	N/A
1	0	0	0	0	N/A
2	0	0	0	0	2
3	0	0	0	0	2
4	0	0	0	0	2
5	0	0	0	0	2
6	0	0	0	0	2
7	0	0	0	0	2
8	0	0	0	0	2
9	0	0	0	0	2
10	0	0	0	0	2
20	0	0	0	0	2
30	0	0	0	0	1
40	0	0	0	0	1
50	0	0	0	0	1
60	0	0	0	0	1
70	0	0	0	0	1
80	0	0	0	0	1
90	0	0	0	0	1
100	0	0	0	0	1
110	0	0	0	0	1
120	0	0	0	0	1
130	0	0	0	0	1
140	0	0	0	0	1
150	0	0	0	0	1
160	0	0	0	0	1
170	0	0	0	0	1
180	0	0	0	0	1
190	0	0	0	0	1
200	0	0	0	2	1
210	0	0	0	2	1
220	0	0	0	2	1
230	0	0	0	2	1
240	0	0	0	2	1
250	0	0	0	2	1
260	0	0	0	2	1
270	0	0	0	2	1
280	0	0	0	2	1
290	0	0	0	2	1
300	0	0	0	2	1

Figure 2.9: FCPx table showing Firefighter related data [12]

2.1.4.4 Facelifted Viewer

In 2005 another viewer was released with the goal of providing a more appealing interface while also adding extra features that would help the analysis of the agents behavior [51]. Figure 2.10 displays the viewer that was developed by Masafumi Ueda, that was named by the author a "Facelifted Viewer".

Figure 2.10: Facelifted Viewer [51]

This viewer has many visual upgrades that not only were more visually attractive, but also provided more information about the running simulation, such as overlapping signs, enabling anyone watching the simulation to tell how many agents are at a given point. Figure 2.11 displays two examples of overlapping signs.

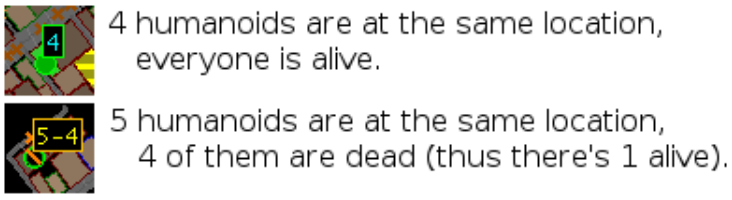

Figure 2.11: Overlapping signs showing number and status of the agents [51]

Another feature which can be of great use when studying how fire spreads in the virtual city is the Building Code indicator. This feature colors the borders of the buildings according to their type (wood, steel, concrete) as can be seen in figure 2.12.

Figure 2.12: Building border colored acording to its type [51]

Other features included in this viewer are:

- High-contrast colorings and fire pattern animation - coloration of buildings according to their status; animation of burning buildings
- Trap signs - when an agent is trapped a backslash appears above the agent indicating its situation
- Death signs - a square with bold borders appears around an agent when it dies
- Score graph - displays time vs score
- Building status proportion graph
- Humanoid health status proportion graph - time vs. proportion of humanoids' health
- Blinking action signs - animation of water and humanoid agents
- Auto frameskip - skipping of frames to reduce load of processing
- Sight Range Circles - a circle is drawn around the agent delimitting its view range
- Zooming and Scrolling - ability to zoom in to a specific area
- Selection of Viewable Objects - when runing the simulation it is possible to select what are the viewable agents; useful for when it is preferable to see the behavior of only one type of agent, for example

2.1.5 Agents

To reach the goal of saving as many lives, extiguish as many fires as possible and reduce property damage to a minimum, different sets of field agents are available, with different functions. Ambulances rescue civilians from collapsed buildings and carry them to refuges, firefighters extinguish fires, and police teams clear roadblocks so that other agents may move freely.

All these agents are part of the RoboCup Rescue (RCR) Agents. These divide into platoon agents and center agents. The aforementioned agents are platoon agents. The center agents are the Ambulance Center, Fire Station, Police Station.

Common to field agents are also some of their limitations such as speed and the ability to suffer damage. An agent can travel at the maximum speed of 20 km/h which means that an agent can travel no more than 333 m in a single cycle. A civilian, on the other hand, moves at a speed no greater than 3 km/h [38]. An agent may get hurt and lose HealthPoints (HP). Although it too, can be rescued, an agent's behavior should reflect this caracteristic and avoid as much as possible situations where it may get hurt.

Performing the tasks of each field agent may be done single-handedly or in groups. Two Ambulances dig up a civilian faster than one, for example. The time required to perform a task does not depend only on the number of agents doing the task. The degree of buriedness of a civilian, or the degree of fieryness of a building also dictate the time required for an ambulance and firefighter, respectively, to complete its given task.

The last agent is the Civilian agent. This agent is the target of rescue by the Ambulance agents.

Each agent has a specific set of capabilities. Table 2.1.5 depicts the actions available for each type of agent.

Table 2.1: List of actions that the agents can perform [38]

Type	Capabilities
Civilian	Sense, Hear, Tell, Move
Ambulance Team	Sense, Hear, Say, Tell, Move, Rescue, Load, Unload
Firefighter	Sense, Hear, Say, Tell, Move, Extinguish
Policemen	Sense, Hear, Say, Tell, Move, Clear
Ambulance Center	Sense, Hear, Say, Tell
Fire Station	Sense, Hear, Say, Tell
Police Office	Sense, Hear, Say, Tell

2.1.5.1 Ambulances

As was stated before, Ambulances have the prime objective of rescuing civilians. After identifying a trapped civilian they load the Civilian Agent (action Load) and carry the civilian to a refuge, where they unload the civilian (action Unload) and thus save its life. If the civilian is buried, the ambulance must first unbury the civilian (action Rescue). Figure 2.13 shows Ambulances in action. They are the white circles.

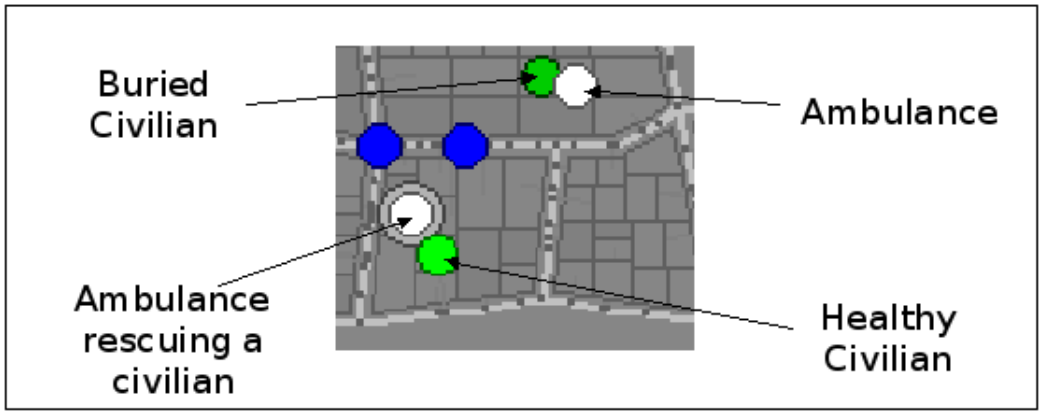

Figure 2.13: Ambulances during simulation [12]

Associated Challenges:

- Rescue scheduling
- Buried agents life time prediction
- Ambulance coordination
- Coordination with agents of different type

2.1.5.2 Policemen

Altough this agent has a task of its own, much of its work (clearing roadblocks of any kind) is done in cooperation with other types of agent. More important than clearing a road, is clearing a road that is blocking an ambulance. This means that communication is vital for a policeman to execute a good and efficient work. In figure 2.14 there are two policemen (blue circles) in action. The crosses around them are roadblocks.

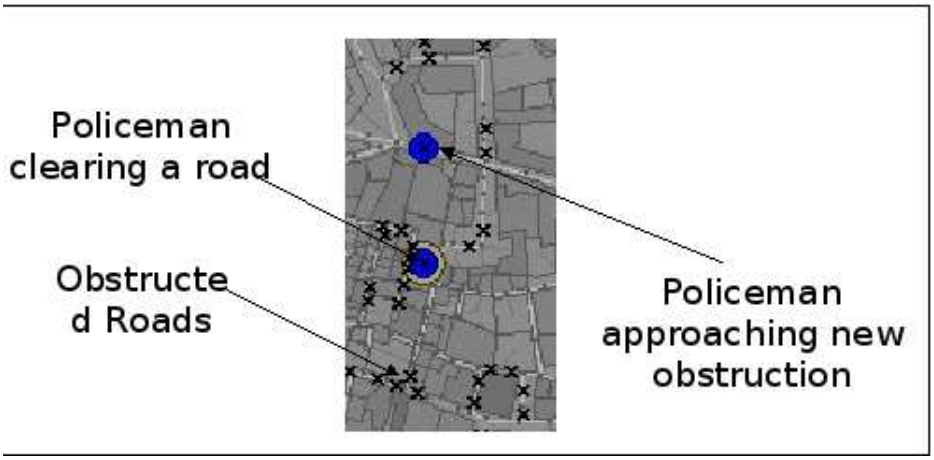

Figure 2.14: Policemen during simulation [12]

Associated Challenges:

- Schedulling of the roads to clear considering: trapped emergency vehicles, principal routes, paths to refuges, fires and trapped civilians
- Coordination between police forces
- Coordination with agents of different types

2.1.5.3 Firefighters

Like the previous agents Firefighters have a crucial role in the outcome of the simulation. Unlike the previous agents, the Firefighter's actions may have repercussions in two aspects: damaged property and civilian lives. When deciding which burning buildings to try to extinguish the agent must consider the damage being done to property and also if civilians are trapped in the burning building, or a surrounding building, and the fire status.

After deciding which fire to extinguish, the agents must decide how will they proceed. They must determine the best placement to extinguish the fire, and if they should attempt to put out the fire, or try to prevent it from spreading by flooding the surrounding buildings.

Figure 2.15 shows two Firefighters doing their work. One of them is already putting out a fire, while the other is moving to a position where it can help its teammate.

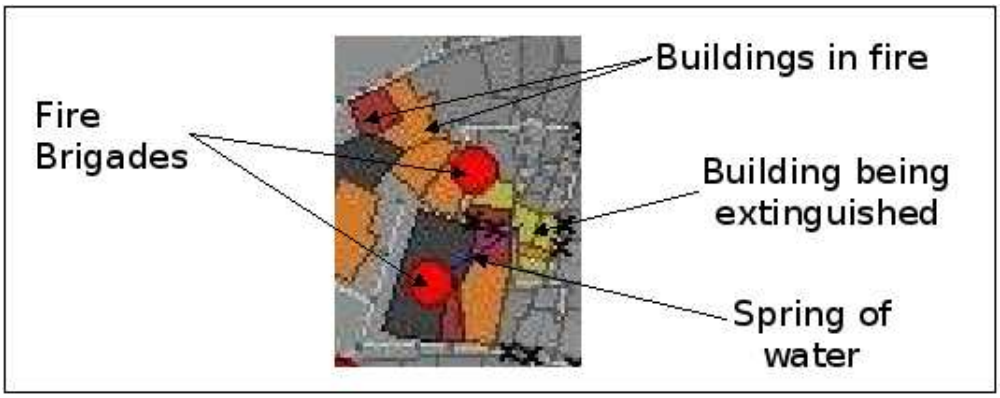

Figure 2.15: Firefighters during simulation [12]

Associated Challenges:

- Choose the best region to extinguish
- Choose the best building
- Anticipate fire spreading (buildings and human lives)
- Collective and individual management of water in tanks
- Firefighters coordination
- Coordination with agents of different types

2.1.5.4 Centers

RCR agents also include centers for the different types of agents (Ambulance, Police and Firefighters). These centers can act as a central unit responsible for coordination for its respective type of field agents, and they can act as proxies to communicate with other types of agents. In fact, one of the rules in RCR Agent Competition is that communication between different types of agents is only possible through each type's respective centers, while communication between agents of the same type is limited only by the number of messages that an agent is able to receive in each cycle.

The field agents are representations of humanoid agents, while centers are buildings. But like refuges, these buildings do not burn or collapse.

Associated Challenges:

- Use global vision to improve high-level decisions
- Communication limitation management
- Coordination with agents of different types

2.1.5.5 Civilians

Finally, RCR also counts civilians as one of its agents. But this is a limited agent. It's only actions are to travel to the closest Refuge, or shout for help in case of being hurt, or trapped. In figure 2.13 there are two trapped Civilians (the green circles).

This agent is also refered to as Car. This is because while traveling to the refuge the Civilian acts as a motorized agent.

The Civilian is part of the RCR simulator, and this agent is not developed by rescue teams.

2.1.6 GIS (Graphical Information System)

The Graphical Information System module, or GIS, provides the initial configuration of all objects of the RCR World, including roads, buildings, and agents of all kinds to all of the Simulator Modules of the kernel system. It also logs every action and evolution of the RCR world during a simulation thus allowing to replay the simulation using a logviewer. As was mentioned in section 2.1.2 the GIS also sends data for the Viewer [49].

2.1.7 Simulator Modules

To respect the modular architecture of the simulator, and to maintain its flexibility, the several aspects of the simulations are managed by individual simulators. Some of the following simulators were added as the RCR simulator evolved, and others were made more complex.

2.1.7.1 Collapse Simulator

This module is responsible for the physical state of the buildings in the virtual city. It it triggered only at the begining of the simulation and it simulates a collapse of 80% to 90% of all buildings, excluding refuges and centers. The degree of damage suffered by buildings varies from building to building.

2.1.7.2 Blockade Simulator

This simulator manages obstacles in the roads of the virtual city. After the disaster simulated in the RCR world some of the roads are blocked, completely disabling some paths, and trapping some of the agents. The causes of the obstacles are many. Blockades can be made by debris for collapsed buildings, traffic accidents or crowds, but its representation is allways the same, a cross on the road as displayed in figure 2.14.

Like buildings, roads also have different levels of damage. A roadblock may block a road completely or partially. When partially blocked it may affect the speed at which the agents pass through the block, or limit the number of lanes of the roads and possibly creating traffic jams when multiple agents attempt to pass through the block simultaneously. The effects of the roadblock are determined by its size which is measured in millimeters.

2.1.7.3 Traffic Simulator

The movement of the agents in the RCR world is controled by this module. It calculates the speed at which an agent can travel through a road using parameters such as the width of the road, number of agents on the same road, and existence and size of a roadblock.

2.1.7.4 Fire Simulator

The Fire Simulator is one of the most complex simulators that compose the simulator modules, if not the most complex. It is responsible for the simulation of fires inside buildings. To further increase the complexity and realism of the simulator some buildings begin to burn right after the earthquake that initiates the simulation. The speed at which a building burns and the way a fire spreads are calculated using an intelligent model. That model will increase the temperature of a building according to its own combustion and/or heat waves from neighbouring buildings. The decrease of the building temperature will only happen if water is being pumped by firefighters. To simulate the combustion of buildings, a model is used where the main factors are temperature and fuel (provided by the building). Each building has its own ignition point and when the temperature reaches this point then the building will start to burn. Reversely, should the building temperature drop below the ignition point, the fire will be considered extinguished [39]. In figure 2.15 we see some burning buildings with firefighters trying to extinguish the fires in the buildings.

2.1.7.5 Miscellaneous Simulator

After a discussion that began in March, 2000, and after kernel version 0.23, this module (miscsimulator) was included to manage the agents' status. For the duration of the simulation it defines the values of buriedness, and damage for civilian agents and other agents alike. To calculate the HP for a Civilian the miscsimulator uses the following equation:

$$hp = damage \times T + 10000$$

where *damage* is equal to -100 in case the building has collapsed, or -1000 if the building is on fire, and T is the elapsed time. Should a civilian be transported to a Refuge *damage* is set to 0.

2.1.8 Communication

An important aspect of the development of RCR agents is the communication performed between the agents. That communication is limited by rules and other constraints. What the agent perceives as sensorial information is transmited by the kernel to the agent. That information may include visual and audio information.

Visual data is sent by the kernel to platoon agents. Centers do not receive any visual data, which is only natural, since they are buildings. The visual data is bound by the viewing range of the agents, which is 10 meters.

There are two types of audio communication: radio and natural voice. Both types of communication have information regarding the sender, and the contents of the message. Natural voice is sent by the speaker and can only be listened to in a range of 30 meters. All types of agent can hear this, except Centers.

Radio communication is not hindered by distance, and all agents, including centers, can listen to radio messages. However, radio communication has other types of limitations. A platoon agent can only listen to radio messages from the center of its type and other platoon agents of the same type. A Center can communicate only with platoon agents of its type and with Centers of any type. For a field agent to communicate with other types of agents via radio, it has to do so using the Center as a proxy. Another limitation of radio communication is the number of messages that an agent can read in a single cycle. An agent can't read more than four messages per cycle, excluding its own messages. A Center, however, is allowed to read $2 \times n$ messages, where n is the number of field agents of the same type. For example, with 5 ambulances, an Ambulance Center can receive 10 messages per cycle.

Figure 2.16 displays how communication is performed between agents.

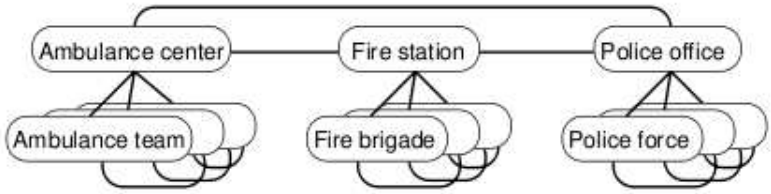

Figure 2.16: Communication between agents [38]

In previous versions of the kernel each type of communication had a type of command associated with it. Currently all audio communication is done using the AK_TELL command. For the agents to be able to tell the difference between a voice message and a radio message, channel 0 has been reserved for voice audio since kernel version 0.49+.

The exchange of messages was initially performed using UDP. However, since 2005 and with simulator version 0.47, the exchange of messages now uses TCP/IP.

2.2 Competition Rules

Each competition has specific rules of its own, regarding the kernel version to use, communication restrictions, alterations on module simulators, maps used for the simulations, numbers of available agents and ignition points, simulation parameters, scoring function and penalisations [42]. Some of these rules change very little between competitions. According to

the Rescue Simulation League Rules for the 2008 edition of Robocup [42], calculations for a simulation score are as follows:

$$V = (P + \frac{S}{Sint})\sqrt{\frac{B}{Bint}}$$

Where:

- P - number of living agents
- Sint - total Health Points (HP) of all agents at start,
- S - remaing HP of all agents,
- Bint - total area at start,
- B - area of houses that are undestroyed,

The following rule was used to compute the value B:

- 0 - no penalty
- 1,5 - 1/3 of area counted as destroyed
- 4 - water damage, also 1/3 area counted as destroyed
- 2,6 - 2/3 of area considered as destroyed
- 3,7,8 - whole building considered as destroyed

The greater the value of V the better the score. The scoring system is used for the competition results, but is can also be helpful while developing the agents as it gives a global idea of the effectiviness of their agents.

The format of the competition is constituted of three rounds of preliminaries, followed by a round of semi-finals, and the final. The number of teams that begin the competition may vary. In 2007 16 teams qualified for the preliminaries [4], while in 2008 17 teams also reached the preliminaries [5]. Of the qualified teams, the top 8 will pass to the semi-finals, where only 4 will pass.

2.3 Final Considerations

The simulator is currently a complex piece of software. Nevertheless there are many improvements to be made in order for the real world to be emulated with greater precision. The collapse simulator is expecting an upgrade that will add the feature of aftershocks, which would make the world even more dynamic by damaging more buildings, with possible consequences to the civilians trapped in those buildings, and creating new road blockades. The miscsimulator can also be altered to make the prediction of civilian death time more complex, or even more easy. For example, Civilians could be modified to have more properties, like age or physical condition, that would allow a more acurate or facilitaded estimate of their death time. There are also some imbalances in the existing simulators, caused by evolving on or

two simulators, in detriment of others. A good example of this is the fire simulator, that has evolved much more than the remaining simulator modules. And even the fire simulator could be improved.

Another aspect to take into consideration is that although it is more appealing to have simulators that mimic the real world with great precision, there are also advantages in taking it step by step. The accuracy of the simulator is directly proportional to its complexity. And the more complex the virtual world is, the harder the development of the agents is. This means that gradually escalading the complexity also means a gradual evolution of the RCR agents.

Chapter 3

Graph Theory

The city map in a RCR smiluation is represented as a graph, as was explained in sections 2.1.3.2 and 2.1.3.1. As such it is impossible, and unadvisable, to avoid the use of Graph Theory methods for search and world modeling. The main fields of Graph Theory used by RCR teams are Path Planning, Graph Traversing, and Graph Partitioning. In this chapter it is presented a brief description of those fields and of some methods developed in them.

Before beggining to describe methods that use graphs, it is best to define what is a graph. A graph is composed of a set of vertices, V, and another of edges, E, and is commonly represented as $G = (V, E)$. The edges connect two vertices (the head and tail) and normally have a cost associated to them that can represent the distance separating the two vertices. Depending on what the graph represents the vertices may also have a cost associated. The edges in E also have other characteristics, like direction. An edge e, linking vertices v_1 and v_2 may allow travelling from v_1 to v_2, but not the other way. A graph with such characteristics is called a *Directed Graph*. If the edges allow travelling in both directions then the graph is called *Undirected Graph* or *Bidirectional Graph*. Again, just like the costs, the directions of the edges depend on what the graph represents.

3.1 Path Planning

There are several and distinct situations in which an agent needs to determine a path. The main difference that may exist between those situations is if the target destination is, or not, known. When the destination is known, we call it Path Planning. In the opposite case it is considered a Search.

Path planning methods are used when it is needed to go from one place to another, while travelling the shortest distance possible, or using the path with the smallest cost. A path may be represented by a set of n vertices, P, composing the path, $P = \{v_1, v_2, ...v_n\}$, where $|P|$ represents the cost, or weight, of the path. The following algorithms aim at providing a path P where $|P|$ is as small as possible.

3.1.1 Dijkstra Algorithm

While searching the algorithm used is the Dijkstra Algorithm [15], a method widely implemented by several teams and with uses in many other fields that require search or the calculation of a short path. This method was used not only because of its reputed speed and

easy implementation, but also because while searching for a node in the graph starting from another point, if there is a path between those nodes, it will find and return the shortest path. For a graph $G = (V, E)$ with $|V|^2$ edges the Dijkstra algorithm runs in $O(|V|^2)$ time. For graphs with less edges, the algorithm sees its time improve to $O(|E| + |V|log|V|)$.

The algorithm itself can only be used with non-negative edge weights. For such cases there are other algorithms. It starts by initializing a set of vertices, V_{aux}, which will be updated with the new weights as the shortest paths progress. Afterwards it will cycle trough a list, V, containing all vertices of the graph, G, until that list is empty. With each iteration, the node with the shortest distance is removed from the list. The node will then be added to the path and its neighbours will be relaxed. The relaxing of the neighbours implies the update of its weight, calculated from the node that was just added to the path, unless that would result in the increase of the node's distance. The following pseudo-code ilustrates a possible implementation of the algorithm.

```
DijkstraSearch(G, source) {
    for (each vertex v in G) {
        distance[v] = infinity
        previous[v] = undefined
    }
    distance[source] = 0
    Vaux = V
    while (Vaux is not empty) {
        u = extract-min-distance(Vaux)
        for (each neighbor v of u) {
            alt = distance[u] + dist_between(u, v)
            if (alt < distance[v]) {
                distance[v] = alt
                previous[v] = u
            }
        }
    }
    return previous[]
}
```

3.1.2 A* Algorithm

When the pair origin-destination is known to the entity calculating the path, the method used is the A* algorithm [45]. This algorithm is also used in various fields, and is quite similar to the method described in the previous section. There is, although a difference between these algorithms that can make the A* method much faster, but also more restrictive. When choosing the next node in the path it is considered not only the cost to travel the edge between the two nodes but also an heuristic function, $h(x)$. To evaluate a node, x, the algorithm sums the value obtained by $h(x)$ with the cost from travelling from the starting point, to the node x, $g(x)$. The sum is represented by $f(x) = g(x) + h(x)$. The complexity of the A* method is the same as the Dijkstra algorithm, but its execution time is faster, in average.

By adding the heuristic function it is intended that the selection of the following nodes is more intelligent because the heuristic is supposed to give an estimate of the distance remaining

until the destination node. For the algorithm to be complete and optimal the heuristic must obey a simple rule. It cannot overestimate the distance to the destination.

To give an example of a use for this algorithm we'll take the graph that represents the city map in the RCR world and we'll try to plan the shortest path between two points in the city. Because, what is intended is to travel the shortest distance possible, a possible heuristic function to use with the algorithm is the Euclidean Distance[1] between the node being considered and the destination node. This distance is also commonly used when dealing with a graph where the nodes have geographical postitions. This heuristic is considered admissible because the distance calculated never exceeds the actual distance between the nodes. The heuristic is also optimistic due to the fact that the costs calculated are lesser than the cost of the real solution. The Euclidean Distance (also used in other implementations of the A* algorithm for calculating paths) calculates the path more quicker, but, inversely, makes searching slower because it would have to calculate a path tree for each pair origin-destination.

The implementation of this algorithm is very much like the Dijkstra algorithm. The following pseudo-code has only one difference when compared to the previous one, that is the calculation of the heuristic function when evaluating the distances of the neighbors to the target vertex.

```
AStarShortestPath(G, source, target) {
    final_list = empty list
    for (each vertex v in V) {
        distance[v] = infinity
        previous[v] = undefined
    }

    distance[source] = 0
    Vaux = all vertices of G

    while (Vaux is not empty) {
        u = extract-min-distance(V)                    // considers f(x)
        final_list.add(u)
        if (u == target)
            return previous[]
        for (each neighbor v of u) {
            if (v not in final_list) {
                alt = distance[u] + dist_between(u, v)
                if (alt < (distance[v] + heuristic(v,target))) {
                    distance[v] = alt
                    previous[v] = u
                    f[v] = distance[v] + heuristic(v,target)
                }
            }
        }
    }
    return failure
}
```

[1]Euclidean Distance - distance between two points measured in a straight line

3.1.3 D* Algorithm

The D* algorithm is a method used in robots for path planning [47]. This algorithm is similar to the A* algorithm, but it has a dynamic characteristic. The costs associated to the edges of a graph may change during the execution of the algorithm. Like previous algorithms for path planning, this one is applied to directed graphs, where a graph is represented by $G = (V, E)$, and each edge connecting two vertices has a cost associated to it. The cost of an edge is given by the cost function c, where $c(X, Y)$ is the cost of the edge connecting the nodes X and Y.

Like some implementations of the A* algorithm, this method also maintains the NEW, $OPEN$ and $CLOSED$ sets of vertices. Initially all vertices are in the NEW set, but as the algorithm executes some move to the $OPEN$ set, and later on some move to the $CLOSED$ set.

For the calculation of the costs and estimates there are several functions used. For the estimate of the cost of traveling from X to the destination vertex, G, there is the *path cost* function, $h(X, G)$, which can be equal to the optimal cost of traveling from one vertex the other, $o(X, G)$. The minimum value of $h(X, G)$ is given by the *key* function, $k(X, G)$, where $X \in OPEN$. This function returns the minimum value of $h(X, G)$ obtained since the vertex X was placed in the $OPEN$ set, and is consulted prior to an alteration of the cost of traveling from X to G.

The key function is used to classify a vertex into the $LOWER$ or $RAISE$ class, or state. A vertex is in the $LOWER$ state if $k(G, X) = h(G, X)$ and in the $RAISE$ state if $k(G, X) < h(G, X)$. Both states provoke changes on the path costs that will be propagated, and can have different reasons to do so. An increased arc cost can cause a $RAISE$ state, which will increase the path cost. For a vertex to be classified in a $LOWER$ state an arc cost may be reduced, or a new path can be calculated, which will decrease the path cost. The mentioned propagation acts by removing vertices from the $OPEN$ set, and for each vertex removed with the propagation, the algorithm expands to its neighbors, by passing them the path cost and adding them to the $OPEN$ set.

The D* algorithm also includes the parameters k_{min} and k_{old}, obtained from the key function. The parameter k_{min} is equivalent to $min(k(X))$ where $X \in OPEN$, and represents a threshlod that indicates that a path is optimal if it as a cost equal to, or less than k_{min}. The k_{old} parameter corresponds to k_{min} before the most recent removal of a vertex from the $OPEN$ set. If no vertex has been removed from the $OPEN$ set, then k_{old} is undefined.

The execution of the algorithm is based mainly on the functions PROCESS-VERTICES and MODIFY-COST, where PROCESS-STATE calculates the optimal path costs to the destination vertex, and MODIFY-COST changes the cost of edges and places vertices on the $OPEN$ set. The PROCESS-VERTICES function is executed repeatedly until the starting vertex is removed from the $OPEN$ set, meaning that the path has been determined, or the function returns -1 meaning that there is no path to the destination vertex. The same function can later be called again after the MODIFY-COST function changes the costs of some edges. The MODIFY-COST function is called when the path previously computed has errors such as obstacles, which are discovered while traversing the path.

To simplify matters the $h(X, G)$ notation will be represented as $h(X)$. Initially the destination node, G is placed in the $OPEN$ set, and $h(G)$ is set to 0. This is similar to the A* but instead of beggining at the starting node, the D* algorithm begins with the destination node. The function uses some functions to perform certain tasks. $MIN - VERTEX$ returns the

vertex, X in the $OPEN$ set with the smallest $k(X)$ value; $GET-KMIN()$ returns the k_{min} value, unless the $OPEN$ set is empty, in which case it will return -1; $DELETE(X)$ removes X from the $OPEN$ set, and places it on the $CLOSED$ set; $INSERT(X, k_{new})$ moves X to the $OPEN$ set if it was in the NEW set, setting $h(X) = h_{new}$, or if X is in the $OPEN$ or $CLOSED$ set $h(X) = min(h(X), h_{new})$.

The resulting shortest path is represented by a set of backpointers, $b[]$, where $b[X] = Y$ means that X has a backpointer to Y. This is also similar to the A* algorithm, where instead of vertices having backpointers they have previous vertices. Also, comparatively to the A* algorithm, where the starting vertex doesn't have a previous vertex, in the D* method the destination vertex doesn't have a backpointer.

The following pseudo-code demonstrates a possible implementation of the PROCESS-VERTICES function:

```
PROCESS-VERTICES () {
    X = MIN-VERTEX()
    if (X = NULL)
        return -1
    kold = GET-KMIN()
    DELETE(X)
    if (kold < h(X)) {
        for (each neighbor Y of X) {
            if (h(Y) <= kold and h(X) > (h(Y) + c(Y, X)) ) {
                b(X) = Y
                h(X) = h(Y) + c(Y, X)
            }
        }
    }
    if (kold = h(X)) {
        for (each neighbor Y of X) {
            if (t(Y) = NEW or ( b(Y) = X and h(Y) != (h(X) + c(X, Y)) )
                          or ( b(Y) != X and h(Y) > (h(X) + c(X, Y)) ) ) {
                b(Y) = X
                INSERT(Y, h(X) + c(X, Y))
            }
        }
    }
    else {
        for (each neighbor Y of X) {
            if ( (Y in NEW) or ( b(Y) = X and h(Y) != h(X) + c(X, Y) ) ) {
                b(Y) = X
                INSERT(Y, h(X) + c(X, Y))
            }
            else {
                if (b(Y) != X and h(Y) > (h(X) + c(X, Y)) )
                    INSERT(X, h(X))
                else
                    if (b(Y) != X and h(X) > (h(Y) + c(Y, X))
```

```
                                and (Y in CLOSED) and (h(Y) > kold) )
                        INSERT(Y, h(Y))
            }

    return GET-KMIN ()
}
```

The following pseudo-code demonstrates a possible implementation of the MODIFY-COST function.

```
MODIFY-COST (X, Y, cval) {
    c(X, Y) = cval
    if (X in CLOSED)
        INSERT(X, h(X))
    return GET-KMIN ()
}
```

After the MODIFY-COST function is executed the edge cost from X to Y will be changed, and X will be on the $OPEN$ set. Next the PROCESS-VERTEX is called and it will expand the X vertex, recalculating the $h(Y)$ value, and placing it on the $OPEN$ list. This process continues to the neighbors of Y.

3.1.4 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm[2] is another method to determine the shortest path between two nodes in a graph [15]. This method, however, does not calculate that single path, but all shortest paths between every pair of nodes in the graph. The algorithm is of complexity $O(n^3)$ and can be solved in polynomial time.

The Floyd-Warshall algorithm is a Dynamic Programming (DP) algorithm. The term *Programming* does not mean writing computer code, but instead refers to a tabular method. What characterizes DP algorithms is that they obtain their solutions by dividing the problem into subproblems, and then solving the subproblems in order to use those solutions to obtain the solution to the initial problem. This strategy is also used in divide-and-conquer methods, but DP extends the same strategy to dependent subproblems, i.e., subproblems with subsubproblems. A DP algorithm solves its problem beggining with the lower problems, and escalades until solving the original problem. The solution to a subproblem, or subsubproblem, is saved in a table (hence being a *tabular* method) to avoid recomputation of the subproblem in the future.

These kind of methods are generaly used in optimization problems, where the idea is to obtain an optimal solution which can be a minimum or maximum value. DP algorithms can be divided into three essential steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.

[2] Also commonly known as Floyd algorithm

There can be a fourth step in a DP algorithm when what is intended is not just the value of an optimal solution, but instead the construction of an optimal solution using the values obtained when executing the algorithm.

Before explaining how the Floyd algorithm works it is first necessary to explain the notion of intermediate point. For a path $p = (v_1, v_2, ..., v_l)$, an intermediate vertex of p is any vertex in p, with exception to v_1 and v_l. To put it in other terms, an intermediate vertex of p is any in the set $(v_2, v_3, ..., v_{l-1})$.

Considering a graph $G = (V, E)$, where $V = (v_1, v_2, ..., v_n)$. The algorithm will determine the shortest path between every pair of vertices in V, and keep them in a matrix, D, of $n \times n$ dimension, where $D_{i,j}$ is the shortest path from i to j. To do so it will consider a subset of V, $(v_1, v_2, ..., v_k)$. For any pair of vertices $i, j \in V$ it will plan paths from i to j where the intermediate vertices of those paths are in the set $(v_1, v_2, ..., v_k)$.

For each pair $i, j \in V$ the algorithm starts with a small set of intermediate vertices and tries to plan a path from i to j with that small set. The set of intermediate vertices expands as the algorithm continues since in each iteration a new vertex is added to the set. Even if the algorithm as found a path from i to j that path may change with a new iteration when a new vertex, v_{k+1}, is added to current set of intermediate vertices, $v_1, v_2, ..., v_k$. A new vertex in the set of intermediate vertices can also mean the discovery of a path between two vertices. If for a set of intermediate vertices, $v_1, v_2, ..., v_k$, a path between vertices i and j cannot be determined, then there may be a vertex v_{k+1}, such that there is a path p_1 from i to v_{k+1}, and another, p_2 from v_{k+1} to j. In both paths the intermediate nodes come from the set $(v_1, v_2, ..., v_k)$. Figure 3.1 shows an example.

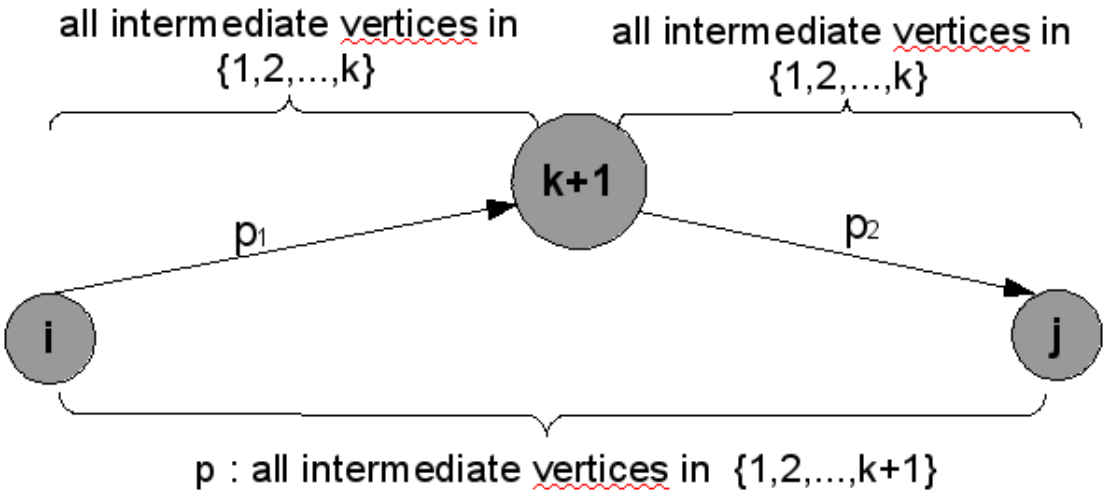

Figure 3.1: Shortest path from i to j where k is the highest intermediate vertex [15]

The following pseudo-code demonstrates how the algorithm works. Note that for this pseudo-code the matrix D is a path cost matrix initialized with the weight costs. If there is no edge linking two nodes, then the cost assigned is ∞.

```
FWSearch (G) {
    n = number of vertices in G
    D = path cost matrix
    for ( k = 1 to n ) {
```

```
            for ( i = 1 to n ) {
                for ( j = 1 to n ) {
                    D[i][j] = min( D[i][j] , D[i][k]+ D[k][j] )
                }
            }
        }
    }
```

3.2 Graph Traversing

The field of Graph Traversing studies methods for systematic and efficient visitation of all nodes in a graph. This field of study is present in many other fields of Graph Theory. When traversing a graph the goal can be to either search the graph, obtain a path from one vertex to another, or to identify the structure of the graph. Next it will be presented some known algorithms of Graph Traversing, like Breadth-First and Depth-First.

3.2.1 Breadth First

Breadth-First traversal is a well known algorithm widely used when dealing with graphs, both directed or undirected. Other search algorithms derive from this one, such as the Prim minimum-spanning-tree algorithm, and the Dijkstra algorithm [15]. This method is also useful when dealing with trees. The total running time of the Breadth First search method for a graph $G = (V, E)$ is $O(|V + E|)$, and thus runs in linear time.

The Breadth-First algorithm starts in a given starting node, s, that belongs the vertices set, V, of a graph, $G = (V, E)$. It will then explore the neighbours of s. After opening each neighbour, it will expand them by exploring its neighbours. This process repeats itself until the construction of a tree containing all accessible vertices from s is completed. For any vertex, v, in the tree, the path from s to v built by the tree is the shortest path, assuming that all edges have the same weight.

While the algorithm explores and expands all reachable vertices from the starting point, it will need to mark which nodes have been explored, and which haven't. To that extent the algorithm will color every node white at start up. When a node is first discovered it will be marked grey, and placed in a queue for future exploration. When a grey vertex has been explored, i.e., its neighbours identified, it will be marked black (other implementations refer to these nodes as *visited*), thus preventing from the algorithm entering a cycle. During the exploration of the graph two sets of data, of size $|V|$, are built and refer to the vertices in the graph. One saves the *distance* from the starting point to that vertex, the other saves the *previous*, or parent, vertex in the resulting tree. The values for the distance and previous vertex for the root vertex are 0 and $NULL$, respectively.

In figure 3.2 the evolution of a search using the breadth-first method can be seen.

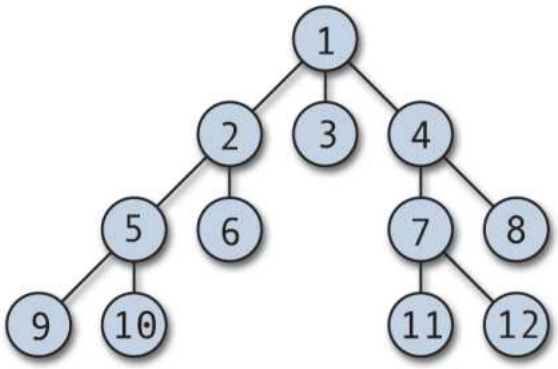

Figure 3.2: The evolution of breadth-first search [55]

The following pseudo-code demonstrates the execution of the breadth-first algorithm:

```
BFS (V, s) {
    for (each vertex u in (V - {s})) {
         color[u] = WHITE
         distance[u] = infinity
         previous[u] = NULL
    }
    color[s] = GRAY
    distance[s] = 0
    previous[s] = NULL
    Q = empty_queue
    ENQUEUE(Q, s)
    while (Q not empty) {
        u = DEQUEUE(Q)
        for (each v neighbour of u) {
            if (color[v] = WHITE) {
                color[v] = GRAY
                distance[v] = distance[u] + 1
                previous[v] = u
                ENQUEUE(Q, v)
            }
            color[u] = BLACK
        }
    }
}
```

As mentioned previously, the resulting search tree is composed of the nodes reachable from the starting node, which is not necessarily all the nodes in the graph as the graph may be disconnected. This algorithm is good for search purposes, but the way it expands is not appropriate for node visiting because two sequenced nodes may not be adjacent. An example can be seen in figure 3.2 with nodes 10 and 11.

3.2.2 Depth First

Another algorithm widely used in the field of Graph Theory is the Depth-first algorithm. This method shares some similarities with the Breadth-first method. It can also be applied to a tree. It works on directed graphs, as well as undirected. The algorithm takes a starting vertex and expands it into a tree. That tree represents the vertices reachable from the starting point. Because the steps performed in the Depth First search are similar to the ones in Breadth First search, the complexity is of the same order, and the Depth First algorithm also runs in $O(|V + E|)$.

The difference between this method and the previous is how the expansion is performed. As the name implies, the expansion is performed in depth. A vertex expands its neighbour, and that neighbour will expand its neighbour, and so on, until there are no more neighours to expand. When there are no vertices to expand, then a backtrack is performed on the tree until a vertex with unexplored neighbours is reached and the process continues. During expansion the vertices are also marked in colors that represent its state. White is for unexplored vertices, Gray is for vertices with neighbours left to expand, and Black is for vertices with no more neighbours to expand.

In figure 3.3 the evolution of a search using the depth-first method can be seen.

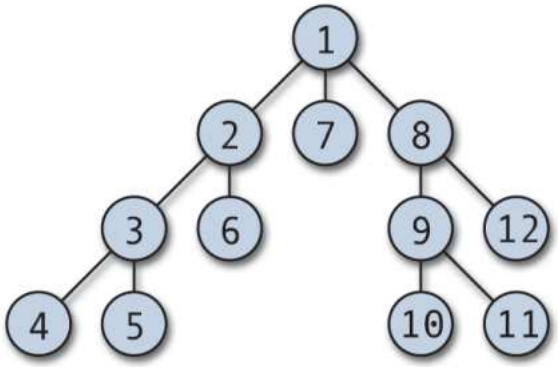

Figure 3.3: The evolution of deapth-first search [56]

It is also common to use timestamps when expanding the resulting tree. Each vertex in the tree will have two timestamps. One for when the vertex was first visited (turned gray), $d[v]$, another for when the vertex was completely visited (turned black), $f[v]$. The timestamp is used by other algorithms and can be useful to determine the performance of the algorithm, or to know how deep can a branch get (the bigger the difference $f[v] - d[v]$, the deeper the branch).

The following pseudo-code demonstrates the execution of the depth-first algorithm:

```
DFS (V, s) {
   for (each vertex u in V ) {
       color[u] = WHITE
       previous[u] = NULL
   }
```

```
    DFS-VISIT(s)
}

DFS-VISIT(u) {
    color[u] = GRAY
    time = time + 1
    d[u] = time
    for (each v neighbour of u) {
        if (color[v] = WHITE) {
            previous[v] = u
            DFS-VISIT(v)
        }
    }
    color[u] = BLACK
    time = time + 1
    f[u] = time
}
```

This method uses a recursive funtion, but an iterative method can be implemented using a stack to keep the vertices that might still have unexplored neighbours.

3.2.3 Related Algorithms

In section 5.3.1 it will be explained the importance of performing an efficient search. When looking at the problem from the graph abstraction it is a matter of visiting all nodes the quickest way possible. That problem has been in discussion in the Graph Theory field of studies for a long time. This is also known as the Traveling Salesman Problem (TSP).

A traveling salesman has to visit a determined set of cities starting from one point, and ending in the same one. The goal is to plan a path that minimizes the travelled distance and ensure that each city is visited only once. Although simple in theory, this problem can be of a very high complexity dependending on the number of cities, and the constraints to travel from one city to the another.

A solution to the TSP will be a Hamiltonian cycle. By definition, a path that visits each node in an undirected graph exactly once, and returns to the starting point. To define mathematicaly the problem it is necesary a graph $G = (V, E)$, and F the set of Hamiltonian Cycles existing in G. To determine the cost of a path it is required a weight matrix, C, where each edge, $e \in E$, has a weight, c_e, associated. The traveling salesmen problem has the goal of determining the hamiltonian cycle in F with smallest cost possible. This problem can be divided into two classes depending on the nature of the graph. If it is undirected then the weight matrix will be symmetric and the TSP is called Symmetric Traveling Salesman Problem, or STSP. In the opposite case, with a directed graph, or digraph, C will be asymmetric and the problem is called Asymmetric Traveling Salesman Problem, or ATSP. Despite having two different classes, an ATSP can be seen as a special case of STSP by introducing into the weight matrix of a directed graph the missing edges and assigning them large weights. The inverse transformation can also be done by doubling the edges between two nodes on an undirected graph (one for each direction).

There are several variations to the TSP.

- Max TSP - the travelled distance is maximized;
- TSP with multiple visits - all nodes are visited at least once. Travelled distance is still minimized;
- Bottleneck TSP - the highest edge cost is as small as possible;
- Messenger Problem - determines Hamiltonian paths instead of cycles;
- Clustered TSP - nodes in V are partitioned into clusters $V_1, V_2, ..., V_k$. The salesman must visit all nodes of one cluster before travelling to the next one;
- Generalized TSP - like the previous, nodes are partitioned into clusters. The salesman must visit all exactly one node from each cluster.

An algorithm that solves TSP can have several applications. It can be used in machine scheduling, cellular manufactoring, arc routing, frequency assignment, structuring of matrices, among others.

The TSP can be considered a Graph Traversing problem because its goal is to visit all the nodes in a graph, like many Graph Traversing algorithms, but the final outcome is a path that the Traveling Salesman will follow in order to visit every city in the map. Because of that the TSP can also be considered a problem of Path Planning.

3.3 Graph Partitioning

The virtual cities where the simulations take place vary in size and shape. If an agent is placed in a large city it will mean that his area of action is also large which means that the agent will need to save more information in its world model, which means more processing required to analyze all that information when deciding its next step. Although it is preferable to have a large set of information when making a decision, such an amount is only useful when its not superfluous. Otherwise it will only slow down the decision making process.

A Graph Partitioning method takes a Graph, $G(V, E)$, and divides the set of vertices V into k disjoint subsets, V_1, V_2, V_3, ..., V_k. The set of edges cut by the division of the graph is represented by E_c. A good partitioning method should be developed with the goal of having resulting partitions which balance the cardinality of the subsets of V, $|V_i|$, as much as possible, and also try to minimize the size of E_c.

Graph partitioning problems are of a NP-complete complexity, which makes viable solutions for those problems to be few, and the existing solutions are very specific regarding the graphs that they can partition. An algorithm can also be very efficient if the size of the graph is not too great or if the number of partitions is 2. The reason for this is that a small increase of the graph can make the algorithm see its running time increase exponencially. But the uses for methods that can partition a graph are many and they are implemented to solve problems of VLSI design, data-mining, finite element and parallel computing. This means that if a method that provides balanced subgraphs with the minimum value for $|E_c|$ cannot produce a solution in a reasonable time span, then an approximation must be used. Previous research has given us some solutions such as simmulated annealing, genetic algorithms, neural networks or ant colony based systems [16]. The template for the mentioned methods includes the algorithm that performs the partitioning, and a method to determine if the solution found at the moment is optimal enough.

3.4 Graph Partition Refinement

With Graph Partitioning came the need for a refinement of the partitions created. Although most algorithms attempt to create balanced partitions it can be very hard to do so in optimal time, or that would make the partitionig algorithm very complex. The solution to this problem was to create refinement algorithms that took the unbalanced partitions and produced balanced ones. This does not make the problem easier to solve, it is still a NP-Hard problem, but the separation of the methods makes their creation much more simple.

In fact, many partitioning algorithms that exist and are developed are actually refinement algorithms, because the partition itself is done by another method, with little regard as to the partitions created[3], with the main interest that it creates those partitions quickly. Nevertheless this may change according to the parameters and applications of the algorithm.

During the development of the partitioning algorithms for this work, it was also needed a refinement algorithm. Like the partitioning algorithms, the development of the refinement process followed no known algorithm in particular, but instead obtained pointers from existing algorithms [25, 23, 13, 27].

3.4.1 Kernighan-Lin

The Kernighan-Lin heuristic [27] was developed in 1972 but it is still used in many Graph partitioning methods. The goal of this heuristic is to partition a graph in a way that reduces the number of connections between the partitions. The algorithm that implements this heuristic will swap connected vertices between partitions until $|E_c|$ has reached a minimum. The swap is always done with a pair of nodes in order to maintain the partitions balanced. While executing, the algorithm considers the external cost and the internal cost of the partitions. The external cost is the sum of the weights of the edges cut by the partitions. The internal costs is the sum of the weights of the edges included in each partition. Both costs can be determined by the sum of external and internal costs of each partition. The difference is that for the external cost, the sum will use all the nodes, whereas the internal cost is calculated using the nodes belonging to each partition. To present these concepts mathematically it is first necessary to consider the costs matrix, C, this matrix holds the costs for each edge. For a graph with n vertices:

$$C = (c_{i,j}), i, j = 1, ..., n$$

To explain the steps performed by the algorithm we'll use as an example a graph divided into two partitions, A and B. The external cost of a vertex of A is defined by the following formula:

$$E(a) = \sum_{j \in B} c_{a,j} \tag{3.1}$$

And the internal cost for the same vertex is calculated using the following formula:

$$I(a) = \sum_{j \in A} c_{a,j} \tag{3.2}$$

The difference between the two previous costs is given by:

$$D(a) = E(a) - I(a) \tag{3.3}$$

[3]Take for example the Ants Algorithm described in 3.4.2 where the partitions are created randomly

When choosing which vertices to trade what is used is the gain, or reduction in cost. For a node $a \in A$ and a node $b \in B$ the gain is given by this equation:

$$g(a, b) = D(a) + D(b) - 2c_{a,b} \tag{3.4}$$

The best pair of nodes selected to be exchanged is given by the previous formula. The pair that provides the biggest gain, is the pair selected. But that selection will only come after the algorithm calculates the value of D for every node. This is the first step. Only after that does the algorithm select the pair to be exchanged, and it will continue to exchange vertices until no improvement is possible. The following exchanges, though, will need some recalculation. Because a, and b do not belong to A and B, respectively, the calculation of the new values of D must reflect that difference. For this reason, the vertices of {A-a} and {B-b} will be calculated as follows:

$$D'(i) = D + 2c_{i,a} - 2c_{i,b}, i \in \{A - a\} \tag{3.5}$$

$$D'(j) = D + 2c_{j,b} - 2c_{j,a}, j \in \{B - b\} \tag{3.6}$$

After the previous step a new pair is selected, and the process repeats itself until all pairs, and its respective gains are identified.

The reduction cost of exchanging the selected pairs is $\sum_{i=1}^{k} g_i = C$. If $C > 0$ a reduction can be done. After the exchanges take place, the resulting partitions, A' and B', will pass through the same process. If $C = 0$ then a solution for the initial partitions has been found. Nevertheless, those partitions may still be further optimized. So, they once again pass through the algorithm.

Figure 3.4 shows how a small graph can be partitioned while maintaining the balance and reducing the connections between the partitions.

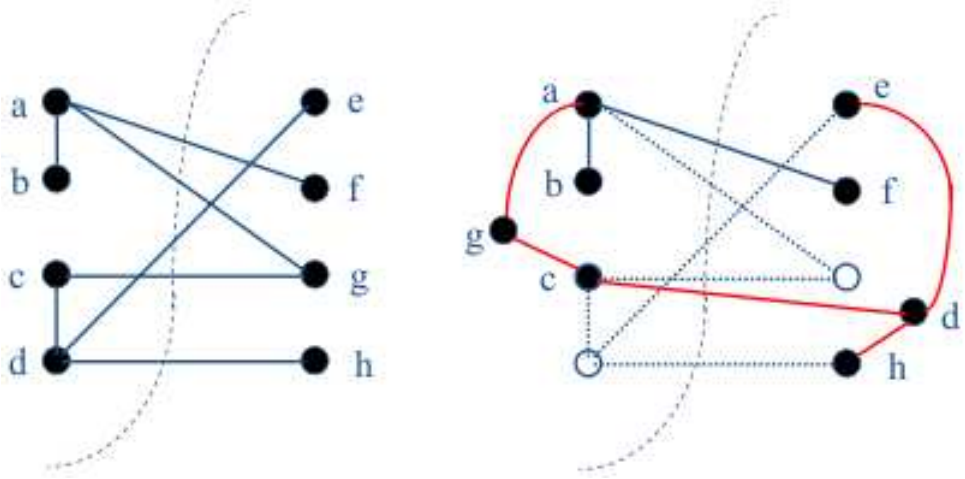

Figure 3.4: A possible solution to partition a graph with the Kernighan-Lin heuristic [10]

3.4.2 Ants Algorithm

This algorithm is a simplification of the ant colony algorithms [13]. The simplification comes from discarding the use of pheromones and local memory. It operates like a multi agent system and is based on parallel search.

According to the number of partitions, k, the algorithm will color the vertices in the graph in a random fashion, although trying to maintain the balance between the partitions. In the end the vertices will be divided into k groups, each with its color. After this step, a predefined number of ants - the agents - will be spread randomly among the vertices of the map. Those ants will move around in the map swapping the color associated to each vertex.

The method that evaluates the optimality of the solution uses a local cost function, and a global cost function. The global cost functions counts the number of times that all vertices are linked to neighbours with different colors. The local cost function calculates the ratio between the number of neighbours of different colors and the total number of neighbours. For each vertex, a ratio is associated.

The balancing of the subgraphs has already been assured in the first step, when the vertices where divided equally between the k color sets. When the agents move around the graph they will try to reduce cardinality of E_c, while maintaining the balancing. The vertices color exchange takes place when the agents move to a new vertex. That vertex will be the neighbour of the vertex where the agent is currently positioned that has the largest number of constraints, which is the number neighbours with different colors. After arriving at the vertex the agent will try to reduce the constraint number by changing the color of the vertex. The new color will be the one that provides the biggest reduction. To maintain the balance between the colors, diturbed by the exchange of colors in the previous vertex, A, a vertex, B, of the same color as the new color of A, is selected and its color is changed to the old color of A. The selection is based not only on the color of the vertex, but also on the local cost function value. The selected node will have the lowest local cost value. If more have the same value, they are randomly selected.

This method was given a probabilistic nature, by making the movement and coloring decisions of the agents dependent of a predefined probability for each type of decision. When moving, the agent will change to the worst adjacent vertex with probability p_m, or it will move to a randomly chosen neighbour. And it will change the color of the vertex to the color that provides the biggest reduction with probability p_c, or it will choose a random color. With the added probabilities the developers intend that the method obtain partitions with $|E_c|$ near absolute minimum, and also that it will escape a local minima.

Figure 3.5 shows an example of an agent moving from one node to another.

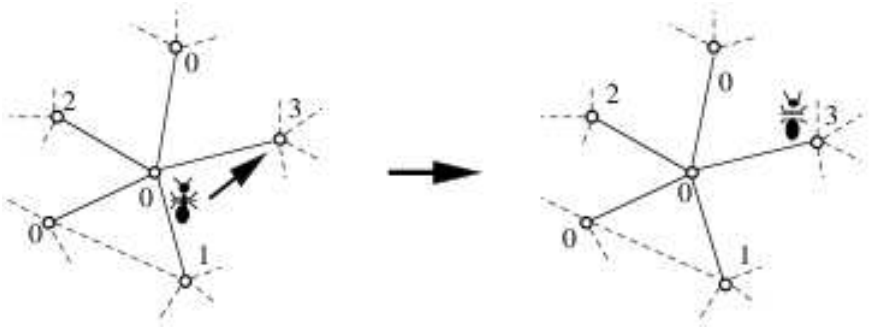

Figure 3.5: Movement of an ant towards the worst local node [13]

The process described here will be carried out until the algorithm converges. The following pseudo-code demonstrates how the algorithm works.

```
AntPart(G) {
```

```
    Initialize
        n (number of ants), k, pm , pc
        Put each ant on a randomly chosen vertex
        Color each vertex of the graph at random forming k balanced sets
        For (all vertices) {
            Initialize local cost function
        }
        Initialize global cost function
        best cost := global cost function
    While (best cost > 0) {
        For (all ants) {
            If (random < pm )
                  Move the ant to the worst adjacent vertex
              Else
                  Move randomly to any adjacent vertex
              End if

          If (random < pc )
               Change vertex color to best color
          Else
               Change to a randomly chosen color
          End if
          Keep balance (Change to the former color a random vertex)
          For (the chosen vertices and all adjacent vertices) {
               Update local cost function
               Update global cost function
          }
          If (global cost function < best cost )
               best cost = global cost function
       }
    }
}
```

3.4.3 Heuristic Method for Balanced Graph Partitioning

Another example of what was said previously is the method for balanced partitioning presented in [8]. In that method, the partitioning is performed by the Cartesian Nested Dissection method, while the algorithm developed is in truth a refinement algorithm based on the Kernighan-Lin method.

This method was developed with the intent of dividing city districs into police patrol areas (yet another used for graph partitioning methods). The vertices in the graph to be partitioned are city blocks, and the weights, which are applied to the vertices, are the crime occurrences in that city block.

This method begins by partitioning the graph using the Cartesian Nested Dissection algorithm. This method was chosen because it was simple to implement and it uses geographical coordinates to partition a graph, which is useful considering the application for the method developed.

The second step is to refine the partitions obtained in the previous step, by ditributing the weights in a more balanced manner, using a strategy similar to the Kernighan-Lin method presented in 3.4.1. The method uses some simple concepts that need to be explained before describing the algorithm itself:

- Cut vertex or articulation point - a vertex that, if removed, will increase the number of connected components;
- Admissible vertex - a vertex that has an edge that links it to a vertex in another partition. This cannot be a cut vertex;
- Partition weight ($w(P)$) - total sum of the weights of the vertices included in the partition P;
- Adjacent partitions - two partitions are adjacent if there is at least one edge connecting vertices in both partitions.

The algorithm will swap vertices between two adjacent partitions, but only one vertex at a time. Of those partitions, one should have a value of $w(P_1)$ greater than the other, $w(P_2)$, and it will be in this direction that the vertices will be exchanged. Swapped vertices must be admissible vertices, but that is not the only criteria for swapping vertices. The admissable vertices have a preference associated to them, and it is the vertex with the lowest preference that will be swapped. Those preferences must also be lower than a local homogeneity index, that indicates how spaced the partitions are in terms of weight. The index is calculated as follows:

$$\alpha = |w(P_1) - w(P_2)|$$

To evaluate the effect of swapping a given node, v from partition P_1 to partition P_2 it is necessary to determine the value of α after that node has been swapped. The new value is given by $\gamma(v)$ and is calculated using the following formula, where $w_{vertex}(v)$ is the weight of v:

$$\gamma(v) = |w(P_1) - w_{vertex}(v)| - |w(P_2) + w_{vertex}(v)|$$

With the goal of optimizing the performance, the algorithm will, at each iteration, swap vertices between the partition with the biggest weight, and the adjacent partition with the lowest weight. The process stops when there it is not possible to swap any more vertices.

The following pseudo-code demonstrates the execution of the algorithm:

```
RefinementMethod(Partitions[]) {   // method receives an array
                                   // containing the partitions created
    For (each partition) {
        // assign optimal weight to partition
        total_graph_weight / number_partitions
    }
    // determine adjacency points in partitions
```

```
    INITIALIZEADJACENCYPARTITIONS(Partitions)
    order Partitions[] descending according to their weights
    indexCurrentPartition = 0
    numberIterations = 0
    currentPartition = NULL
    while (indexCurrentPartition != Partititions.length) {
        currentPartition = Partitions[indexCurrentPartition]
        order partitions adjacent to currentPartition
          ascending according to their weights
        For (each partition adjacent to currentPartition) {
            numberIterations = LOCALREFINEMENT(adjacentPartition,
                                currentPartition)
            if (numberIterations != 0)
                  stop for-cycle
        }

        If (numberIterations = 0)
            indexCurrentPartition = indexCurrentPartition + 1
        Else {
            indexCurrentPartition = 0
            numberIterations = 0
            UPDATEADJACENCYPARTITIONS(Partitions)
            order Partitions[] descending according to their weights
        }
    }
}
```

3.5 Final Considerations

The algorithms described in this section provided the methods needed for the problems encountered. However, that does not make them the optimal solutions. There are at least two essential features for an algorithm that can allways be improved. Those features are complexity, i.e. its reduction, and speed. The problem is that most times, the decrease in complexity and speed often leads to the decrease in the optimality of the solution provided by the algorithm. This means that depending on the problem and the resources available, the developer will have to compromise. If there is the need for an immediate solution, then optimality will have to fall to a secondary plane of importance. This, of course, may change according to the problem at hand.

Chapter 4

Agent Interaction

In a complex environment like the RCR world, the agents have specific tasks and many obstacles in the way of performing those tasks, including time limitations. Therefore it is very unlikely that an agent could present good results by acting on its own (specially considering that it might begin a simulation between two roadblocks and thus be unable to move). To make things easier, in each simulation there are several agents of each type. Increasing the number of agents with the same goal seems an obvious way of achieving the objectives set for those agents, but that might not be the case. If each agent, although with the same goal, is out there for its own, then there is the possibility that the agents might get in the way of each other. This means that individual action will probably lead to poor results. In that case, if there are several agents, then they should all work together as a team and improve their results. This is a problem that affects nearly every multi-agent systems with homogeneous and heterogeneous agents, and as such it is widely studied by researchers in the field of Artificial Intelligence.

To implement cooperation among agents it is extremely useful that the agents can communicate among themselves, which leads to other questions. The first one is what do agents say to each other? What language do they use? In certain environments there are limitations in the network used, which means limited amount of information passed with each interaction, some messages might be dropped or there may be other constraints imposed on the agents, like in the RCR world. These questions lead to two different fields. One deals with the language used by the agents, the other deals with the communication protocol, or the steps the agents take when communicating.

Cooperation with communication is hard allready, but there may be situations where the agents do not communicate. That may happen because the agents simply do not possess such ability, or because communication has been cut-off. In any of these cases, the agents should know how to operate in a coordinated manner. In this chapter it will be presented several strategies for these two main concerns of agent development, and of great relevance in the RCR domain.

4.1 Cooperation

In [46] the authors define cooperation as "a type of relationship that is evident within structured teams when an agent is required to cooperate with and explicitly trust instructions and information received from controlling agents". This is true particularly in scenarios where

the RCR Center Agents are available, since they mostly act as central units that instruct the platoon agents on their actions. But in RCR simulations the agents cannot limit their cooperation for when there are Centers available, which is not a constant. That coupled with the fact that there is no central point that Center agents communicate with, then that statement loses strength. A more generic and correct definition for what is cooperation between agents would be the act of working together and compromising to reach a common goal.

The same authors establish colaboration as a different concept than cooperation and define colaboration as something that "involves the creation of temporary relationships between different agents and/or humans that allows each member to achieve their own goals" [46]. An example of this statement in practice can be given by the relation between the different types of agents of RCR, although it is not totally applicable because RCR agents have the same global objective. Using as an example Ambulances and Polices, an Ambulance by calling a Policeman to clear a roadblock that is preventing it from reaching a trapped civilian is establishing a temporary relation where both agents will gain something.

4.1.1 Locker Room Agreement

In 1999 a new method for agent cooperation was presented by Stone and Veloso [48]. This method's goal was to present a valid solution for the coordenation of teams of agents in real-time environments with limited communications. The result of the research performed by Stone and Veloso was a team agent architecure that allows a flexible change of team agreements. From those agreements come the team formations, which specify roles that divide the task space. The roles are assigned to agents, that can switch with homogeneous agents, just as agents can switch formations.

The nature of the environment where the agents perform their tasks led to the introduction of Periodic Team Synchronization (PTS) domains. In such a domain, agents have two separate states depending on the available communication. When there is no communication the agents must act autonomously while performing their given tasks. The second state ia a periodic setting, when agents are able to communicate and synchronize themselves.

In the first state, when working autonomously the agents will function according to pre-established strategies. Those strategies were set for the entire team of agents, and are called locker-room agreements. Although some say that pre-determined agent strategies lack robustness and flexibility, they also contribute to agent cooperation, since all agents will be following the same strategies which are based in cooperative behaviors. In the second state, agents use the unlimited communication to inform other team agents of their progress, actions executed and share information gathered. During this period the locker-room agreement may change, and the agents will have new pre-defined strategies to guide them.

The architecture used with locker-room agreement allows the agents to sense their surroundings, keep the agent's view of the world, and make decisions regarding its actions on the world. When pausing for synchronization with remaining team members, the agent will be able to save the new locker-room agreements, which will only interact with the internal behaviours. Figure 4.1 displays the agent architecture, and how the locker-room agreements fit into the agent's behaviour.

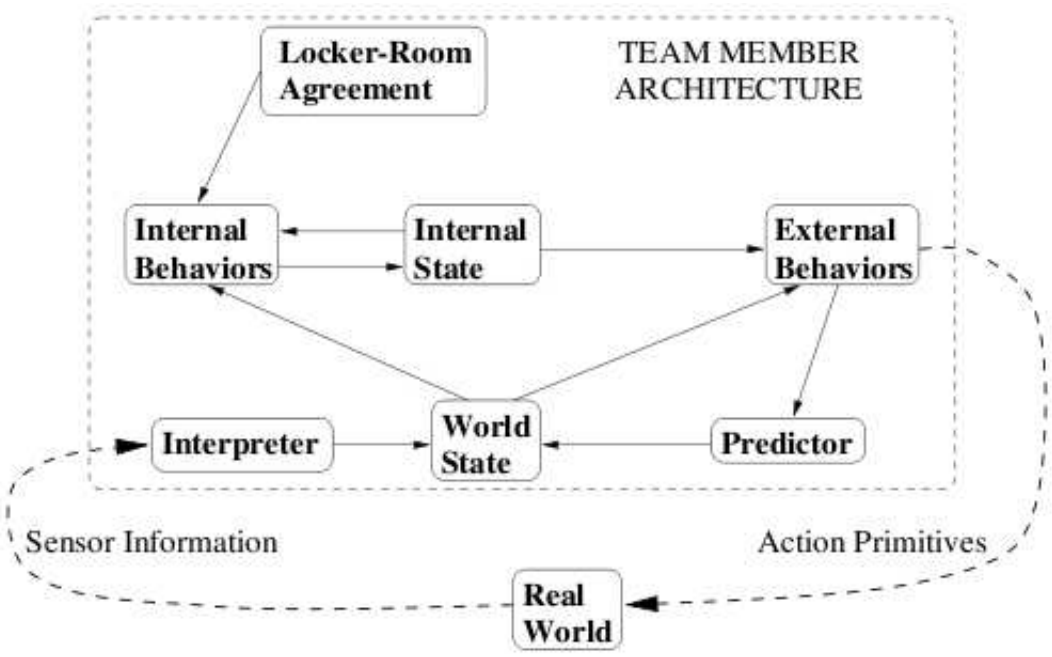

Figure 4.1: Agent architecture featuring locker-room agreements [48]

The locker-room agreement is composed of formations, the roles present on those formations, the protocols that allow agents to switch roles and formations, and set-plays. It also includes the mapping of the agents to the roles.

Roles A role specifies the internal and external behaviour of the agent. It can be rigid or flexible, regarding the agent's behaviour. If the role is rigid, then the agent must behave exactly as the role describes. With a flexible role, the agent has parameters and behaviour options that shape the agents actions, and allow a degree of flexibility. For example, an agent that must blow a whistle 75% of the times that the hour changes. At a given hour, the agent can choose whether to blow the whistle or not. For example, in robotic soccer simulation a role could be the position of midfielder.

Formations The formations are what defines the roles the agents will take and what will ultimately result in agent cooperation. To avoid having roles left unfilled, the formations will only define roles for the exact number of available agents. A formation can be composed of sub-formations, or units, which are composed of roles, a unit captain, and interactions among the unit's roles. To define a formation, and its components mathematically we have a team of n agents $A = \{a_1, a_2, ..., a_n\}$. The formation is defined by $F = \{R, U_1, U_2, ..., U_k\}$, where R is a set of roles $R = \{r_1, r_2, ..., r_n\}$, where no two roles are equal, although they may be equivalent by defining the same behaviour in both roles. Note that the number of roles and the number of agents is the same. A unit U_i is a subset of R, where $U_i = \{r_{i,1}, r_{i,2}, ..., r_{i,l}\}$, such that $r_{i,j} \in R$, $l \leq k$, and no two unit roles are equal, although a role may be a part of several units.

The purpose of units is to solve sub-problems that do not warrant the attention of the whole team. For that kind of local problems, a unit is created with roles able of solving such

issues. One of the roles included in a unit is the captain, $r_{i,1}$. This role has special privileges that enable the agent filling the role to command the remaining unit members.

Set-Plays For situations that occur repeatedly it would be useful to have pre-defined actions in response to those situations. That set of pre-defined actions is called a set-play. A set-play is defined in the locker-room agreement so that all agents are aware of that. Because a set-play does not necessarily involve all agents, a function that maps the roles in the set-play to the agents is required, and it is also included in the locker-room agreement. That function is called whenever a situation happens where a set-play can be used.

To define a set-play it is necessary:

- A trigger condition that indicates the states in which the set-play is activated;
- A set of set-play roles $R_{sp} = \{spr_1, spr_2, ..., spr_m\}$, where $m \leq n$. Each set-play role includes:

  A set-play behavior to be executed

  A termination condition so the agents can tell when they can terminate their set-play roles, and resume their normal behavior

Remembering that the agents are autonomous and that they operate in a PTS domain, it is very difficult for an agent to know what roles are being filled by the remaining team members during the periods of limited communication. The solution is to periodically communicate with the other teammates to inform each other of the roles they are currently performing. For the communication to be successful, the protocol used will have to take into consideration the existing limitations, like low-bandwidth.

Figure 4.2 displays the evolution of the formations and the roles taken by the agents throughout time.

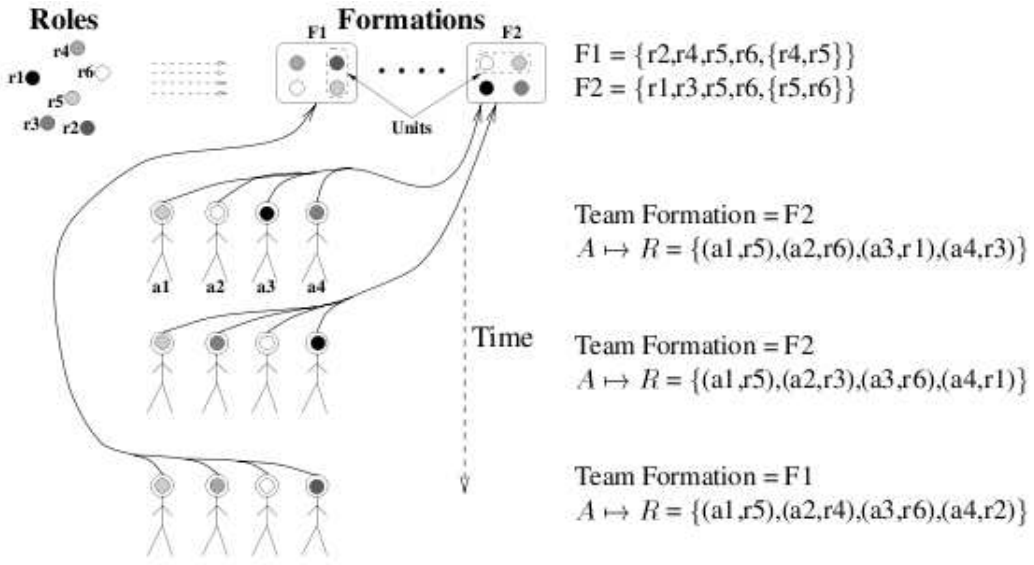

Figure 4.2: Agent architecture featuring locker-room agreements [48]

4.1.2 Situation Based Strategic Positioning

Situation Based Strategic Positioning [44], or SBSP, is a coordination mechanism used by the FC Portugal team for the RoboCup2000 Simulation League, which finished with the Portuguese team in first place in the European and World championships.

Before detailing the SBSP mechanism it is necessary to first define some concepts used in the description of the mechanism.

Team Strategy The team strategy is composed of a set of Tactics, Tactic Activation Rules, a set of agent Roles, and information concerning Opponent Modeling Strategy, Teammate Modeling Strategy and Communication Protocols.

This mechanism is responsible for the selection of the strategic positioning of every agent on the team. To reach a decision on the positions to be assumed by the team it will use the situation information, the current tactic, the current formation, player role, and positioning in the formation.

Tactic A tactic is composed of sets of Formations, Formation Activation Rules and Preset Plans. A tactic is selected according to Tactic Activation Rules defined in the Team Strategy, together with the Tactics.

Formation A formation holds the Positioning for the agents.

Preset Plan A preset plan is composed of Plan Activation Information, agents Positioning Evolution, Role Evolution and Actions along the time.

Positioning Positioning defines where an agent stands inside a formation, and is composed of Reference Position, Positioning Role and Positioning Importance.

Reference Position The reference position is defined by two coordinates:

$$ReferencePosition_{i,j,p} = (ReferencePositionX_{i,j,p}, ReferencePositionY_{i,j,p})$$

$$ReferencePositionX_{i,j,p} \in [-field\_length, field\_length]$$

$$ReferencePositionY_{i,j,p} \in [-field\_width, field\_width]$$

$$\forall i = 1, 2, .., ntactics, \forall j = 1, 2, .., nformations, \forall i = 1, 2, .., nplayers_{i,j}$$

This position defined using as parameters the agent type and the agent's strategic position, which is given by the situation information.

Position Importance The position importance uses the following qualitative scale: $PositioningImportance_{i,j,p} \in \{VeryLow, Low, Medium, High, VeryHigh\}$. For example, in the robotic soccer domain, the center defenders have a $VeryHigh$ importance in defensive situations, whereas center forwards have the same level of importance when in attacking situations.

Positioning Role The characteristics of a position in a formation are defined by the positioning role. The agent that occupies that position will assume those characteristics. Using once more the example of the soccer domain, a Role may define characteristics like $AggressiveDefender$, $PositionalAttacker$, etc.

Role A role that belongs to the set of Roles of a Team Strategy is composed of Active Characteristics, Strategic Characteristics and Critical Situation Rules. The Critical Situation Rules define which situations will cause an agent to alter from a strategic state, to an active state. Critical Situation Rules are defined by the Action Selection Information and they are evaluated using the current Situation (attack, defend, etc). The Role Strategic Characteristics and Active Characteristics are domain dependent, and must be defined accordingly.

Figure 4.3 shows an example strategy and its various components.

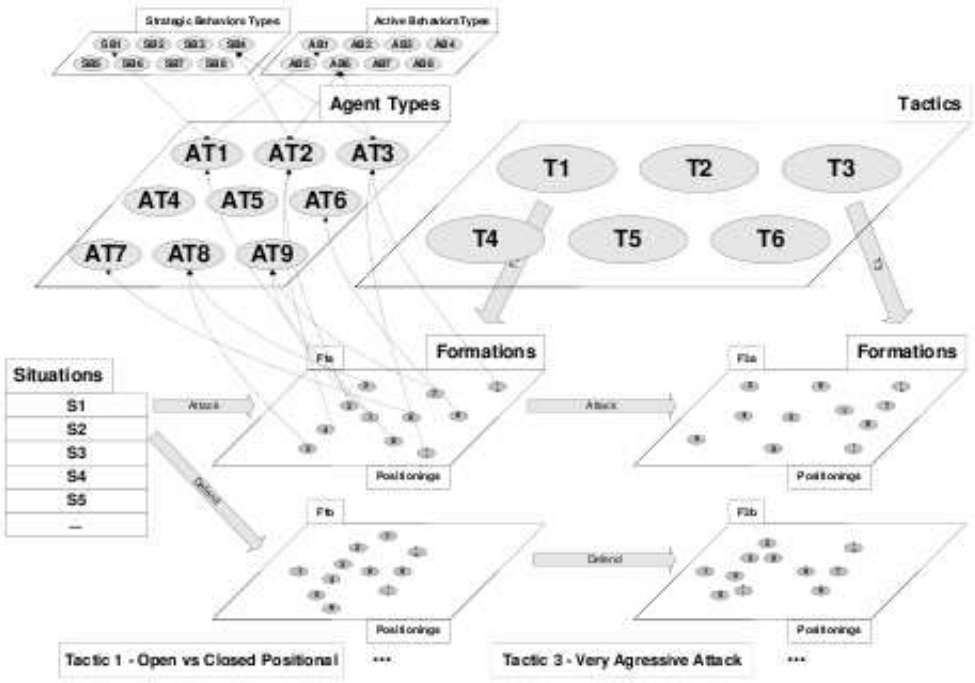

Figure 4.3: An example team strategy [44]

The SBSP is a coordination policy responsible for telling the agents what will be their positions in the formation. Those positions are determined by the SBSP mechanism using the situation information, the current tactic and formation, the role being filled by the agent, and the current position in the formation. The agents whose behavior is modeled by this coordination mechanism have two distinct types of situations: *strategic* and *active*. For each situation the decisions are made using different methods. For active situations the agents use domain specific high-level and low level skills, and for strategic situations they use SBSP. When in the latter situation, the agents will use SBSP to learn their positions. An agent finds itself in an *Active Situation* if one or more Critical Situation Rules have been fired. A *Strategic Situation* is exactly the opposite, where no Critical Situation Rules have been fired.

When on a strategic situation the agent will obbey the position assigned to it by the SBSP. That position may change with time, according to each situation that comes with

the game. There are several parameters involved in the SBSP coordination model. The information involved has several sources, since Game information, to Situation information and also Predefined strategic information. The following list gives a more complete idea of the variables taken into consideration when deliberating the agents' positioning.

- Game information:
  - competition time
  - competition result
  - opponent modeling information
  - competition mode
  - several statistics
  - attack and defensive information
  - positions
  - states and velocities of the objects in the world
- Situation based information:
  - plan currently in execution
- Predefined strategic information:
  - tactics
  - formations (for game situations)
  - agents' behaviors inside formations

Strategic behaviours, taken by the agents via the positions assigned to them by the SBSP, introduce a more cooperative behaviour to the agents. That is because of the positions that are allocated to the agents, which are defined using a wide view of the world and aim at maximizing the cooperation between the agents by assigning different tasks to different agents that will allow them to cover the entire field.

4.1.2.1 Dynamic Positioning and Role Exchange

An agent may change roles during a game, and the Dynamic Positioning and Role Exchange is what enables the agents to perform those changes. It is a concept based on the work described in section 4.1.1, but it extends it by adding an utility function that will evaluate if the change in roles brings any improvement to the team. To maintain the balance in the formation, the change occurs between two agents. The exchange in roles is facilitated for homogeneous agents, whereas for heterogeneous agents it would be necessary to consider the adequacy of the agent to the role.

The utility function uses the situation information and the predefined strategic information to estimate the strategic position for the agent and its teammates for each situation. Empowered with this knowledge, the agent can decide if there is anything to gain from changing roles, and positioning.

For a tactic $tact$, a formation $form$, and two players p_a and p_b, the utility of swapping roles is given by the following formula:

$$DPREUtility(tact, form, p_a, p_b) =$$

$$+Utility(Position_{p_a}, Positioning_{tact,form,p_b}, PlayerRole_{tact,form,p_b})$$
$$+Utility(Position_{p_b}, Positioning_{tact,form,p_a}, PlayerRole_{tact,form,p_a})$$
$$-Utility(Position_{p_a}, Positioning_{tact,form,p_a}, PlayerRole_{tact,form,p_a})$$
$$-Utility(Position_{p_b}, Positioning_{tact,form,p_b}, PlayerRole_{tact,form,p_b})$$
$$\forall p_a = 1..nplayers_{tact,form}, \forall p_a = 1..nplayers_{tact,form}, p_a < p_b$$

If a value obtained from the previous function is positive, then the agents may exchange roles, otherwise they maintain their present roles. The function also takes into consideration the distance between the agent's current position to its strategic position, the positioning importance, its state (physical conditions, objects carried, etc.) and the role's characteristics.

Since the agents possess local perspective, the utility function may return different values for the same exchange in each of the agents involved. To reduce the impact of these differences, the agents use communication to synchronize the exchange. After the exchange, if one of the agents was against the swap it may come to reconsider its opinion because the new position tends to increase the utility value calculated for that exchange. This fact reduces the relevance of the divergence in utility values calculated by the agents.

Figure 4.4 presents an agent architecture with the SBSP mechanism included.

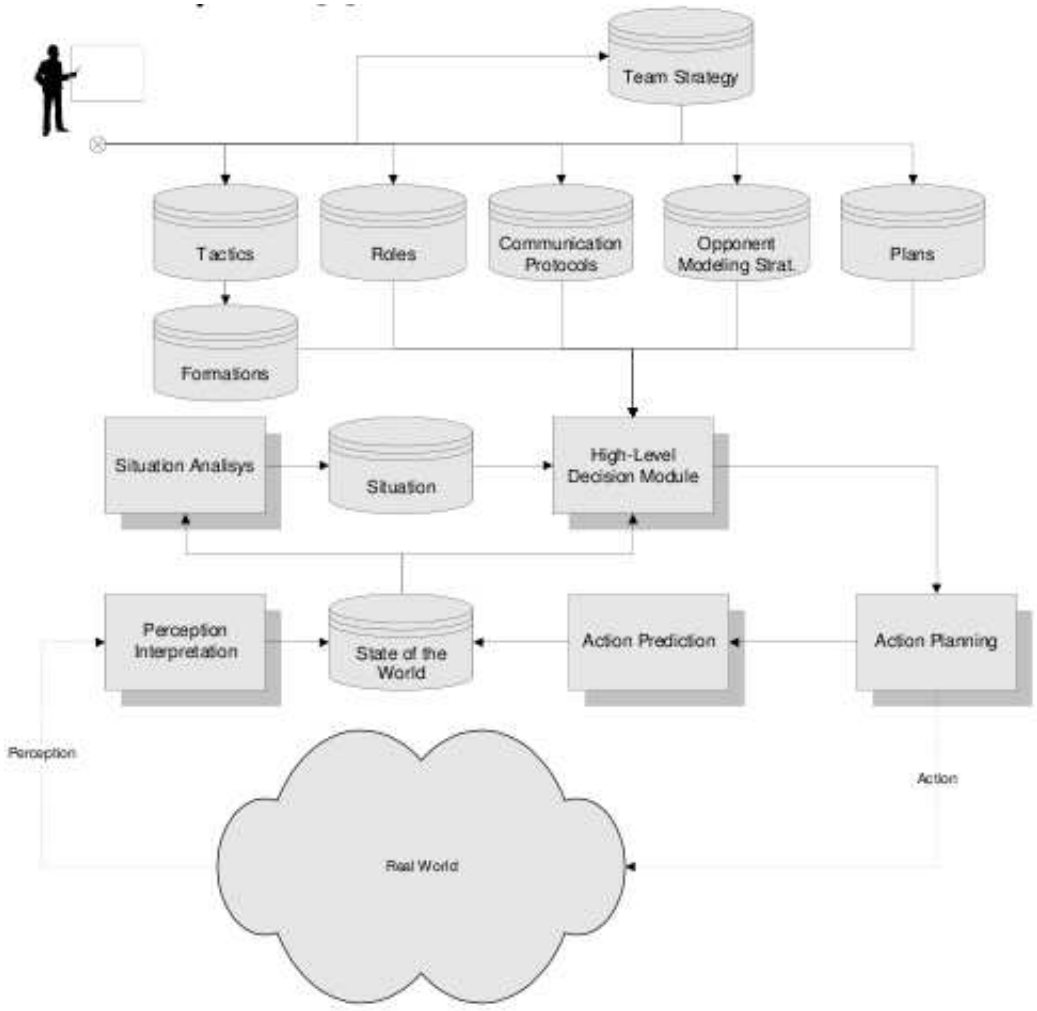

Figure 4.4: Agent Architecture with SBSP mechanism [44]

4.2 Communication

Communication between agents is much more than mere communication between computer systems. There is a much higher level of abstraction and the one of the main concerns is the language used by the agents, where the goal is to have a language similar to human's.

There are two types of communication between agents. Direct, and Assisted [26]. With direct communication the agents communicate directly with the target agent. There is no need for intermediaries. With assisted communications all messages pass trough *facilitators*, which are agents responsible for the reception and sending of messages to their respective destinations. Both types have their advantages and disadvantages. With direct communication the agents are more independent, but the communication is not coordinated. With assisted communication it is simpler to control communication, but like all centralized systems, there is the risk of communication break down if the facilitator agent fails, or is otherwise unable

to relay the messages. In the RCR world we have both kinds of communication. There is direct communication between agents of the same type, and assisted communication between agents of different types, where the messages must pass through the centers.

4.2.1 Language Components

When studying languages between intelligent agents there are some specific aspects that need to be considered [26]:

- Vocabulary - definition of the symbols used in the messages
- Syntax - establishes how the symbols are structured
- Semantics - what do the symbols in a message mean
- Pragmatics - how the symbols are interpreted

By joining semantics and pragmatics we have meaning, which has several dimensions as listed below:

Descriptive vs. Prescriptive Description is something natural to human communication, but difficult to implement in machine language. For that reason agents normally exchange simple information and behaviors.

Personal vs. Conventional Meaning The exchange of messages may happen between heterogenous agents with different pragmatics and/or semantics. In these cases a message may be misunderstood because the receiving agent assigns a meaning different from the one the sending agent intended. To avoid this, communication must be designed in the more conventional, or standard, way possible.

Subjective vs. Objective Meaning By interpreting a message externally to the agents involved, its meaning is interpreted more objectively. An interpretation made by the sender or the receiver would be more subjective.

Speaker's vs. Hearer's vs. Society's Perspective Despite the concerns of the previous two dimensions, a message may be expressed according to the perspective of the sender, the receiver, or other observers.

Semantics vs. Pragmatics Pragmatics focus on how the communication is used, including the mental state of the intervening agents and the environment in which they exist. These considerations are not shared by the syntax and semantics.

Contextuality For a message to be correctly understood it needs to be contextualized. To contextualize it is necessary to consider the agent's current state and its environment, and the evolution of the environment to that point. What can also affect the interpretation of the messages is the previous actions and messages of the agents.

Coverage The language selected for communication must give agents the ability to express all necessary meanings, but should not be too extensive to a point where it is not manageable.

Identity A message may assume different meanings depending on the agents involved, and the roles they play. The same may happen depending on how the intervening agents are specified.

Cardinality The meaning of a message may be different when transmitted to a single agent or broadcasted to a set of agents.

4.2.2 Types of Messages

There are two types of messages that are exchanged between agents [26]. If a message requires a response, then it is a query. To respond to a query, or to send a message that does not require a response, the agents use assertions. Agents may be divided into different types from the communication point of view. Depending on their types, they can send and/or receive assertions and queries. Table 4.1 shows the relations between the types of agents, and the messages they can send or receive.

Table 4.1: Agent types vs Messaging abilities [26]

	Basic Agent	Passive Agent	Active Agent	Peer Agent
Receive assertions	✔	✔	✔	✔
Receive queries	✖	✔	✖	✔
Send assertions	✖	✔	✔	✔
Send queries	✖	✖	✔	✔

4.2.3 Communication Protocol Structure

When designing a protocol for agent communication Huhns and Stephens [26] defined three levels in which a protocol is divided and a general data structure for the messages exchanged by the protocol.

The first, and lowest, level will specify how the agents interconnect; the second level holds the syntax specifications; the top level is where the semantics are, and where the meaning of the messages is specified.

As for the data structure for the messages, it has the following five fields:

- Sender
- Receiver (may be multiple receivers depending on whether the protocol is unicast, or multicast)
- Protocol Language
- Encoding and decoding functions
- Actions to be taken by the receiver(s)

4.2.4 Speech Act Theory

Communication between agents is often modeled after human communication. Speech act theory models the communication among humans into acts [26]. In other words, any utterance made by someone is an act. A communication act has three aspects:

- Locution - uttering a statement, or sending a message, in the software agent case
- Illocution - the meaning intended by the speaker
- Perlocution - the action that might result from the locution

An example could be: A tells B to close the door. The locution is when A requests B to shut the door. The illocution is that A wants B to shut the door. The perlocution is B closing the door.

To make communication more clear Speech Act Theory uses *performatives* to identify the illocutionary force. Some examples of performative verbs are: promise, report, convince, insist, tell, request, and demand. Illocutionary forces may be divided into five classes:

- Assertive - statements of fact
- Directives - commands in a master-slave structure
- Commissives - commitments
- Declaratives - statements of fact
- Expressives - expressions of emotion

To sum up, speech act theory helps the design of agent communication by creating the concept of illocutionary force and performative to identify the type of message and what is the action intended with that message.

4.2.5 KQML - Knowledge and Query Manipulation Language

The KQML is a language for communication among agents that is independent from the information beeing passed in the message. This allows the use of the same communication language for multiple contents. For the agent to be able to compreend the contents of message, it carries the information necessary for its understanding.

Bellow is the syntax of a KQML message:

```
(KQML-performative
                    :sender       <word>
                    :receiver     <word>
                    :language     <word>
                    :ontology     <word>
                    :content      <word>
                    ...)
```

First there is the performative of the message. KQML organizes its performatives into seven categories:

- Basic query performatives - evaluate, ask-one, ask-all, ...
- Multiresponse query performatives - stream-in, stream-all, ...
- Response performatives - reply, sorry, ...
- Generic informational performatives - tell, achieve, cancel, untell, unachieve, ...
- Generator performatives - standby, ready, next, rest, ...
- Capability-definition performatives - advertise, subscribe, monitor, ...
- Networking performatives - register, unregister, forward, broadcast, ...

The *:sender* and *:receiver* fields are self explanatory. The *:content* field is where the information the agent wishes to send is placed. The *:language* and *:ontology* fields are what allows the receiver to understand the message it has just received. The *:language* field indicates the language in which the message was written, and *:ontology* indicates the vocabulary of the symbols in the message. Together, the fields *:content*, *:language* and *:ontology* form the semantics of the message.

The KQML assumes asynchronous messaging, and for that reason features two more fields, *:reply-with* and *:in-reply-to* to be used by the sender and receiver, respectively. These two fields allow the linking of two messages with the use of a common identifier passed in the first message with the *:reply-with* field, and then used in a response in the *:in-reply-to* field.

Next there is an example KQML message. The language used is KIF, which will be detailed in the next section, the ontology is Block Worlds where the symbol *Block* illustrates a wooden block, and the content of the message is Agent1 telling Agent2 that a block A is on top of block B.

```
(tell
      :sender    Agent1
      :receiver  Agent2
      :language  KIF
      :ontology  Blocks-World
      :content   (AND (Block A) (Block B) (On A B))
      )
```

4.2.6 KIF - Knowledge Interchange Format

Ideally machine language should be like human communication and use natural language. However, implementing systems that use natural language is very complicated because the language is very vast, and it can be interpreted in different ways. The solution is to use a logic language which is simple to implement, and can express a wide range of concepts, including facts, definitions, abstractions, inference rules, constraints, and even metaknowledge (knowledge about knowledge).

In that line of thought Knowledge Interchange Format (KIF), a language based in first order logic, was developed with the intent of describing knowledge. KIF was intialy developed as a standard to describe facts in expert systems, databases, intelligent agents, and other similar systems. KIF is commonly used with KQML as the language that describes the

content of the KQML messages. The message used as an example in the previous section used KIF to express the contents of the message.

KIF specifies a syntax and a semantic with rules and definitions, and uses a prefix notation like the Lisp language. The logic used by KIF includes several operators like negation, disjunction, rules, and quantified formulas. The simple format and recognizable logic in this language allows it to serve as an intermediary in the translation of different languages, and also allows its reading by humans.

With KIF it is possible to express simple data, or more complex information, like the encoding of metaknowledge, or even program procedures, that the receiving agents can follow.

Next there is an example of a KIF statement, that means that *chip1* is larger than *chip2*.

```
(> (* (width chip1) (length chip1))
   (* (width chip2) (length chip2)))
```

4.2.7 FIPA ACL - Agent Communication Language

The Foundation for Intelligent Physical Agents (FIPA) is an international organization devoted to the development of intelligent agents, and is the responsible for the creation of the Agent Communication Language (ACL) [2].

This language has a syntax very similar to KQML. It is also designed to send messages independently from the context. Like KQML, the messages are in the form of performatives, that indicate the type of messages, and parameters that compose the message.

Next there is an example of an ACL message. This message is sent from agent i to inform agent j that it is raining today. The similarities between ACL and KQML are obvious.

```
(inform
  :sender    (agent-identifier :name i)
  :receiver  (set (agent-identifier :name j))
  :content   "weather (today, raining)"
  :language  Prolog)
```

Tables 4.2 and 4.3 lists the current communicative acts existing in the ACL language.

Table 4.2: FIPA ACL Performatives [3]

Performative	Summary
Accept Proposal	The action of accepting a previously submitted proposal to perform an action
Agree	The action of agreeing to perform some action, possibly in the future
Cancel	The action of one agent informing another agent that the first agent no longer has the intention that the second agent performs some action
Call for Proposal	The action of calling for proposals to perform a given action
Confirm	The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition
Disconfirm	The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true
Failure	The action of telling another agent that an action was attempted but the attempt failed
Inform	The sender informs the receiver that a given proposition is true
Inform If	A macro action for the agent of the action to inform the recipient whether or not a proposition is true
Inform Ref	A macro action for sender to inform the receiver the object which corresponds to a descriptor, for example, a name
Not Understood	The sender of the act (for example, i) informs the receiver (for example, j) that it perceived that j performed some action, but that i did not understand what j just did
Propagate	The sender intends that the receiver treat the embedded message as sent directly to the receiver, and wants the receiver to identify the agents denoted by the given descriptor and send the received propagate message to them
Propose	The action of submitting a proposal to perform a certain action, given certain preconditions

Table 4.3: FIPA ACL Performatives (cont.) [3]

Performative	Summary
Proxy	The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them
Query If	The action of asking another agent whether or not a given proposition is true
Query Ref	The action of asking another agent for the object referred to by a referential expression
Refuse	The action of refusing to perform a given action, and explaining the reason for the refusal
Reject Proposal	The action of rejecting a proposal to perform some action during a negotiation
Request	The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act
Request When	The sender wants the receiver to perform some action when some given proposition becomes true
Request Whenever	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again
Subscribe	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes

In table 4.2.7 there is the complete list of message parameters in the ACL language.

Table 4.4: FIPA ACL Message Parameters [2]

Parameter	Category of Paramenters
performative	Type of communicative acts
sender	Participant in communication
receiver	Participant in communication
reply-to	Participant in communication
content	Content of message
language	Description of Content
encoding	Description of Content
ontology	Description of Content
protocol	Control of conversation
conversation-id	Control of conversation
reply-with	Control of conversation
in-reply-to	Control of conversation
reply-by	Control of conversation

4.3 Final Considerations

The work presented here led to an evolution in the way agents interact among each other. However, like in the previous chapter, each method has its ups and downs and improvements can be made. In the communication field there is allways the goal of transmitting as much information as possible, using as few messages as possible. For coordenation, it is important to develop methods that can overcome any situation that presents itself to the agents using cooperation as a means to achieve the common goal. Fortunately there has been an increase in the research of new strategies and new architectures for agent cooperation, and new protocols for agents to communicate properly which will eventualy bear fruit.

Chapter 5

Rescue Agent Strategies

The work presented in this thesis was developed a part of the portuguese RCR Team FC Portugal [34]. FC Portugal began to participate in RCR Competitions in 2005. The first agents implemented by FC Portugal were developed in C++ [35], a laguange also used in the development of this work. This would also ensure the usage of a more efficient language, as well as greater reuse of the source code made by previous teams of FC Portugal. Nevertheless, the number of teams that still develop their agents in C++ grows shorter by the year. The vast majority of teams, includings the top teams, use Java.

The Robocup Rescue project, and its Agent Competition in particular, engulfs many problems of Artificial Intelligence. However, the main focus of this thesis is on the aspects of world modeling, map searching, agent cooperation, agent communication, and rescue planning. All these aspects are connected to each other, and because of that, the development of one aspect will eventualy leed to the development of another. To build an efficient schedule for rescuing, a good understanding of the world is vital, which leads us to World Modeling. To create a good World Model as quickly as possible the agents will have to search the map and communicate to inform each other on their findings. A good search uses cooperation to improve its efficiency. Cooperation is also vital for a successful rescue task. Coordenating a team of agents is facilitated when there is a good communication model to support it. Previous teams of Robocup Rescue competitions have used several strategies, some of their own design, others adapted or based on the work of other teams. The reader should keep in mind that the work developed for these competitions is a collaborative effort, and the proof of such is that the source code for the three top teams of each year's competition must be shared with the remaining community. Also, since the work is ultimately a collaboration, some teams choose to improve only one type of agents, or a specific characteristic of the agents, like communication.

Next there is a collection of methods and techniques used by developers throughout the competitions in the various aspects of agent development.

5.1 Communication

Coordination can be done implicitely or explicitely. In the latter case it will require communication. Communication is also vital to the creation of a World Model as complete as possible. When developing RCR agents communication is a problem all of its own. Its uses are many, but so are its limitations as was explained in section 2.1.8. This leaves developers

with the problem of having to choose which messages are truly important, and how to pass on the information.

To deal with these limitations the Impossibles team of 2006 made several suggestions [17]. First it is suggested a scheme of priorities in order to guarantee that important information is sent first. This deals with the problem of the number of messages that an agent can receive. It is also proposed two different strategies for communication. In the first one the center agent sends one message to a specific agent. This way ensures that the agent receives the data necessary, and also that the agent is still capable of receiving more messages. In the last one the center sends 3 messages to all agents. Doing so it is assured that the agents possess the same information, but it almost exhausts the receiving ability of the agent (the fourth message slot is left for hear messages).

Another strategy that deals with the limitation of messages an agent can read is employed by the CSU Yunlu Team of 2006 [41], where the messages are filtered in the centers in order to reduce the messages sent to field agents. This also limits the number of lost messages. To ensure that the platoon agents are able to receive the data coming from the centers, they do not hear messages from teammates.

This centralized strategy was also employed by the IUST team, in 2006 [24]. To ensure that the platoon agents are able to receive the data coming from the centers, they do not hear messages from teammates. Another problem attacked by this team was the maximum message size, 256 bytes. Since that size cannot be increased, they opted to shorten the length of data sent in the messages. For example, instead of sending the complete ID of a building, the message includes the index of the building in the building array. This reduces the necessary size from 4 bytes to 11 ou 12 bits. In the event that the 256 bytes will not be enough, the center is free to send more messages, to the agents since they do not receive messages from nowhere else (except natural voice commands). It can send a total of 256×4 bytes. This team extended their model to the communication between centers of different types. When a center transmits the data to the platoon agents, it will also send to the other centers. The model uses three cycles to pass the information around, like figure 5.1 shows.

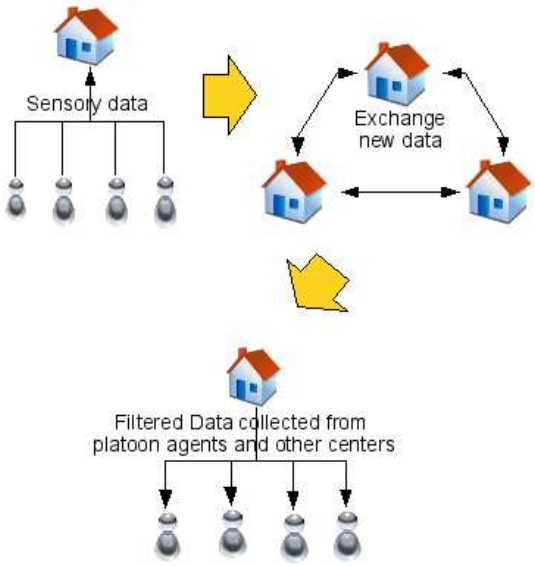

Figure 5.1: Communication method employed by the IUST team of 2006

The first cycle sees the platoon agents send their data to their respective centers. The second cycle is for the center agents to exchange information with each other. The third cycle will be used to transmit to the platoon agents the information gathered between the centers. This way it is assured that the platoon agents receive the data in no more than three cycles.

To prevent the centers from getting into infinite cycles of exchange the same information, the messages indicate the cycle to which the data refers to, and the values stored by the agents also has the values according to the cycles to which they refer to. For example, the fire center informs another center that the fieriness of a building is 3 at cycle 100. That center will transmit that data to its platoon agents and to other centers, including the Fire Center. To prevent the fire center to resend the information that it has just received, it will check if the fieriness value of that building has already been set for cycle 100. If it has it will not resend that information.

While communicating, the protocol of communication is as important as the structure of the messages being sent and received. This particularity was focused by the GoldenKnight team in 2007 [28]. Each message has the following format:

$$\text{M} = \text{<Performative, Task<Purpose, ObjectID, Utility>, RecipientID>}$$

$Performative$ indicates the type of message, whether it is a Request, Response, or Assignment. The $Task$ segment is a tuple where $Purpose$ refers to remove a blockade, extinguish a fire or save a civilian. $ObjectID$ refers to the target ID of the task, a road, building, or location of the civilian, respectively. The $Utility$ field in the $Task$ tuple represents the utility of a given agent in performing the specified task. This utility is given by a utility function and it is calculated by the agent. For now that field has no specific use, but it could be useful in the future for a central decision making agent. $RecipientID$ is the ID of the destination agent.

The Poseidon team [19] used in the 2006 competition a Communication Infrastructure. The intent was to develop a communication section that was as independent from the remaining as possible. This would allow to perform modifications in the different sections without having to worry with the others. The resulting architecture can be seen in figure 5.2.

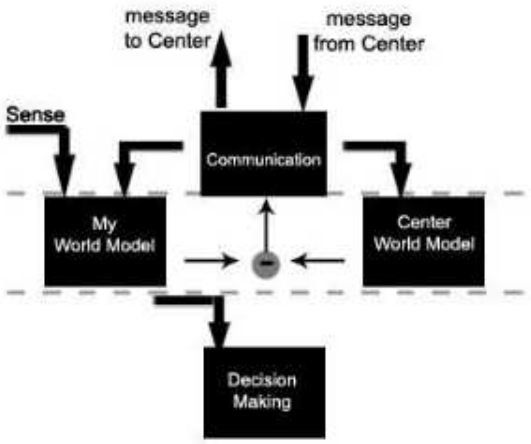

Figure 5.2: Poseidon Agent Architecture [19]

Another concern of this team was the number of messages that an agent is allowed to hear in a single cycle. As was explained in section 2.1.8, the center agents can hear up to $2 \times n$ messages, where n is the number of platoon agents of the same type, and the platoon agents can hear up to 4 messages. The number of platoon agents normally surpasses 4, which means platoon agents can't communicate with all teammates. And they should also leave room to hear screams for help from neighbouring trapped civilians.

Another question was the number of messages that can be sent to other Centers. The number of platoon agents of each type may be different, which means that the number of messages that a Center can still hear after receiving the updates from its respective platoon agents is also different. With the knowledge that, after receiving the messages from its platoon agents, the Centers can only receive n more messages, a method was developed to prevent Centers from failing to hear messages sent by the other Centers. First the minimum n of all three types is selected.

$$\text{MIN} = \text{minimum(F,P,A)}$$

Where F is the number of Firefighters, P the number of Policemen, and A the number of Ambulances. After receiving the sensory information from the platoon agents and updating the data in the World Model the Center will send to the remaining Centers the new data. All the information may not fit into a single message, so the Centers will have to segment it into a set of messages. The maximum number of messages that can be included in a set is given by the equation: $\frac{MIN}{2}$

Considering the previous equation the Centers will receive up to $n + 2 \times \frac{MIN}{2}$, where $\frac{MIN}{2}$ is double because there are 2 centers sending that same amount of messages to each Center. This is a homogenuous method, as it does not differentiate the number of messages that can be sent to the various types. Figure 5.3 displays the communication method described above.

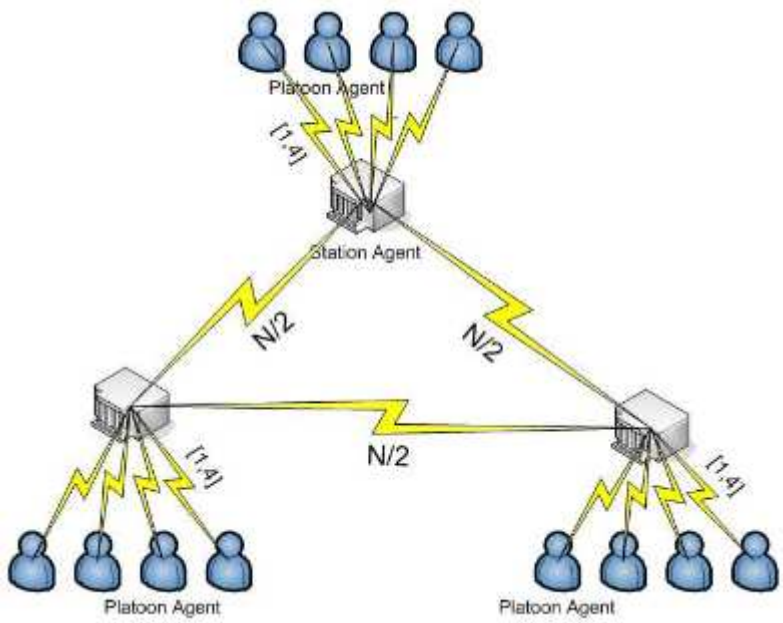

Figure 5.3: Poseidon Communication Structure [19]

So far it has only been assured that the agents receive their messages, albeit network problems. However, the number of messages allowed still limits greatly the amount of information that can be sent. For the centers to be able to send the new information to the agents, this team considered 4 strategies:

1. The Center sends four messages to a single channel which all platoon agents are listening. The Center would have to determine what would be the best information to send to the agents. Agents may receive data already known to some of them, and see their hearing capability blocked to receive cries for help from Civilians. This method does, however allow the Centers to send large sets of data;

2. The Center will send a personalized message to each Agent. The Center will have to see what the agent already knows, and then send a message with only new data. This implies that the Center should know the World Model for each agent up to that moment, besides its own. It will also have to select which new data to send in the event that not all of that data can be sent in a single message. This makes this method unefficient;

3. The third strategy joins the previous two. The Centers will send four messages to groups of four agents. This way the amount of repeated information is reduced, and the Center has more resources to send greater sets of data. Nevertheless, the platoon agents still can't here cries for help from Civilians because their hearing capability has been exausted.

4. The last strategy sugested by the Poseidon team is the most complex one. It divides the new data into Information Units (or IU). Those IUs will be included in messages, that will later be sent to the agents. One message will be sent to four agents. To determine how the IUs will be placed in the messages, an auction method is used. The IUs have a value and the agent that is the highest bidder will receive that IU in his message. The agents evaluate the IUs by calculating the priority using a specific function. The team developers recognize that this solution is not the best one, and also sugest the use of a dynamic programming solution

Due to the complexity of the fourth strategy, it was not implemented. For the inverse reason, the team opted to implement the second strategy.

Remembering what was said at the start of this chapter about some teams developing specific aspects of the agents coordenation the modularization used in the architecture also helped with the aforementioned colaboration, with the ITANDROIDS team using the Poseidon's communication strategy in the 2007 competition [54].

5.2 World Model

Knowledge is power. The greater the knowledge, the greater the power. In the case of RCR agents, power refers to the ability to make good decisions. The best way to do this is to gather has much useful information as possible from the agents' environment. The information gathered by the agents is colected in the World Model, which is the world as the agents perceive it. It is from this collection of information that the agents will make their decisions. Besides saving the agents' sensorial information the world the World Model can

also keep a map of the city, with its roads, buildings, blockades, and references to other agents like centers, teammates, civilians.

The World Model grows as the simulation continues and the agents can update their knowledge base using the sensory information available to them. That information includes visual and hearing data. That data is limited and it is only referent to the agents position at the start of the cycle. This leads to very incomplete information, and the agents, should rely on their teammates to build a more extensive World Model using communication. Some teams implemented a third source of information. It won't exactly provide unknown information, but using Action Prediction techniques it is possible to estimate with some precision the effects that actions performed by other agents will have on the World Model. Figure 5.4 displays the main sources of information of the World Model.

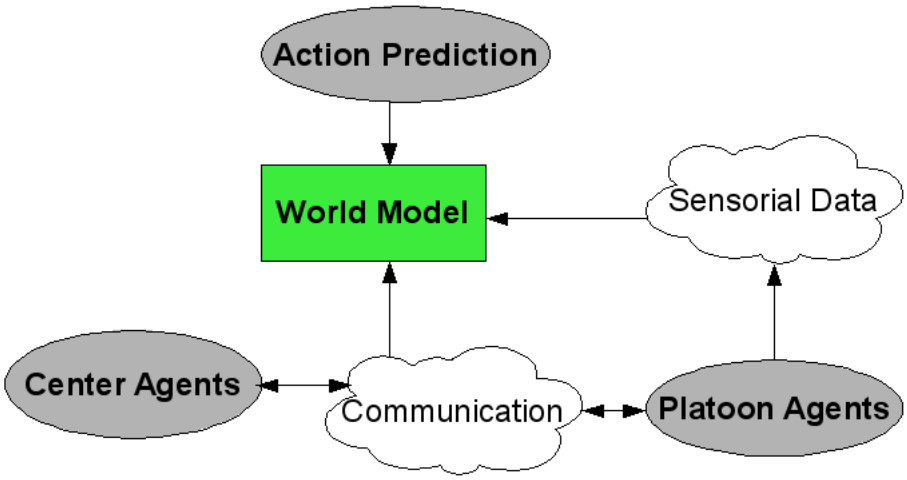

Figure 5.4: Sources of information for the World Model

The IUST team of 2006 [24] extended the features of the World Model structure provided by the YapAPI to allow the storage of all dynamic properties of the objects, such as building fieriness. This upgrade was needed because the default structure would only save the last known value of the dynamic properties. With the complete list of values for each property the agent has a better notion of the pace at which a building is burning, or a civilian is losing HP. It also eases the implementation of online learning methods. In section 5.1 it was described the communication model used to update the World Model of the agents of team IUST. That model provides information to the platoon agents with a delay of no more than three cycles. This delay, although short, is still capable of creating differences in the knowledge of the agents which may, or may not affect the cooperative efforts of the agents to some extent. To diminish the effects of this the developers implemented a second world model, the Virtual World Model. This is the model that the agents will use to make their decisions and is only updated with the data provided by the centers. This way it is assured that all agents have much more similar World Models, and their considerations on the teammates action is much more reliable. The real World Model is updated with the sensory information and the data sent by the centers.

The agents developed by the Impossibles team of 2007 [18] create their World Model using cooperation and communication. In a three stages model, the agents send the new data to their respective center, which will then send the data to the remaining agents, and finally

send the complete data to all agents. To deal with limitations in the size of the messages, the data is minimized as much as possible by eliminating repeated data and/or old data. This method is used by other teams such as the 2006 IUST team.

Besides the data regarding sensory information, and kernel data, the world model can save data specific to each type of agents. That is the idea followed by the GoldenKnight of 2007 [28]. In the case of Ambulances, an ambulance team can also keep in the world model a list of the civilians assigned to it for rescue by the Ambulance Center. The Ambulance Center will keep a table that records rescues in execution, or waiting to be executed. The table includes, for each Civilian, its ID, the ID of the ambulance assigned to it, and the value resulting from the utility function of the Ambulance assigned to the rescue.

5.3 Cooperation

In order to achieve the goals set for the simulation it is wise to use all available resources. It has already been discussed the World Model that agents create. Another resource available to developers is the number of agents available. By developing an efficient cooperation model, team developers ensure a more intelligent use of resources. This will lead to higher scores, which translate into more civilians saved and less property damage.

It is common to use the centers of each type of agent to act as a central point for information gathered and decision making. In the Team Description paper of the Impossibles team of 2006 [17] two approaches are discussed: Centralised and Distributed agents. In the centralised case it would function as mentioned. In the distributed model the agents communicate with a center that aggregates the information gathered individually, and then passes it on to all agents. The decisions are made by the agents. The distributed case is preferred for several reasons. First, in some simulations there are no centers, or no communication. Second, due to communication limitations, the distributed model acts more quickly. Lastly, the distributed model is more robust because it does not depend on a single point, which can suffer from any kind of error or bug.

In 2006, the NITR team aimed at developing agents giving special atention to cooperation [50]. As it is mentioned in [50] a good result on a simulation does not imply a good cooperation between the agents. It could also mean that the individual ability of the agents was good. To analyze the behaviour of the agents and determine wether the agents are cooperating well or not, the NITR team developed a tool that would help them to know which aspects of the agents needed to be changed in order to obtain a good cooperation model. First they identified some Cooperation Indicators. Using a Log Analyzer that would output several data that might measure agent cooperation, they determined which, among those provided by the Log Analyzer would correlate strongly with cooperation. The indicators chosen were: Extinguish Rate, Rescue Rate, Extinguish time for same target, Rescue time for same target.

The canadian team DAMAS [40] presented in the 2004 competition a team of ambulance agents, center included, that used a centralised model. The idea was to reduce the amount of messages exchanged, and with it the penalty of lost messages. The platoon agents first send the sensory data to the center, which will process the information and return the list of agents to rescue. All ambulance teams will then proceed to the rescue of the agents. After successfully unburying a civilian, the ambulance with the lowest ID will transport the civilian to the nearest refuge, and will communicate with the center to anounce that the civilian is safe. This cooperation method is exemplified by figure 5.5.

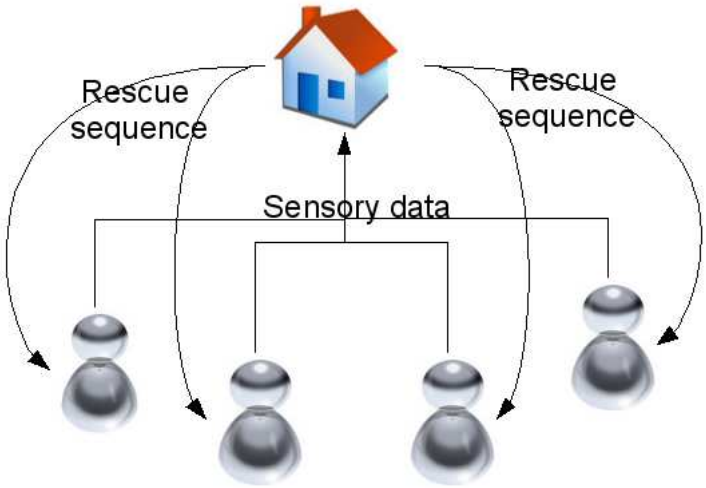

Figure 5.5: Centralised strategy used by the DAMAS team

5.3.1 Search

Search plays an important part in the outcome of the simulation and, as such, deserves much attention. Several methods have been implemented by different teams.

In 2006 and 2007 the Impossibles team [17, 18] calculated a minimal set of nodes that would allow to see the map completely. The agents would first model the world into a graph, where the vertexes would be the buildings, and each building would be connected to the buildings in its field of vision. This is a Dominating Set problem, of NP complexity, and there is no polinomial solution. In this case, if the set of nodes is being calculated online, then an approximation is required. The set can be calculated offline. However, in a map with a great number of buildings, the algorithm can take a large amount of time to reach a solution, and an approximation must also be used.

In 2004, DAMAS team [40] would only perform a search after rescuing all known buried civilians. It would then create a list of buildings that were inside a perimeter of 30 m around a point where a scream for help was heard. The agents would then search for buried civilians in the buildings included in that list.

The RoboAkut team [6] in 2008 used different strategies for different situations. If the simulation allows communication then the search is coordinated between all agents using a market plan. If there is no communication, the agents divide the map and assing at least one agent per region. The agents decide which buildings to visit first by calculating the probability of it being a likely location of buried civilians. When the agent is in a no communication scenario and it is exploring its assigned region, it behaves differently. While exploring it tries to maximize the distanced traveled, instead of performing its search based on buildings.

At times an agent must choose whether to continue the search or to proceed to its specific task (rescue, fire control ou clearing blockades). The agents implemented by the 2006 Aladdin team [43] decide what to do using priorities. Those priorities are calculated using Bayesian estimation techniques, a technique also used by the ITANDROIDS team in 2007 [54]. To perform the search itself, agents are allocated to specific areas of the map. Since those agents can, at any time decide to perform their specific function, the allocations must be continuously updated. Furthermore, this team determines specific unexplored areas which

are more suitable to have the types of events that the agents search for. Those areas are determined using estimations and the information learned by the agents during the simulation. The determined areas are also prioritized according to the results of risk analysis techniques that take into consideration the uncertainty of the estimates and the penalty of not discovering the events. In the 2007 competition, the Aladdin team [20] had their agents partition the map before performing the search. The division of the map was made using the k-means clustering algorithm.

Some teams use multiple solutions for the search problem. For example, the IUST team, in 2006 [24], used three strategies for the routing problem. The goal set is to calculate the optimal path in a half cycle. Sometimes it is not possible to obtain the best path in that period, so a trade-off must be made between taking longer to calculate the path, or to follow a non-optimal path. The first solution is the usage of a tool provided by the YabAPI, the API available for Java developers. That tool is a greedy search method, where the paths returned may not be optimal due to the costs associated to the roads. It sets the cost for open roads as 1, and $Java.MAX\_VALUE$ for unknown roads, and $Java.MAX\_VALUE^2$ for blocked roads. The problem here is the unnecessarily large value for the unknown roads. The solution implemented was to change the cost of known and unknown roads to $road\_length$ and $3 \times road\_length$ respectively.

The next method for routing developed by the IUST team used the A* algorithm (a algorithm widely used by other teams such as GoldenKinght [28]). Altough better in theory, this proved to be a worst solution than the greedy algorithm. The reasons appointed for this were the heuristic function that must be used with the A* algorithm and the need to execute the algorithm for each pair origin-destination.

The last method employed was the Floyd algorithm. This algorithm was found to be of very little pratical use due to the heavy processing and time consumption. The only use found for this method was to calculate the all paths for each pair of main nodes (which are defined as a node that has one or more than two connections). These paths, however, would only be used after the policemen cleared all roads.

For the task of searching for trapped civilians the IUST developers used Policemen to help the Ambulances. This would only happen after the main roads were cleared. The map would then be divided into pie pieces of the same number as half of the remaining active Policemen. Each piece is assigned to two Policemen. One with the task of searching for civilians, and the other with the task of clearing any remaining roadblocks. The divisions of the map are updated in the event of the death an agent. The agent assigned to search each partition will visit the unseen buildings of its assigned partition. When there are no more blockades in the city, the map will be divided once more in pieces equal to the number of active Policemen. Then each Policemen will have a piece assigned to it and they will all continue the search for trapped civilians. After all buildings have been visited the agent will visit the civilians found in order to have updates on their status. A similar strategy is applied to the Ambulances while searching. The main difference between the two types of agents is the requirement defined by the developers that the Ambulances stick together. To prevent them from getting in the way of each other a different building is randomly assigned to each Ambulance. Since the search performed by the Ambulances is cooperative, it uses the Virtual World Model detailed in section 5.2. Unlike the Policemen who perform the search independently, and use the real World Model.

The Kshitij team in 2005 [33] also used Policemen as their main agents for map searching, with support from the Ambulances and Firefighters. The search for the agents implemented

by this team is performed after a division of the map is made. Each type of agent divides the map into districts and each district is assigned to an agent. After the district assignment has been performed, the agent can initiate the search. Each agent will then select a target location in the map where the distance to that location is short, and the number of observable locations is large. The search, however, has further considerations. The Ambulance agents give priority to unexplored areas surrounding burning buildings, whereas Firefighters prioritize search in areas with trapped civilians. To determine the target location for search a utility function was devised which takes into consideration the travel cost, the number of observable locations, the distance to the closest fire and the distance to the closest trapped civilian.

Although most teams choose to use Policemen as the agents responsible for most of the search, the ITANDROIDS team of 2006 [11] decided to give this task to the Ambulances. This strategy was developed because the developers of this team found the task of rescuing civilians to be too complex for the rewards it gave. The search itself does not differ greatly from other strategies implemented. The map is divided into zones, and the Ambulance Center assigns a zone for each Ambulance. The Ambulance will then move to a node inside that area and continue to explore by itself. If the agent strays from the assigned zone to another that has already been explored, the Ambulance Center will assign a new zone for the agent to explore. When the search reaches a point where it is considered unnecessary, whether it si because the city has been completely search, or no relevant areas are left to search, the Ambulances will begin to Rescue civilians.

5.3.2 Rescue Planning

When developing Ambulance agents the goal is to develop efficient strategies for search and rescue. As such, after having identified trapped civilians while performing the search, the agents are now left with the task of rescuing all those civilians. Although sometimes it is not allways possible to save all of them, the effort of saving as much as possible must still be done. The best way to do that is to create a Rescue Plan that tells the Ambulance agents which civilians they should try to rescue, in what order, and how many agents should try to rescue each civilian.

The DAMAS team, in 2004 [40], placed at the top of their priorities the rescue of fellow platoon agents. If there were no injured agents in need of rescue the agents could then proceed to the rescue of civilians. To that end they developed an algorithm that would provide a list of known injured civilians by order of priority. The algorithm, greedy in nature, attempts to save the maximum number of civilians. It takes into consideration the time expected for the agent to die, the time necessary to travel to the civilian, rescue it, and transport it to the refuge. Using these parameters, the rescue list is calculated by placing in first place the agent that maximizes the number of agents that can be saved after that one.

A more complex and generic method was developed by the RoboAkut team in 2008 [6]. It uses a queue system, where the queue holds subqueues ordered by priorities. The agents begin by executing the tasks present in the highest priority queue, and continue to the next queue when the current one is empty. The priority assigned to a given task is first determined by default values, and will change with time. The calculation of a priority update is done considering other tasks of the same type.

The assignment of tasks to the agents can be performed in two ways. What determines the method to be used is the type of task, i.e., if it is a common task, or a rare one. For each method an auction algorithm and a market algorithm, respectively, are employed. The

auction is performed by the Center, preferably, or by the available agents in case a Center is not available. To prevent the loss of cycles, the auction takes place while the agents perform their tasks. In the specific case of Ambulances, the planification of the rescue tasks depends on the expected life time of the Civilian, and its proximity to a disaster area. According to those parameters the identified Civilians are classified into four categories: helpless, will die soon, will die and will live. The classification is in charge of a Bayesian classifier that works with Civilian data reduced to one dimension using PCA.

Like the DAMAS team, RoboAkut also make the rescue of injured platoon agents the top priority for Ambulances. In the event of a platoon agent getting hurt, Ambulances drop their present task, unless they are already in the process of rescuing a Civilian. An example of priority queues for Ambulance agents can be seen in figure 5.6 with the first queue, reserved for platoon agents, empty.

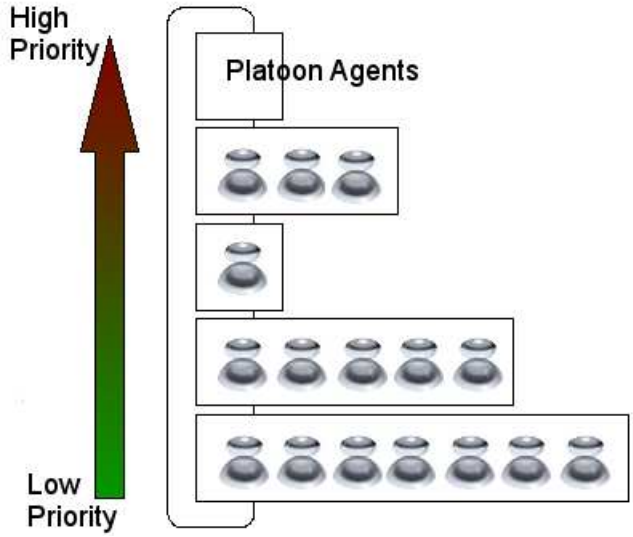

Figure 5.6: Ambulance agents rescue priority queue

A new approach for Rescue Planning is given by the Kshitij team of 2005 [33]. The agents group civilians by proximity and name it a civilian site. After grouping all civilians, the Ambulances will proceed to the rescue on the site that results in the rescue of more civilians in less time. By reducing the number of civilians that the Ambulances need to consider also reduces computaional effort.

To create the rescue plan the CSU YunLu team of 2006 [41] used machine learning to create predictions on the expected time of death of all known civilians. This would allow the classification of civilians into victims and survivors. This last class exists because not all civilians will see its HP reach 0 until the end of the simulation. Some lose HP very slowly, and others don't lose HP at all, even though they are trapped. First a tool was created that would generate datasets by running several simulations using various maps. The datasets contained the values for health and damage for each civilian at each cycle of the simulation. For regression and classification the developers used WEKA machine learning tool. For classification it was used the C4.5 algorithm (which uses decision trees), and Adaptive Boosting (Ada Boosting) for the time regression task. After the datasets where gathered and the regression and classification tasks were performed the resulting regression

trees were evaluated to ascertain their confidence. The accuracy of the predictions was given by the difference between the observation and the civilian's actual time of death.

Like other teams, the 2006 IUST team [24] determined the civilian to be saved using a system of priorities, which took into consideration several factors, such as: expected time of death of the victim; proximity of the victim to a rescue agent; proximity of the victim to a fire; distance of the ambulance agents to the victim; distance of the victim to the nearest refuge. After completing the list of priorities the agents will proceed to the rescue. If a civilian is saved or it is not possible to reach it, then the agents move on to the next on the list. The Ambulances also have a rule which doesn't let them separate from each other. By sticking together they can all tackle a rescue simultaneously, performing the task in less time.

Most teams try to help primarily civilians with less HP. However, Impossibles team for the 2006 competition [17] planned its rescue table differently. They tried to maximize the number of rescue actions performed by the agents. The assumption here is that a Civilian that will survive the next 20 cycles and needs 5 Ambulances to be saved in time to be carried to a Refuge, is as valuable as a Civilian that will survive the next 100 cycles but only needs one agent to rescue it. With this assumption when building the rescue plan, first came the civilians that only needed one agent to rescue them because it would leave more agents free to perform more rescue actions. In second place in the rescue plane came the civilians that needed two agents, and so on. To build the planning efectively it was necessary to determine the time of death of each civilian, knowing its current HP and damage. From there it was also possible for them to calculate a function that would give them the number of needed ambulances through time.

This last strategy is also adopted by the Poseidon team of 2006 [19]. It has the advantage of being possible to execute even wihtout a Center to make all the decisions. It does, however, require that the World Model of each agent is very similar, and that the location of each ambulance is known to all team agents.

As was explained 5.3.1 the 2006 ITANDROIDS team [11] prioritized map searching over rescuing civilians, for Ambulances. After the search is considered useless the Ambulances will be notified by its Center to begin the rescue of Civilians. The known civilians will then be ordered by their priorities which will be calculated according to their buriedness, where the main concern is to minimize the time spent with each civilian, and HP with the thought that it is counterproductive to save a civilian that will no live until being rescued.

This team devised a new strategy for the following competition, in 2007 [54]. The calculation of priorities was the same as the previous year, but the agents are not ordered by their priorities alone. They are now divided into four groups:

1. High priority (civilians in need of urgent rescue)
2. Intermediary Priority
3. Low priority
4. Low Viability (civilians whose rescue would be too costly)

The division of civilians into each group is done based mainly on the rescue time and the amount of HP left for each Civilian. As an example, given a civilian with a great number of HP and a short time necessary for its rescue, that civilian will be placed in the Low Priority group. By dividing the civilians into groups, the processing required to make decisions is

smaller, because the number of agents was reduced. The Ambulances will focus on only one group at a time, from the first to the third group. The Civilians placed in the last group will not be rescued. The Ambulances change their target group when the highest Civilian HP of the group being rescued drops below a critical level. There is a case in which the order established for the rescue might be interrupted. It will happen in the event of a fire spreading in the direction of a Civilian who can still be saved.

Some teams also build scheduling functions for Rescue Planning which attempt to maximize the final score, like the Aladdin team in 2006 [43]. More specificaly, the method used calculates a deadline for each civilian that represents the latest time at which the civilian can be successfuly rescued, taking into consideration the number of ambulances assigned to the rescue. This deadline may change in case of firespread. The ambulances will attempt to save the civilians in the order of their respective deadline. The civilians with shorter deadlines will be saved first. In case there are multiple civilians with short deadlines then the number of ambulances will be evenly distributed among those civilians. As the team explained in their 2007 Team Description Paper [20] this method has some problems. For starters, the deadline for each civilian is calculated with the assumption that all ambulances are available to perform the rescue. This is very optimistic because agents might be trapped between roadblocks, occupied with other tasks, or they might be dead. Also, when allocating the ambulances to the civilians the travel times are calculated with very low precision, which can result in having an ambulance taking too long to achieve their destination and arriving too late. Also, the information gathered about the civilians and used to calculate their respective deadline had a large degree of uncertainty, which resulted in untrustworthy estimates.

When planning the rescue of civilians there are two main questions to be answered: which civilians to save first (scheduling) and which Ambulance should save which civilian (agent allocation). Some of the teams mentioned previously chose to worry only about scheduling and assigned all available agents to each rescue. The GoldenKnight team [28], however, decided to investigate both questions. The scheduling and the agent allocation will be performed by the Center, but the source of the deciding factor is different. Scheduling will be based on the priority for each rescue mission, and agent allocation will be based on utility values provided by the agents. After receiving the utility values for a given task, the Center will then determine the best allocations by using a ContractNet based implementation. The overall behaviour of the Ambulance Teams developed by GoldenKnight can be seen in figure 5.7.

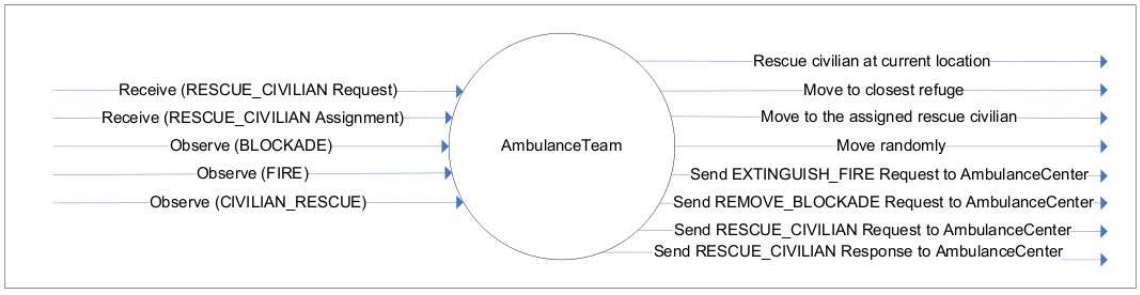

Figure 5.7: Behaviour of a GoldenKnight's Ambulance [28]

A method that has been developed by some teams, or at least planned, but hasn't seen a wide implementation is the Genetic Algorithm. In 2005, the ResQ Freiburg team developed a rescue sequence planner based on Genetic Algorithms [32]. Like other rescue planning

methods this one attempted to maximize the number of living civilians at the end of the simulation. To do so, it calculated the number of survivors of a given rescue sequence using the estimated time needed for rescuing each civilian and the estimated life time of each civilian.

To estimate the time required to save a civilian a linear regression was used that was based on the buriedness of the civilian and the number of ambulances sent to its rescue. The travel time was calculated using averages of travels in previous simulations. The idea was to avoid calculating the exact cost of the distance traveled, that could reach $O(n^3)$ using the Floyd-Warshall algorithm, that was described in section 3.1.4.

Initially they used a greedy method that simply ordered the civilians by the time necessary to reach and rescue them, but it provided unsatisfying results. The new idea was to obtain a rescue sequence, $S = \{t_1, t_2, ..., t_n\}$, of n targets, and evaluate that sequence using an utility function, $U(S)$, that would return the expected number of surviving civilians. Because the number of civilians on a map can be quite large (some maps can have as much as 80 civilians) going over all $n!$ sets would be an imense computational effort. The best solution was to use a genetic algorithm (GA) to optimize the rescue sequence.

The GA starts with solutions provided by heuristic functions, such as sets where the civilians are ordered by smallest life time, or shortest time to rescue. When using genetic algorithms the initial solutions (or populations) are normally random. But by providing solutions that are already close to an optimal solution it might speed up the process. Next comes the fitness function that will evaluate the existing solutions. The function used for the GA in question was the same utility function described previously, $U(S)$. To improve the quality of the solutions an *elitist* selection takes place where it chooses the best solution found, and maintains it in the population. This also garantees that the solutions in the population are always as goog as the ones provided by the heuristic functions. Other parameters used in the GA were simple one-point-crossover strategy, a uniform mutation probability of $p \approx \frac{1}{n}$, and a population size of 10.

In a simulation cycle, an Ambulance Center can obtain 500 populations. From those populations, the best sequence is selected and radioed to the platoon agents that will begin the rescue activities.

The dynamic nature of the world places a problem in creating a rescue sequence. New civilians may be encountered, their status changes with time and can be updated by visiting the civilian again, or by predicting its status. When using predictions it is possible to control their effect on the planned sequence by assigning them confidence values. Only if a prediction is within a certain interval will it change the civilians status, and consequently alter the rescue sequence. For the civilians that are found by other agents, they will be evaluated for emergency situations. Civilians fall in this category if they are not rescued in the next round. If a civilian is considered an emergency then the Ambulances will go to its rescue unless if leaving the current civilian being rescued would result in its demise.

Figure 5.8 shows the results for both methods tried, the greedy, and the genetic algorithm.

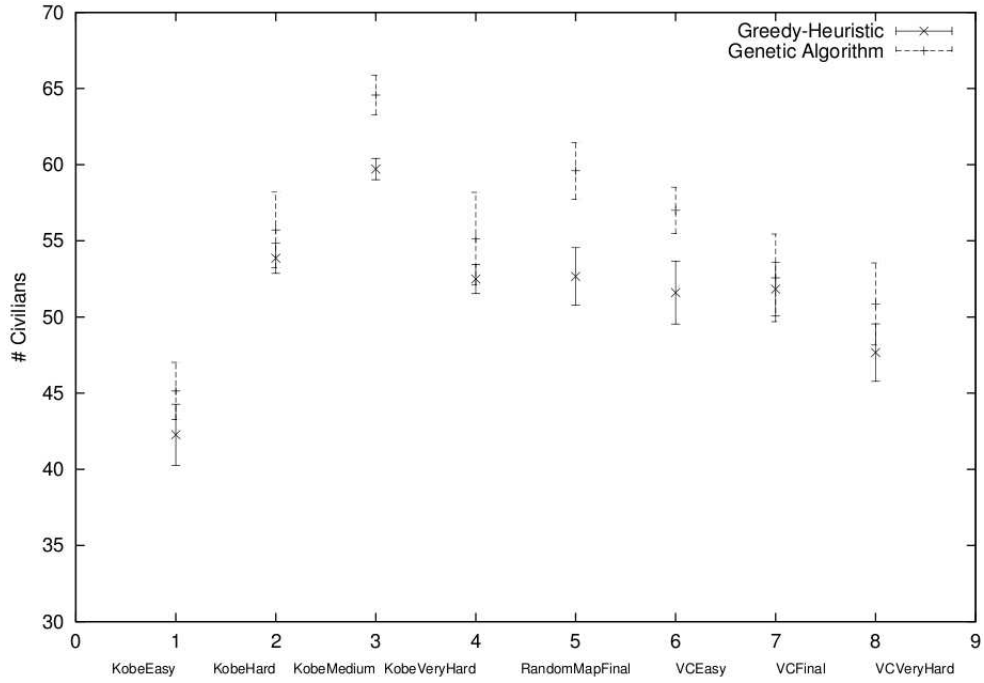

Figure 5.8: Results comparison between greedy method and method using genetic algorithm [32]

5.4 Final Considerations

All the strategies presented here have their positive and negative sides. Also, to improve a certain aspect of a strategy it usually costs a downgrade on another aspect, which leaves developers with some difficult choices. Many other solutions not presented here were not adopted by teams due to computacional costs. Because RCR Competitions, although growing in fame and prestige, are still far from presenting a valid solution for the problem set by the project the support for the teams is very limited, or even inexisting. This is a problem that affects all leagues, and not only the Simulation League, or even RCR. With bigger support from universities, governments or private companies, the evolution registered each year would be much greater. Not only in the development of the agents, but also in the development of the simulation infrastructures, which, although fairly complex at the moment, still have lots of room for improvement. Nevertheless, these strategies may be useful in other areas, where the complexity is greatly diminished. In fact, the method that might still be long from being able to see a reflexion in real life is that of the Rescue Planning. This is because it can be very complex and subjective to evaluate the medical condition of a person, the basis for most Rescue Planning algorithms. It requires data that is not available for rescue teams, like a medical history, assuming there is one and that it is comprehensive and updated. Also necessary for the evaluation of a human being it is necessary to perform some exams in order to have an good idea on the extent of the injuries suffered by victim. These exams, even if accurate, might be inconclusive. Besides determining the condition of a victim, real rescue

teams save victims taking into consideration the environment of the victim. If a person is in good health, but its surroundings look like they might collapse then they will save that person immediately, instead of first traveling through the disaster area, and then creating an ordered list of the victims to be saved, like some strategies implemented.

Fires are a different matter altogether. The parameters that are required to take into consideration for a prediction of the evolution of a fire to be accurate are imense. It is necessary to know a great deal of information about the building: the construction materials, the structure, what occupies the building (housing, office, school), if there are any kinds of materials in the building that might fuel the fire, etc. It is also necessary to have detailed information about the surroundings of the burning building, and the evironment. All this information, although vast in amount and type can be arranged, as long as cities maintain their records updated and in a format that is accessible to the Rescue System. Should the simulator have access to all that information, and if armed with a comprehensive physics simulator, and a powerful computer system, it might be possible to calculate a prediction of how a fire raging inside a building might evolve with a fair degree of reliability.

In conclusion, it is safe to say that although the road left to travel is very long, the results and effort of researchers from all around the globe leave us with some optimism.

Chapter 6

Agent Development

6.1 Agent Development Tools

One of the challenges of development under a simulation environment is to discover bugs and other errors which are not obvious. In a complex environment such as the RCR world it makes the matter all that more challenging. The RCR simulator records most of what happens during the simulation in log files that the developers can use to debug their agents. Still, those logs are not allways sufficient to determine the reason for an error in the agents behaviour. For that reason some teams developed tools for evaluation of agents' performance, like the augmented viewers described in section 2.1.4, or the cooperation evaluation tool developed by the NITR team in 2006, and mentioned in section 5.3.

Most of the developed features are statistical, and sometimes only allow its consult after the simulation. But at times there is a need for real time data to be displayed in an empirical way, much like the Simulator viewer does. The problem, however, is that the viewer shows a wide view of the world, with all the objects in it. This is a problem because there are situations in which the developers need to know what the agent knows and how it sees the environment around it. In other words, sometimes its necessary to view the agents' World Model.

6.1.1 Agent Viewer

In order to fill the gap mentioned above, a new tool was created that will represent in real time the agent's World Model, the Agent Viewer. The World Model is the pillar for any decision that the agents make. And for a decision to be as correct as possible, the more information there is in the World Model the better. This means that there is much information to display, according to the World Model developed for the agent.

Currently the Agent Viewer displays the city map as the agent perceives it. Also, it colors the buildings which have been visited or seen, and the traveled, or seen, roads differently from the remaing unvisited areas of the map. It also marks the trapped civilians found by the agent.

Figure 6.1 displays the Agent Viewer during a simulation. The buildings in white represent visited buildings, and the roads painted yellow are traveled roads.

The Agent Viewer was developed using Simple Directmedia Layer (SDL) [1]. SDL "is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer" [1]. The drawing

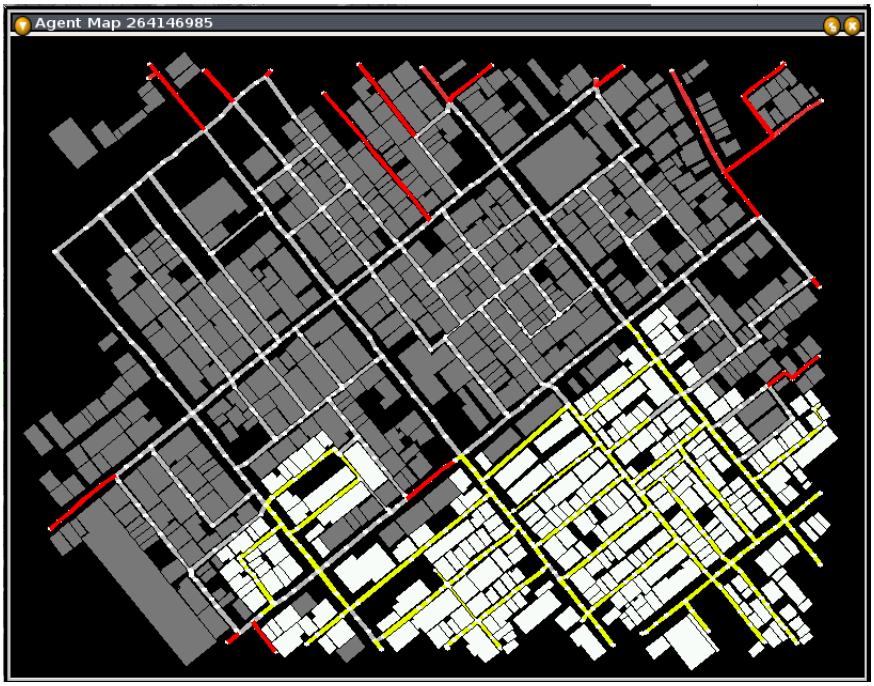

Figure 6.1: Image of the Agent Viewer

methods were developed in a very similar way to the methods of the RCR Simulator Viewer. This tool can be used by all types of agent.

The Agent Viewer was initially developed using functions from the X library. This was after considering the use of the functions in the Morimoto viewer (which was developed using Java) with the aid of wrapper classes in C++. SDL was chosen over the X toolkit library because it provided more options for the creation of shapes, and also because it was more simple to use. As for the use of the functions of the Morimoto viewer, it would limit the options of the agent viewer to the options of the Morimoto viewer.

The viewer proved to be a valuable tool for debugging. By adding functions that would better demonstrate what the agent knows it would also allow to see clearer the behaviour of the agent and detect possible errors.

Each agent may have a viewer associated to it. To identify the agent to which the Viewer's window belongs its window title has the agent Id as the figure 6.2 shows.

Figure 6.2: Agent Viewer window identification

6.1.1.1 Future Work

Presently the Viewer shows only the areas searched by the agent, a useful feature when developing map traveling and searching techniques. But more functionalities are planned. Some are statistical, others of a visual nature. For statistical data we have:

- Number of buildings searched/unsearched
- Number of civilians identified
- Number of civilians rescued
- Next civilian to rescue
- Next building to visit
- Team members (for when the agents form small teams)
- Traveled distance
- Remaining HP

Visually the viewer is planned to have the following features:

- Draw area around the point where a cry for help was heard
- Paint burning buildings differently
- Draw road blockades
- Draw the path taken by the agent when travelling
- Paint identified civilians to reflect their HP and/or position in the priorities list for rescuing
- Zoom in and out
- Select an area to analyse

6.2 Robocup Rescue Ambulance Team

The development of strategies for the Ambulance Team was the main focus of this work. This agent has the ability of performing actions in the RCR world. Its actions range from loading and unloading of an injured civilian, to rescue (or unburying) of a trapped civilian. In addition Ambulance Teams can also move, and communicate with other agents [38]. Those actions will be put to use in order to fulfill the objective set for the Ambulances, which consists of saving civilians lives.

The main concerns when developing an Ambulance Team are the search, rescue and communication methods. The work presented here has a greater focus on the search methods required to indentify the locations of the civilians. For any of the concerns mentioned the methods created must be applicable to any of the scenarios provided by the simulations (no communication and/or no Center). This means that the decision making may be centralized,

but there are situations in which the agents must have the capability to decide for themselves the best course of action.

For searching purposes the agent should be able to locate trapped or hurt civilians in the shortest time possible. For the rescue task the team of agents must decide which civilians to save, in what order, and how many Ambulances to use in each rescue. Communication is directly related to the previous tasks. It is a way of establishing cooperation between the agents in order to maximize their usefulness. The challenge with communication comes from the limitations that it currently has.

6.3 Robocup Rescue Ambulance Center

The Ambulance Center is a very useful agent. It cannot perform actions other than communicative actions, but it has bigger communication capabilities than platoon agents. It can communicate with other types of agents using the radio, and can receive more messages in a single cycle, than platoon agents [38]. An Ambulance Team can receive 4 messages in a cycle, and an Ambulance Center can receive $2 \times n$ messages, where n is the number of Ambulance Teams.

These communication capabilities make the center a great tool to colect in a single point data gathered by platoon agents in each cycle, and where it can be merged with previous data. After joining the new data with the old, it can filter the relevant information so it can be sent to the agents to make their individual World Model more complete.

By making the Center a central point for knowledge gathering, it also makes it appropriate for decision making because it has a wide view of the world and can make decisions based on a larger knowledge base.

By combining the three activities that a Center can perform, it is clear that the Ambulance Center is essential for a centralized strategy with the Center coordinating the Ambulance Teams.

6.4 Overall Strategy

After first learning how the simulator works, and how the agents function, a high level strategy was planned regarding how the Ambulances would operate in search and rescue tasks.

Due to the size of the city maps, and the number of agents available the first idea was to divide the map into smaller areas, or partitions. To each partition a team of agents would be allocated and that team would be responsible for the search and rescue tasks within that area. This means that it would be convenient to have a division as fair as possible. The teams must be equally divided and each team must be composed of at least three agents.

If a team were to successfully finish its task before other teams have done the same, then it would be free to assist the remaining teams. Whether the team members should split among the remaining teams, or if they should help the team more far behind is still up for discussion.

After a team is placed in its designated partition then it will begin searching for trapped civilians. From here there can be different approaches. Either all team members search for all civilians in their area, and only after do they begin rescue, or they rescue the civilians as they find them. Another solution is to take a bit from both strategies and allocate one of the Ambulances to be constantly searching while the remaining Ambulances save civilians as they find them.

This division brings several advantages. For starters, it reduces the domain of each agent, and the amount of information that it must process, and thus reducing the complexity of the environment. Since the agents only care about their partition then they do not need to know about the remaining areas, and because of that, communication is greatly reduced. A drawback is that the scheduling of rescue tasks is done on a local basis, which means that on a given partition, the Ambulances could be busy rescuing a civilian, when there are more urgent cases in other partitions. To avoid such situations, a cooperation mechanism can be used for such an occasion, in which a team that finds a civilian with a short time to live can request the assistance of other teams. After the rescue as been completed, the outside agents can return to their respective partitions.

Another advantage of this strategy is that it is facilitated by the existence of a Center, but it does not depend on it. The division of the map can be made by the agents, as well as the rescue schedulling, and with few agents on each team there is no problem in exchanging messages between them. Also, this approach can be employed with agents of other types, with the greatest difference being in the scheduling of each type's respective task.

6.5 Single Agent Scenario

In an environment with multiple heterogenous agents, it may seem odd to discuss Single Agent Scenarios. But the fact is that regardless of the different types of agents, and the number of agents that can be present in a simulation, there are cases when the agents do not have the aid of Centers, or even radio communication.

In the first case, there is still the possibility that the agents can coordinate themselves using radio communication, although it would be much harder because of the communication limitations. There is even the possibility of setting one agent to act as a coordinator, but that would also be very complicated due to the communication constraints. In these cases the agents are responsible for all the processing for decision making, search and rescue planning and coordination. Also, the way they use their communication resources must be well planed, since they are needed to exchange sensory data to add to the World Model, and to coordinate search and rescue tasks.

In simulations with no communication, even if there is a center, it cannot help the platoon agents in any way. In such a simulation, the agents are alone to make their own decisions, which will have less information to be based on because it is not possible to share sensory data between the platoon agents.

The platoon agents are still left with the advantage of knowing how many agents are there, of what types, and what is their initial position. From this information they can estimate and predict some data. For example, an agent will not visit an area where it knows that another agent is in the vicinity. Such strategies imply a great deal of assumptions and it is the way the agents are structured, and the strategies that they use, that will ultimately reveal the quality of their predictions.

The last case is a very unlikely one, but it is, nevertheless, possible. There may be some simulations in which a type of agents can only have one agent. The 2008 rules state that the minimum number of agents of any type, with exception to Civilians, is 0. Table 6.5 lists the minimum and maximum number of agents that can be in a simulation.

Table 6.1: Number of agents available in a simulation [42]

Type of Agent	Minimum	Maximum
Civilian	50	90
Ambulance Team	0	8
Firefighter	0	15
Policemen	0	15
Firefighter Center	0	1
Ambulance Center	0	1
Police Center	0	1

6.5.1 World Model

As was discussed in section 5.2 the World Model (WM) is of great importance because it is the basis for all decision making processes. In the WM the agents will store the information regarding the world that surrounds them, and use it for nearly every action that they perform, since path planning, to rescue scheduling, and other tasks. For a WM to be of use it is not enough to have a large set of data. The organization is also crucial, because information is only useful when it is readable. This means that the data should be well divided, easy to access, and easy to read.

In order to be able to organize the WM we must first define what to save there. What is stored in the WM depends on the strategies defined. For example, an Ambulance may know that a building is burning, but it does not need to know anything more specific than that. The idea is to keep only data that will be used, otherwise too much information will have nefarious effects. For the strategies implemented with this work an Ambulance Team needs to know the data listed in table 6.2.

Table 6.2: World Model data and purpose

Data	Purpose	Properties
City Map	Know what areas to search, to plan paths to certain destinations, to know locations of centers, fires, civilians, etc.	Map saved as a representation of a Graph
Current Partition	Same as the City Map, only on a smaller scale.	Partition of the City Map allocated to the agent represented as a Graph
Buildings	Know which have been visited, and which haven't	Two sets, one for visited, another for unvisited; mark burning buildings
Other Ambulance Teams	Prediction of other ambulances' behavior in search and rescue tasks, and also to communicate	Set of teammate's references, and their status; radio channel to communicate
Ambulance Centers	Communication	Radio channel to communicate with Center
Team Formation	Knowing what roles to take according to the Formation of the team	Description of the Formation; Roles taken by the agent and its teammates
Refuges	Plan a path to the nearest refuge	Set of references of the refuges
Found Civilians	Plan rescue schedule	Set of civilian's references, their status, and location
Visited Areas	Know what areas have been visited	Set of nodes already visited, seen, or that do not need to be visited
Rescue Schedule	List of civilians to rescue, and in the correct order	Set of civilians references
Personal Data	Self evaluation	Agent's HP, position, state, etc

6.5.1.1 City Map

In chapter 3 it was mentioned that the city map of a RCR simulation can be represented as a graph. Many teams of RCR developers have used this kind of representation with their agents. The work presented here was no exception. The graph built for the agent's WM can be represented by its common form $G = (E, V)$, where the objects of the type $NODE$ are included in the V set, and the objects of type $ROAD$ are included in the E set. The roads allow traveling in both directions, which means the graph is undirected.

To represent the components of the graph there are two structures, GraphNode and GraphEdge. These structures are a combination of C++ data types, and data types provided by the RCR simulator. Besides the common fields for data structures representing nodes and edges in a graph, the structures implemented have aditional fields for uses in various situations.

There is a field to identify the node, or edge. That field is the Id that every object in the RCR world uses to identify itself and is, therefore, unique. For that reason it is used in the GraphNode and GraphEdge structures to identify both objects in a way that can immediately be used in RCR simulator tools.

Nodes and edges each have their own metrics. Nodes save the number of buildings to which they are connected, and edges have their length. The number of buildings is important for graph partitioning, and the length of roads is necessary for calculation of the distance traveled in a cycle.

Nodes also keep their geographical coordinates. This information is used for graph partitioning and for path planning.

Additionaly edges have a field to indicate its block level. At the start of the simulation this field is set to 0 in all edges, but as the simulations continues and roadblocks are discovered, the edges where the roadblocks are located are updated to reflect the existing obstacles.

Alleys Like many real cities, the maps used in simulations have alleys. An alley is a dead end. To the graph that represents the map it is a set of nodes and edges forming a road that contains no cycles, and where the connection to the remaining nodes not belonging to the alley is made by only one node. Alleys are created, or detected, after the graph is initially created.

The process of modeling the city map is completed after the detection of alleys in the map. The set of nodes and edges that compose an alley are discovered after first identifying the nodes in the map that only have one neighbor and following the sequence of nodes until hitting a node with several neighbours. The reason for separating these neighbours from the rest is that when an agent passes by a node with connection to an alley the agent will visit the alley first. This will save time when travelling through the map, since the set of nodes and egdes that compose the alley are only accessible through one node, and if the agent does not visit the alley in that moment, it will have to return to that node in order do visit the alley.

There may be alleys within alleys. Outside of the outer alley the agent perceives it as only one. However when travelling through this alley, it will behave the same way as before, and visit all suballeys that it encounters.

The final result of map analysis, including graph creation and alley detection, can be seen in the figure 6.3. All roads painted in tones of grey are alleys. Those with lighter color represent a new level of alleys.

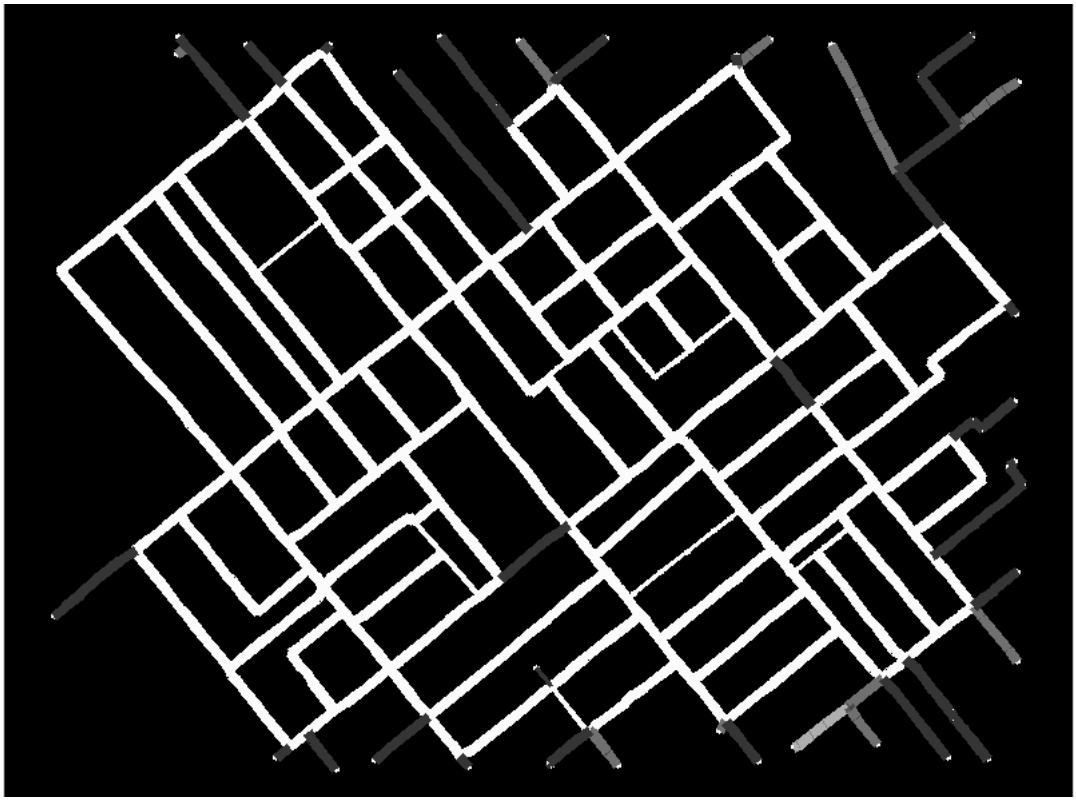

Figure 6.3: Image of a RCR map with its alleys in shades of grey

6.5.2 Search

As was detailed in section 2.1.5, each type of agent has its specific function. However, should any of those agents be free from assignment to their respective functions, they are free to search the map for trapped or hurt civilians, burning buildings, or blocked roads, and then report them to the proper teams of agents. This being the case, search is the only function common to all types of agents.

Search is the first step for RCR agents to achieve their goal. The way the agent performs the search will have a great impact in the final score. Before detailing the implemented strategies, first it must be defined what are the targets of each type of agent when searching. Common to all types of agents is that they do not know the location of their targets and must search for them. If a civillian is in need of rescue, then he must be inside a building and that is where the ambulances must search for civillians. Only buildings burn, so Firefighters will only search for fires in buildings. Hence, Ambulances and Firefighters have in common the the places where they will look for their targets, which are buildings, whose location is known to the agents. Policemen have the mission of clearing the roads of obstacles. However their effort is better applied when aiding ambulances or firefighters reaching their desired destinations, that are buildings. Joining the information of the unknown location of the targets, together with the places where they are located (buildings) and the fact that the location of the buildings in the map is known, it is clear that after inspecting every building in the map all targets of each type of agent (Ambulance and Firefighter) will be found. For

this reason a search is considered complete when all buildings have been inspected.

After a first search was completed, a second one is required in order to obtain a more comprehensive knowledge on the agents' targets. Returning to a building where a civillian is trapped will allow an Ambulance to determine the rate at which the civillian is losing HP. Firefighters need to return to burning buildings to determine the rate at which the fire is spreading. Having this kind of information will allow the agents to better determine the best course of action in order to save as many civillians as possible, and to reduce the damage to property to a minimum. This search, and others that may follow it, will not be performed the same way as the first search. Since the targets locations have been detected in the first search, the agents can plan a direct path to the intended targets.

6.5.2.1 Evolution of Map Traversing

In order to reach a efficient method for map traversing, many solutions, and improvements were tested. The reason for this was not only the search for a good method, but also to understand better how the RCR world worked in a learn-by-doing approach. With the information obtained from each step, new ideas appeared which led to successive improvements.

Traveling to Unvisited Nodes At the very beginning, and has an exploratory method for map traveling and lerning how the agents function, the agents had the objective of traveling to all nodes. The agent would plan a path until a point where there were no unvisited nodes. The path planning method used was based on the Civilian code which planned a path to a Refuge using the Breadth-First Search algorithm. All nodes traveled through with that path would be marked as visited. After reaching the end of the path it would then backtrack to a node with unvisited buildings. This backtracking would be calculated using a list of ancestors that would keep the previous node of each visited node. The problem with this method is that it did not allow to obtain the information required and described in section 6.5.2, since it wouldn't stop to search any buildings. It did, however, serve its purpose as it allowed to understand how the agent gathers information necessary to build a path, and how that path is built. This method was mostly based in the method used by civilians to travel to a Refuge, which is a breadth-first search for a Refuge.

Renewed Search For Unvisited Nodes Continuing with the goal of visiting all the nodes in the map a new method for calculating a path to an unvisited node was developed. Instead of backtracking through the list of ancestors, as was done in the previous method, the agent, after reaching a node with no unvisited neighbours, planned a path to a node with unvisited neighbours using the Dijkstra search algorithm [15]. The algorithm expanded its search tree until hitting a node that wasn't in the visited list.

This improvement allowed the agent to travel shorter distances to reach a node with unvisited neighbours.

Figure 6.4 shows an example where an Ambulance is surrounded by visited buildings and travelled roads. It will determine the shortest path to an unvisited node, which is represented by the arrow.

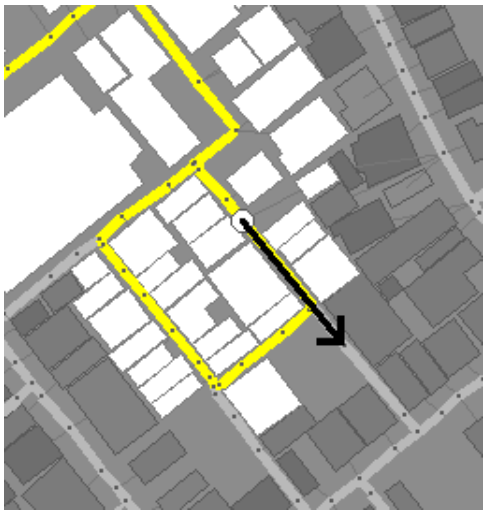

Figure 6.4: Example of traveling to a node with unvisited neighbors

Backtracking And Path Planning In the previous methods the path calculated for exit a completely visited area was traveled in a cycle, and only after reaching the destination the agent calculated a path through unvisited nodes. With this new method, when the agent needs to leave a visited area it will calculate the necessary path, but it will not imediatly travel that path, but will, instead, calculate the path through unvisited nodes starting at the end node of the path calculated in the previous step. After both paths have been calculated they are joined together forming a single path that will be traveled in that cycle.

Creating paths with this method saves a cycle that would be wasted each time the agent needed to exit a visited area.

Searching For Buildings Until now all of the previous methods searched for unvisited nodes. But, as was established at the beggining of this section, the objective of the search should be buildings, and not nodes. For this reason an agent, when planning his path, will have as destination a node with connections to unvisited buildings. The agent will not proceed to another node until the current node has all of the buildings connected to it visited. In order to obtain the information regarding a building the agent must stop at the building for it to receive the building's data, including whether or not a civilian is trapped inside the building, which is provided by the RCR kernel as a list of 'sensed' objects.

At this point the agent begins to search in a manner that goes toward its objectives.

From Visiting Buildings To Looking At Them Further tests showed that an agent doesn't need to enter a building for it to receive the building's data. If the building is in the agent's viewing range (10 m)it is also included in the sense list that the kernel sends to the agent. Figure 6.5 displays an agent's viewing range and a set of buildings encompassed by it.

By looking at buildings, instead of entering them an agent can save several cycles. For example, if a node has access to four buildings, it would take four cycles to visit them all. However, if they are in the agent's viewing range from the node, they can all be checked in a single cycle.

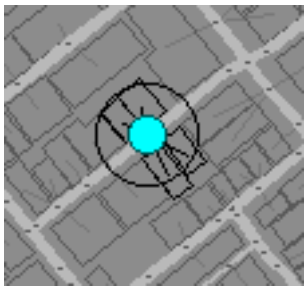

Figure 6.5: Image of an agent with buildings in his viewing range

With this method, an agent travels to a node with connection to unvisited buildings at stops at that node to allow to receive information regarding nodes in the viewing range. If any buildings are left to check, then the agent will enter them personally.

Neighbour Lookup With the intent to further improve the search time, a new method was implemented. This method is characterized by the ability to look up the list of Reachable Buildings (RB) of a node and its neighbours. The RB list is composed of buildings that are in the viewing range of an unvisited node and also the buildings accessible from that node. If the RB is empty then there is no point in stoping at that node. Even if there are buildings in the RB set then that does not mean that the agent will stop in that node. In order to optimize map visiting the agent considers unvisited neighbours of that node and analyses the RB from the neighbours. If a neighbour's RB_n set contains the RB set of the node being analysed, then the agent does not need to stop at that node and can continue to that neighbour. This method is particularly useful in the case RB_n contains but is not contained in RB.

The following pseudo-code helps to understand how an agent decides whether or not to stop at a node.

```
path := empty_path
node := agent current location
while ( !stop ) {
      path.add(node)
      stop := true
      nodeViewBlds := getUnvisitedBldsInViewRange(node)
      nodeAccessBlds := getUnvisitedBldsAccessibleFromNode(node)
      nodeReachBlds := nodeViewBlds + nodeAccessBlds
      for ( each unvisited neighbour of current node ) {
            neibViewBlds := getUnvisitedBldsInViewRange(neighbour)
            neibAccessBlds := getUnvisitedBldsAccessibleFromNode(neighbour)
            neibReachBlds := neibViewBlds + neibAccessBlds

            if ( neibReachBlds is NOT empty )
                  if ( neibReachBlds CONTAINS nodeReachBlds ) {
                        node := neib
                        stop := false
                        - exit for-cycle
                  }
```

```
        }
}
```

Discarding Nodes Until the previous method, if the agent found an unvisited node, it would follow it until reaching a node with no unvisited neighbours. This was a disadvantage as the agent would follow that path even if there were no buildings to check at any point of the path. To prevent this from happening the agent still plans the path through unvisited neighbours. However, if it reaches a node with no unvisited neighbours, and it hasn't passed through a node with a non-empty RB set, then that path, will be discarded, and the nodes that compose that path will be marked as visited.

After discarding a path, the agent would then calculate a new one. This led to a reduced search time, as the agent would not lose time in traveling to nodes with nothing to gain from them.

Direct Entrance To Buildings When checking the RB set of a node to consider whether to stop at that node or to continue to the next neighbour, an agent does not distinguish a viewable building from a building connected to the node. So, if a node had a non-empty RB set, and the agent deemed necessary to end the path at that node, the agent would stop there even if there are no buildings in the node's viewing range. This meant lost cycles, as the agent would stop at the node and learn nothing of the buildings. Not only that, but to investigate every building connected to the node it would take two cycles for each building. One to return to the node, and another to visit to the next building.

To prevent this great waste of time the agent is now able to distinguish when it should stop at the node, or just go straight to a building.

6.5.2.2 Results

The evolution registered here led to improvements in the total search time of the map. Some brought more visible results, others were more discreet, but all helped in achieving a more efficient method of map searching. Not all attempts were successful, as can be expected. One failed attempt was a different backtracking method, that planned a path to the nearest node with unvisited buildings, instead of searching for the nearest unvisited node. This method had poor results, and was discarded.

One of the improvements that was more noticeable in the search method, was the change from backtracking through the ancestors list, to planning a path to nodes with unvisited neighbours. Table 6.3 shows results taken from 10 simulations in each map. When this method was developed the first method for map partitioning was also developed, and that is why the results in the table were taken with the map divided in four partitions, with one agent in each partition. In this table the difference between the two methods can be evaluated.

Table 6.3: Results comparing average time required to visit each partition completely with each method

		Partition 1	Partition 2	Partition 3	Partition 4
Kobe	**Building Number**	*123/128*	*206*	*148*	*255/260*
	Backtrack	108.5	160.5	149.3	162.3
	Travel to node with unvisited neighbor	96.3	140.9	132.4	142.8
VC	**Building Number**	*401*	*405*	*198*	*262*
	Backtrack	257	243.3	221.7	206.5
	Travel to node with unvisited neighbor	232.3	226.3	196.2	176.7

Table 6.4 displays the improvements on the average number of visited buildings brought by looking up RB of the node's neighbours. The results were obtained by performing ten simulations using one agent in three different maps without roadblocks, and they show that there is a significant difference between search with lookup and without.

Table 6.4: Results comparing average number of visited buildings during a simulation with neighbour building lookup and without

Map	Kobe	VC	Foligno
Buildings In Map	737	1263	1080
With Neighbour Lookup	497.8	482	372.2
Without Neighbour Lookup	416.8	394.9	311.6

The last modification described (Direct Entrance To Buildings) also provided reduced search times. In one of the maps tested there is a partition that saw a reduction of almost 80 cycles in its search time. Table 6.5 shows the detailed results obtained from ten simulations in three maps divided into four partitions. These results presented in the table show that there is a clear improvement brought by this modification. As with the previous simulations, the maps used were stripped of roadblocks.

Table 6.5: Results comparing average time required to visit each partition completely with direct entrance to buildings and without

		Partition 1	Partition 2	Partition 3	Partition 4
Kobe	**Building Number**	*204*	*183*	*170*	*183*
	Stopping before entering	110.2	139	121.2	123.2
	Direct Entrance	97.8	106.6	101	103.6
VC	**Building Number**	*311*	*363*	*245*	*347*
	Stopping before entering	197.2	197.2	230	186.8
	Direct Entrance	164.3	163	172.4	162.9
Foligno	**Building Number**	*306*	*222*	*249*	*304*
	Stopping before entering	244.2	260.8	241.2	235
	Direct Entrance	199.8	183.6	179.5	194.2

The search method has a flaw that results on longer search times. When planning a path the agent simply uses the neighbors of the nodes that compose the path (unless one of them has a connection to an alley). Otherwise the neighbors are not selected under any criteria. The problem that comes from this is that the agent sometimes chooses to travel to a given node, when there are other unvisited nodes nearby that would be preferable as a destination. The impact of this is more noticeable when the agent continues its search and gets farther away from that destination. A good example of this is shown on figure 6.6. Notice the buildings highlighted and how all the surrounding area has been inspected. The agent will continue its search and will eventually return to that point, which results in a poor search time.

Figure 6.6: Area with unvisited nodes

To summarize, the search method implemented does not calculate a global solution such as a path that visits all the buildings, but instead determines a path to the closest point where the agent can see an unvisited building. This method may not be the most efficient, but it is robust and preferable to the planning of a global path for several reasons. The first one comes from the dynamic nature of the environment, and specifically the location of blockades which is unknown at the start of the simulation. Considering the implications of calculating a global path, which include the complexity of such an algorithm, like the Travelling Salesman Problem [22], added to the size of some of the maps which have many nodes, cycles, and alleys, computing a global path might take some time. If an agent were to follow that path and run into a roadblock, then it would either have to recalculate the global path, or stay there waiting for a policeman to come clear the roadblock. Either way would represent a great loss of valuable time.

6.5.3 Rescue

When engineering a rescue strategy there are several factors that should be taken into consideration, such as the number of agents to use for a rescue (if it is only one, or all the available Ambulances, or if the number of Ambulances depends on certain parameters), and if the civilians should be rescued upon discovery, or if the agents should collect more data before determining what is the best course of action. Remembering that agents might be deprived of communication to coordinate their rescue efforts, the strategies developed should be able to act in such situations.

When an agent is on a no communication scenario, it can either assume that the remaining agents will act in a certain way, or that it must proceed to the rescue all by itself. When counting on other agents there might be some complications. The other agent could find themselves unable to reach the rescue location due to traffic jams or road blocks, and with no way to communicate their status to the other agents.

Currently the agents use a greedy rescue strategy, with no consideration for the actions of their fellow teammates. They save civilians as they find them and then carry them to a refuge. To prevent unnecessary rescues an Ambulance will only save a Civilian if it is buried, or suffering damage. If this is not the case, then the Civilian can survive the rest

of the simulation without aid, and the Ambulances avoid wasting time with its rescue. This method, although similar to real life rescue strategies, is not very efficient because it might waste time on a civilian that doesn't need to be rescued either because its HP won't reach 0 until the end of the simulation or because the Ambulance won't be able to successfully rescue the Civilian before its HP reaches 0. A situation such as that is worsened by the fact that while an Ambulance is busy performing an unnecessary rescue, another civilian in dire need of help could be dying. This happens because the Ambulances do not try to predict the remaining lifetime of the civilians that they encounter.

In the event that more than one Ambulance is performing a rescue on the same civilian the Ambulance with the lowest ID will be in charge of carrying the civilian to the refuge, while the remaining Ambulances continue searching the map. This method is simple to implement and it doesn't require communication, because the agents can see if there are other Ambulances in their viewing range, and what is their position.

With this method the Ambulances have a total of three states: Rescuing, Searching and Carrying To Refuge (implies the loading and unloading of the civilian). The transition between these states can be seen in figure 6.7.

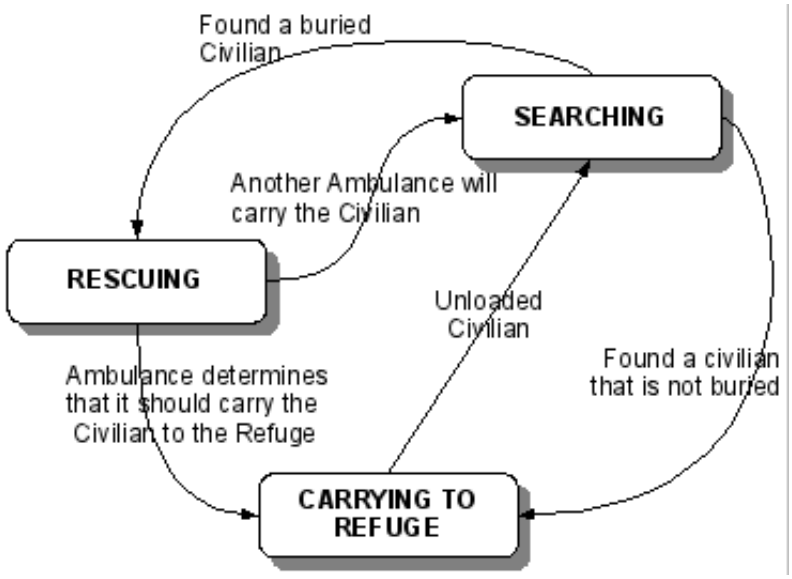

Figure 6.7: States of an Ambulance agent

6.5.3.1 Results

In table 6.6 there are results that show the performance of the Ambulances while trying to save as many Civilians as possible. The results are an average of the surviving Civilians, obtained from running ten simulations on each of the two maps (Kobe and Foligno). These simulations did not include roadblocks and burning buildings. Please note that in the Kobe map, four Civilians die at the beggining of the simulation.

The number of surviving civilians include those rescued, and others that were not rescued, but still survived until the end of the simulation. The latter also include the civilians that the Ambulances decided not to rescue because they were not hurt. The results for the Foligno map were not as negative as it would be expected due to the size of the map, and the number

Table 6.6: Results evaluating the performance of Ambulances rescuing civilians

Map	Kobe	Foligno
Number of Civilians	72	60
Number of Ambulances	6	8
Number of Refuges	3	4
Number of Buildings	737	1080
Surviving Civilians	50.8	53.2

of buildings that the Ambulances need to search to find the Civilians. The reasons for these results are directly connected to the increased number of Ambulances, when compared to the Kobe map, and to the damage suffered by the Civilians at the beginning of the simulation, which was a lot more in the Kobe map. In the Foligno map, there were more Civilians suffering no damage at all, which meant reduced work for the Ambulances. Also, the damage that the other Civilians suffered was not as serious as the one inflicted to the Civilians in the Kobe map. The last factor that gives an advantage to the Foligno map is the increased number of Refuges. In a map as big as Foligno's it is important to have several Refuges well dispersed throught the city, unlike the Kobe map, where the three Refuges are close to each other. Figures 6.8 and 6.9 show the location of the Refuges in both maps.

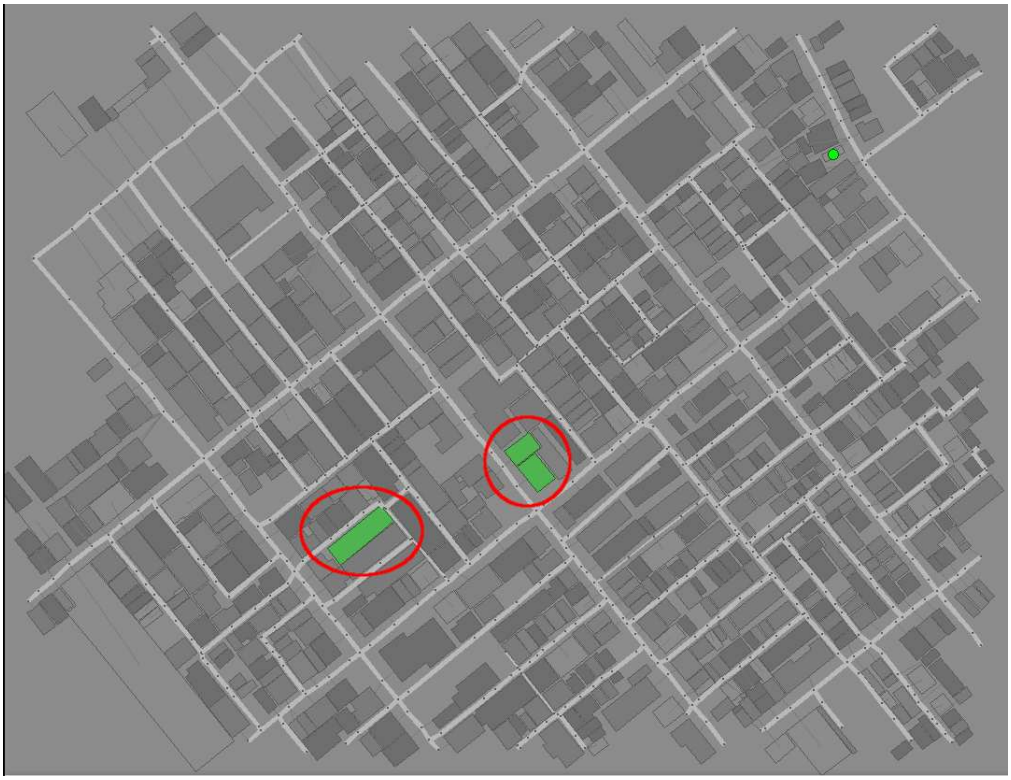

Figure 6.8: Location of Refuges in the Kobe map

Figure 6.9: Location of the Refuges in the Foligno map

6.6 Multi Agent Scenario

The most common simulations allow communication among the different types of agents, and also include Centers. A simulation with communication already makes a great difference in the strategies the agents employ to achieve their goals. If there are centers too, then there is an even greater difference.

In order to take advantage of the conditions given in such a simulation, methods for communication and cooperation must be developed. Those methods must be efficient, must use as many resources as possible, and take into consideration the existing limitations, which come mostly from the constraints in communication.

Communication and cooperation are not the only changes brought by Multi Agent Scenarios. The process of map searching and civilian rescuing may suffer changes, due to the new resources available.

6.6.1 Subgraphs

We'll begin by the changes to the search method of RCR agents in Multi Agent Scenarios. To use the number of agents properly it was chosen to divide the map into partitions, much like other teams have previously done [6, 24, 33, 11, 20]. The division of the map can be used

for more than searching. By assigning a team of agents to a partition, they can be responsible for the search and rescue of the civilians in that partition, just like the strategy described in 6.4. This reduces the domain where agents act, and, consequently, reduces the amount of information that they must process. Partitioning a map also brings several advantages to the search task. By dividing the map into similar partitions and assigning each partition to a team of agents, or to a single agent, it is ensured that no area is checked needlessly. By creating partitions according to the number of available agents it is also guaranteed that all areas of the map are covered. This division of the search area will also lead to a reduced search time because of the cooperation existing between the agents. Since the simulation map is represented by a graph, Graph Partition concepts are useful for designing a method to divide the map into several partitions which will aid in turning the search more efficient.

The work presented here implemented Graph Partitioning algorithms, but not any known algorithm. Nevertheless some research on the subject was made, which provided pointers on how to create a new partition method. Several studies [27, 13, 9, 25] in this field were reviewed to obtain the methods implemented.

The following subsections present three different methods of map partition as well as the steps required to perform the partitioning. Please note that during this section, references to total number of agents, or teams of agents refer to a single type of agent only.

6.6.1.1 Common Steps

For all the methods developed there are some common steps that take place before and after the actual partitioning. Those are: determining the number of partitions required; obtaining map limits for the partitions (before); assigning agents to partitions; and detect new alleys (after).

The number of partitions will depend on the number of available agents. The number of partitions may be the number of agents, or the number of teams of agents. In some cases it may be preferable to divide the map into several partitions, even if there aren't enough agents to assign those partitions. This is true due to the search method that does not provide a solution that encompasses the entire map. As a result, searching a large area will eventually lead to a situation where the agent calculates a path to a node, when it would be preferable to visit other nodes. In a smaller area, the effects of this would be significantly reduced.

Calculating map limits has a different significance for the three methods, but the operation is the same. Each object in the RCR world ha a set of x-y coordinates. What is done in this step is search for the minimum and maximum values of x and y.

After the map division a recalculation of the alleys will be necessary because the cuts made during graph partitioning will result in new alleys that didn't exist when creating the original graph.

6.6.1.2 Equally Sized Partitions

This method was initially created on the assumption that partition area would be a fair metric to use when dividing the map. Therefore this method will return equally sized partitions (ESP) so that agents will have the same area to cover.

The first partitions created with this method were very different from the ones created by the final version. At the beggining the method was a lot simpler and but also had some flaws. The map was divided by an horizontal line cutting the map in two equal parts, and then

the partitions were divided as equally as possible between the two parts. This led to a poor division of the map because there were partitions with different areas like in the left image of figure 6.10. After obtaining the limits for each division, the partition would begin by selecting a node inside the limits of the partition being created, and expanding to its neighbours until there was no more option for expansion. Neighbours outside of the rectangular area where ignored. Essentialy this initial method for partitioning was more useful as an exploratory method, much like the first steps in the development of the searching method.

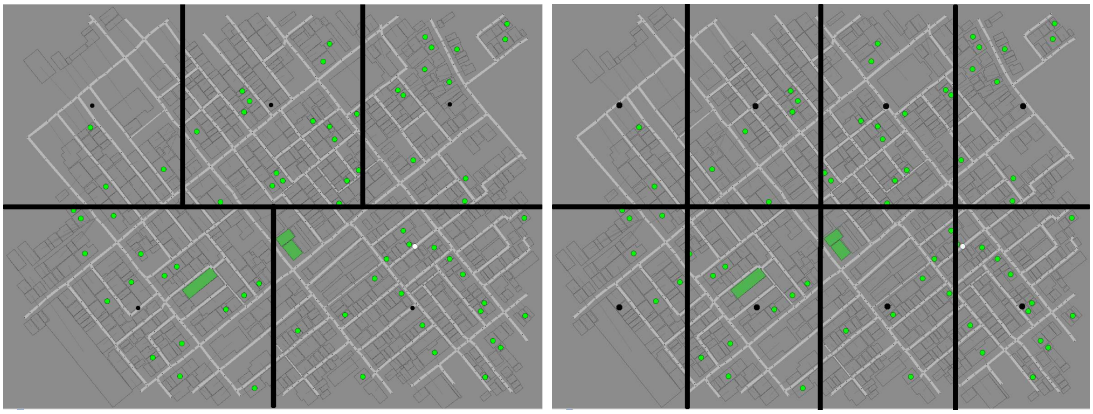

Figure 6.10: Initial cuts for original ESP method

The final method shares some steps of the initial method like the calculation of the total area of the map, and the area required for each partition, which is a simple division between the total area and the number of partitions calculated previously. Since a map can assume any shape, calculating the exact area would prove somewhat of a challenge. Not only that, but to calculate the area for each partition would prove even more difficult. To avoid all the steps that would calculate a more precise area, the shape assumed to calculate the total area is a rectangle with the measurements obtained from the map limits. Figure 6.11 displays an example of the area calculated for the Foligno map. This map is also a good example of what makes calculating the real area so difficult.

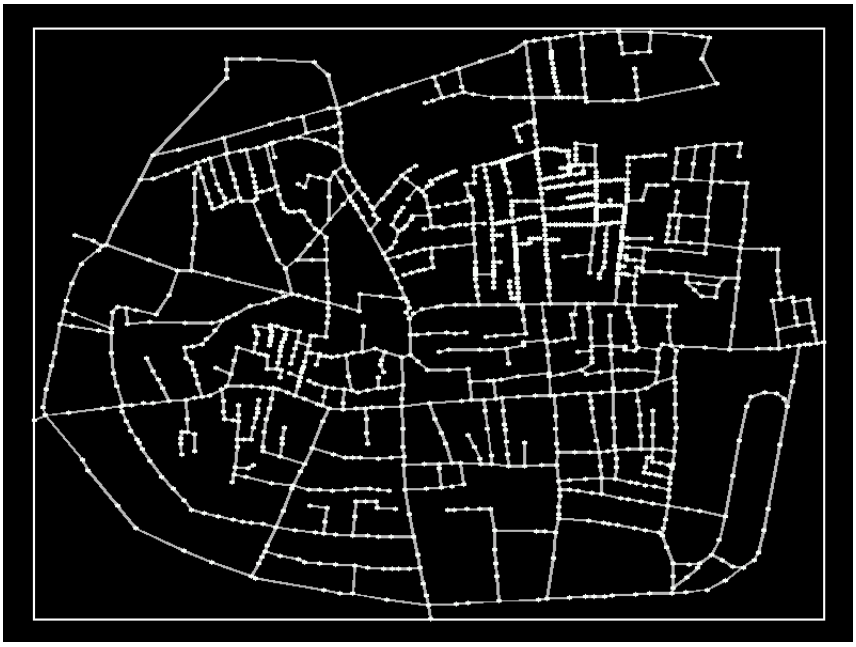

Figure 6.11: Total area calculated for Foligno map

The following step is calculating the two principal directions that exist in the map. The cuts made with those directions will grant a cleaner division of the map. These are calculated using as a basis the Hough Transform [52]. Whereas in the Hough Transform it calculates two parameters, r and θ the partitioning method requires only the angle θ. The differences do not stop here. Hough Transform calculates its parameters by plotting a line from the referencial origin to the data point, as the figure 6.12 shows.

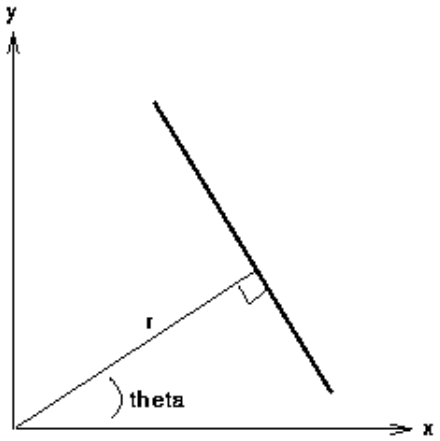

Figure 6.12: Traditional calculation of Hough parameters

However, since there is no need for the parameter r, and in order to simplify the calculations, the method used to calculate the main angles present in a map merely obtains the angle between two nodes, taking into consideration the nodes position. Figure 6.13 shows two examples of how this is done.

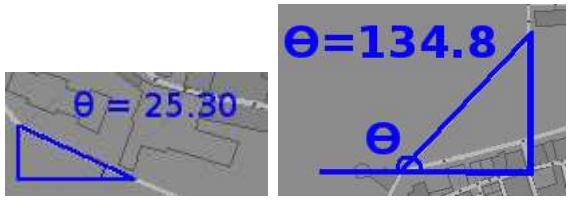

Figure 6.13: Example of calculating main directions

After the main directions are calculated, the next step is to determine where the cuts will be placed in the map to obtain the ESP. Figure 6.14 shows the Kobe map with the cuts set for nine partitions.

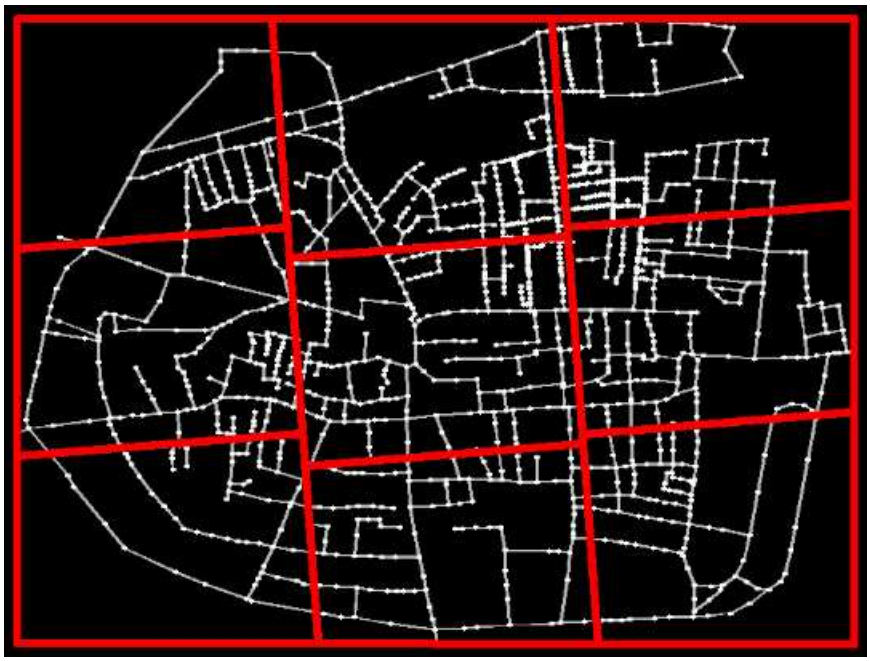

Figure 6.14: Example of a map divided into 9 partitions

After placing all the necessary cuts, the creation of the partitions begins by assigning all nodes to their respective partitions according to the x-y coordinates of the node and the areas defined by the cuts. Each partition is created separately. After a node has been allocated to a given partition it is checked for links to alleys. If it has any, then the alleys will be added regardless of their coordinates being in the area defined by the partition to which they are being assigned. This is done because, as explained in section 6.5.1.1, an alley is only accessible from one node.

After all the nodes have been assigned to their respective partitions, the next step is to determine which nodes have access to buildings and assigning the buildings to the partition the node's partition. There are buildings with access provided by more than onde building. To prevent from one building being assigned to several partitions it is ensured that a building when assigned once cannot be assigned a second time.

This method is the fastest of the ones presented here, and it is conceptually very simple.

The implementation, however, was quite extensive, mostly due to the step where it decides to place the cuts in the map. Nevertheless the biggest flaw of this method is that it might produce partitions with few or no nodes at all. For example, in figure 6.15 only the initial cuts have been determined and already there is an area with no nodes. This flaw results from two causes: no attention is given to the number of nodes in the partition; the limits defined to create the partitions do not respect the map's real shape. The effects of this flaw are minimized in maps with shapes that are more close to a square, or rectangle (like the VC ou RandomLarge maps), or if the map is divided into fewer partitions.

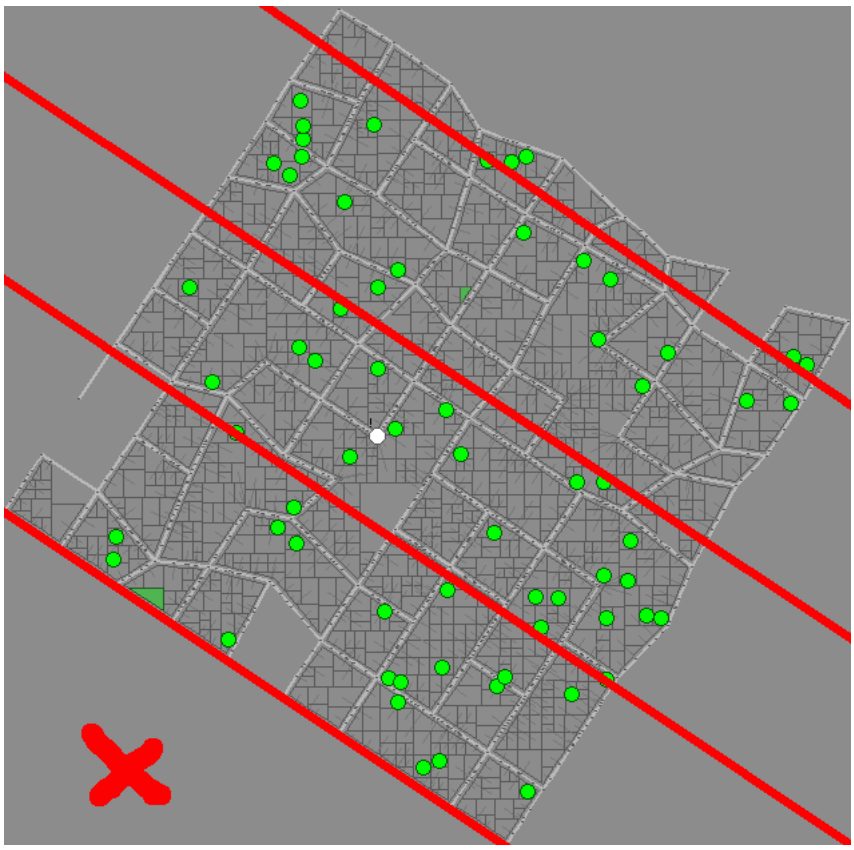

Figure 6.15: Example of a division with no nodes

6.6.1.3 Building Oriented Partitions

As explained in section 6.5.2 the main goal of searching are buildings. This means that no matter how big an partition is, if it has only two buildings, it will be completely searched very quickly. In contrast, a small partition may have a great number of buildings, which would lead to a longer search time. A situation which may occur in the previous method. This raises the problem of graph balancing that is mentioned in several graph partitioning algorithms. Altough in most algorithms the balancing factor is the number of nodes in a partition, for this problem the number of buildings in a partition was chosen as the metric for balancing in this method.

This method creates partitions by expanding in a spiral, clockwise fashion from a node of

origin. The node of origin is determined by calculating a number of nodes in the map, equal to the number of required partitions, that are as equally spaced from each other and the real map limits as possible. After the nodes of origin are determined, the partition creation will commence. Its expansion will behave in a way similar to that described in the figure 6.16.

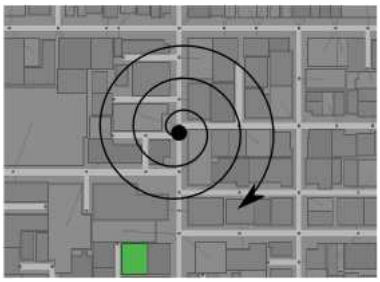

Figure 6.16: Example of partition expansion

Unlike the previous method, all Building Oriented Partitions (BOP) are created simultaneously. The following pseudo-code displays how a partition is created.

```
maxBldsPart := 0
while ( remain unassigned nodes ) {
    for ( each open partition ) {
         - choose next node
             if ( no node returned )
                 - close partition
             else {
                 - add node to partition
                 - add buildings to partition
                 - update partition limits

                 if ( partitionBlds < maxBldsPart )
                     - continue for-cycle on current partition
                 else
             if ( partitionBlds > maxBldsPart )
                         maxBldsPart := partitionBlds
             }
    }
}
```

When choosing the next node it will be from one of three outcomes. If it's the first iteration the node of origin will be returned. Otherwise it will check for 'lost' nodes (a node that is inside the limits defined by the partition and is neighbour to a node belonging to the partition, but hasn't been added to that partition yet) first, and if there are none, it will choose a node considering the orientation of the expansion. The last two options both use the partition limits, and that's why it must be updated every time a node is added. The partition limits are calculated the same way as explained in section 6.6.1.1. To avoid an accumulation of these nodes, which would lead to unbalanced partitions, the algorithm looks for an occurrence and adds it to the partition's node list. If the selection of the node is made according to the

orientation of the expansion it will look for nodes that fall outside the partition limits, and are neighbours to a node belonging to the partition.

As in the previous method, a special attention is required when dealing with alleys. When a node is selected, it must be checked for alleys. Should it have any, then they must be added to the partition, and any buildings resulting from it must also be taken into account.

In the event of a partition having nowhere to expand to, that partition will be closed regardless of its number of buildings. A closed partition is not considered in the for-cycle of the pseudo-code shown above.

This method provided more balanced partitions, and the implementation was not very complex. On the downside this method does not escalate well. Large maps can take some time to be partitioned. Another problem of this method is that it might produce partitions with malformed shapes, like the one in figure 6.17.

Figure 6.17: Malformed partition

Although this partition may have an approximate number of buildings just like the remaining partitions, the size of the partition as a negative effect because the agent will have to travel long distances to reach the buildings in the partition.

The result of the partitioning algorithm can be seen in figure 6.18.

Figure 6.18: Foligno map divided in 4 partitions

Although the partitions may vary in size and in shape, they hold approximate numbers of buildings. For example, in the partions presented in figure 6.18 the top two partitions, from left to right, have 273 and 271 buildings, and the bottom two partitions, from left to right, have 274 and 274 buildings.

6.6.1.4 Observation Points Partitions

The third an last method of partitioning derived from conclusions taken with the previous method, and from the method implemented by the Impossibles team in 2006 and 2007 [17, 18]. Despite the improved results that the BOP partitions brought, another limitation was discovered. A close analysis of table 6.7 gives pointers to the limitation discovered. The table shows the results gathered from running 10 simulations in each of the two maps, divided into four partitions, with an agent in each partition. The partitions were created with the two previous methods. The tests were also made using the RandomLarge map, but the results obtained from those simulations where inconclusive.

Table 6.7: Results comparing average buildings visited with ESP and BOP

Number of cycles to complete partition search								
		Time needed for each partition				**Average**	**Maximum**	**Standard Deviation**
Foligno	**ESP**	210	214.2	148	177	187.3	214.2	31.03
	BOP	212.7	173.1	165.8	204	188.9	212.7	22.93
VC	**ESP**	137.6	143.9	188	183.8	163.3	188	26.25
	BOP	137.9	166.9	205.3	146.3	164.1	205.3	30

The results in table 6.7 do not show the expected improvement, which would be search times more close to the average. Although the registered times where not that far apart from each other in the Foligno map, the same did not happen in the VC map. This means that partitioning the map according to the number of buildings will may result in balanced partitions, but the search times do not match the balancing. To explain this we'll consider two partitions with similar values of area and buildings. In each partition there will be an agent in charge of searching its partition. The agent that finishes more quickly is the agent placed in the partition that allows to check more buildings without actually having to enter them. Figures 6.19 and 6.20 show the two distinct cases. In figure 6.19 there are four buildings in the viewing range of the agent, and thus all can be inspected in a single cycle. In figure 6.20 there is the opposite case, in which the agent must enter all the buildings in the figure, and lose a cycle for each.

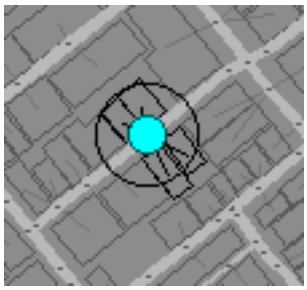

Figure 6.19: Buildings close to each other

Figure 6.20: Buildings far from each other

The goal of the partition methods developed was to create partitions that would be balanced, in a sense that the agents would take approximatelly the same time to visit their respective partitions. The results shown in table 6.7 proved that the number of buildings was an imprecise metric to use in the division of the map due to the reasons presented. Considering that an agent rarely takes more than one cycle to reach its destination, and that it must stop in order to receive the sensory information, then for the partitions to be balanced, the agents would have to stop a similar number of times to observe buildings. Each position in the map where an agent can observe any number of buildings is called an Observation Point (OP), and it can be a node, a road, or a building. By dividing the map according using OP as a metric for the balancing, then the resulting partitions will take the agents approximate times to search the partition completely.

The first step in the creation of the partitions is to determine what are the OP in the map. The problem of determining the smallest set of OP is the same as the Minimum Dominating Set problem which is NP-hard [21], and thus it is not solveable in polonomial time. To solve this problem an approximation was used. The set of OP is determined by applying the search method only without having the agents travel through the map. The search begins at a middle node, and proceeds until it identifies all the stoping points that allow a complete search of the map. Figure 6.21 displays a possible set of OP for the Kobe map.

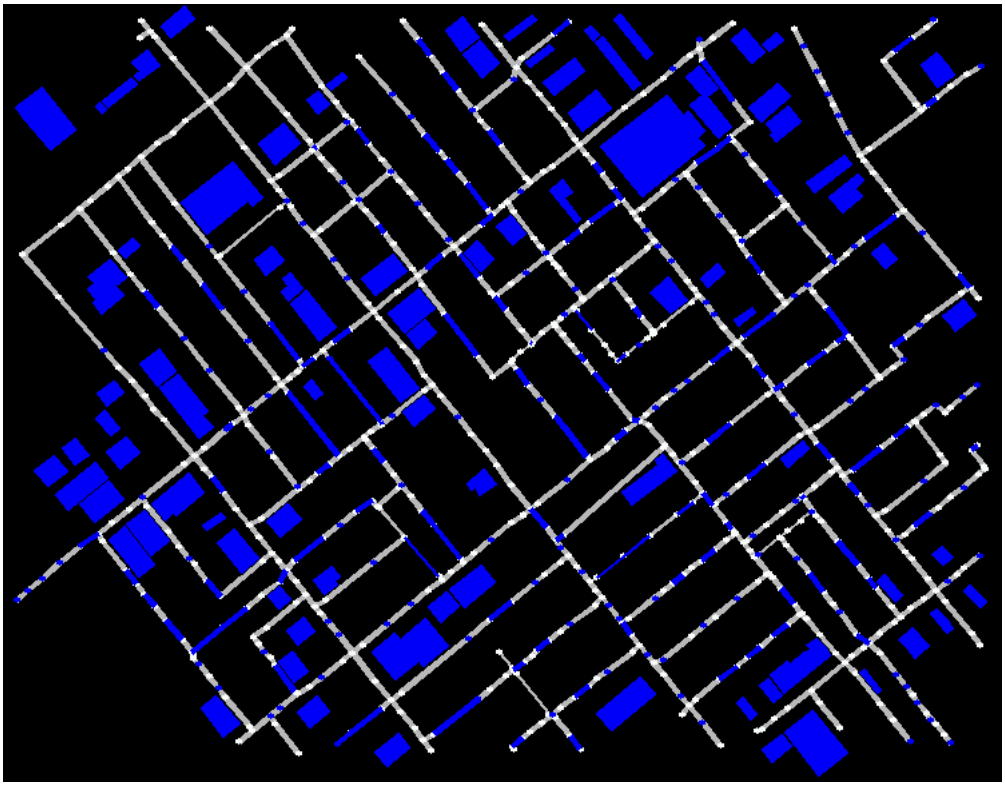

Figure 6.21: Observation Points for the Kobe map

After having all the OP in the map, the process continues with the partition of the map according to those OP. For the partition itself the same method used for the BOP was applicable with the OP. Only a slight change was needed due to the fact that roads can be an OP. When the partitioning algorithm selects a node to add to a given partition, it has to

check the edges that are connecting that node to neighboring nodes to see if they are OPs. If any of the edges is an OP then it has to add the edge, as well as the neighbor that the edge is connecting the node to.

6.6.1.5 Partition Refinement

The purpose of obtaining more balanced partitions with the previous two methods was achieved. However its balancing could still use some improvement. To that end a partition refinement algorithm was created for each method. The refinement will only be applied if a threshold is passed. The threshold is the difference between the weight of the partition with the biggest weight, and the partition with the smallest weight. If the difference is over a pre-established value, then the refinement process will be executed. Each method had its own threshold values, with 5 buildings for BOP and 3 OP for OPP.

Should refinement be necessary then there will be an exchange of units of either metric. The partition with the most units will pass a number of units to the adjacent partition with the least units. This second partition must not have lost any node in previous iterations. The reason for this is to prevent cycles in which two partitions exchange nodes between them continously. The number of swapped units is half of the difference between those partitions. First a selection of border nodes is made. Then the algorithm selects the minimum nodes that will fulfill the number of units required, and adds them to the adjacent partition, altering the ownership of the units that will come with them. After that a new evaluation of the balance between partitions is required to determine if further refinement is necessary.

The following pseudo-code demonstrates the general procedure of the refinement algorithm.

```
partitions[] = list of partitions
do {
    largePartition = partition with greatest weight
    smallPartition = partition with smallest weight
    partitionDifference = |largePartition| - |smallPartition|
    if (partitionDifference > threshold) {
      adjacentPartitions[] = empty list
      for (each partition p in partitions[]) {
          if (p is adjacent to largePartition)
              adjacentPartitions.add(p)
      }
      smallAdjPartition = adjacent partition with smallest weight
      unitsToTrade = (|largePartition| - |smallPartition|) / 2
      while (unitsToTrade > 0) {
          borderNodes = selectBorderNodes(largePartition,smallPartition)
          nodesToTrade = selectNodesWithMostUnits(borderNodes)
          nrUnitsTraded = numberUnitsInSelectedNodes(nodesToTrade)
          switchNodesBetweenPartitions(largePartition, smallPartition,
                                                          nodesToTrade)
          unitsToTrade = unitsToTrade - nrUnitsTraded
      }
    }
```

```
} while (partitionDifference > threshold)
```

Again, as mentioned in section 6.6.1.3, a special attention is needed because of alleys. When choosing the nodes that will be passed to the adjacent partition, the selection is based on the number of units of each node. However, if a node has alleys, the units present in those alleys must be added to the node's units.

The process is generally the same to refine either BOP or OPP, but with the second method it is necessary an extra precaution like with the initial formation of the partitions, and for the same reason. When selecting which nodes will provide the greatest number of OP to exchange between the partitions, the algorithm must be careful and check if the node being checked has any edges that are part of the OP set. Since the algorithm will only exchange the nodes after it has a subset of nodes that allows it to respect the rule of trading a number of units that is half of the difference of weights between the partitions it has to keep a record of the OP already accounted for. This list of OPs will prevent the algorithm from counting the same OP more than once, and also prevent it from expanding to nodes already checked. As with the alleys, the number of OPs obtained are associated with the node that initiated the expansion.

6.6.1.6 Results

Table 6.8 compares the results obtained with all the partitioning methods. These results were obtained after 10 simulations in each map, which was divided into 4 partitions with one agent in each partition.

Table 6.8: Results comparing average buildings visited with all partitioning methods

		Number of cycles to complete partition search						
		Time needed to finish each partition				**Average**	**Maximum**	**Standard Deviation**
Foligno	**ESP**	210	214.2	148	177	187.3	214.2	31.0
	BOP	212.7	173.1	165.8	204	188.9	212.7	22.9
	OPP	199.8	183.6	179.5	194.2	189.3	199.8	9.4
VC	**ESP**	137.6	143.9	188	183.8	163.3	188	26.3
	BOP	137.9	166.9	205.3	146.3	164.1	205.3	30
	OPP	163	172.4	162.9	164.3	165.7	172.4	4.5

The results in table 6.8 show that the final method obtained more even search times in all partitions as was expected. Also worth mentioning is that the number of buildings in each partition created with OPP was not even approximated which corroborates the idea that the number of buildings is not directly connected to the time necessary to search the map. Table 6.9 shows the number of buildings in each of the partitions created with the three methods.

Table 6.9: Number of buildings in each partition created with the different partitioning methods

Number of buildings in each partition					
		Part 1	**Part 2**	**Part 3**	**Part 4**
Foligno	**ESP**	290	358	208	234
	BOP	273	271	274	273
	OPP	306	222	249	304
VC	**ESP**	198	405	262	401
	BOP	318	317	317	314
	OPP	363	245	347	311

Searching the map with it divided into partitions also improved the overall search time, when compared to a simulation with four agents searching an unpartitioned map. This happens because the problem described in 6.5.2.2 is significantly reduced when the agents are acting on a reduced environment like the one provided by a partition. Table 6.10 shows results obtained from ten simulations on each map, unpartitioned, using a total of four agents for each map.

Table 6.10: Results comparing average search time between partitioned maps and unpartitioned maps

Map	**Kobe**	**VC**
Buildings In Map	737	1263
Partitioned map	142.8	172.4
Unpartitioned map	178.4	287.2

The effects are more noticeable on the VC map because it is larger. The Foligno map was also used to collect results, but the agents were unable to finish the map in any of the simulations, while on a partitioned map it took an average maximum of 199,8 cycles.

Despite the obvious evolution the last two methods have a common disadvantage that is the running time required to create the partitions, with the OPP method being slower than the ESP method because it has to calculate the set of OP. That time is worsened by the execution of the refinement algorithm. The ESP method is quite faster, but as was explained in section 6.6.1.2 it may return empty partitions, which doesn't happen in any of the other two methods. To solve the time problem there are several solutions. The first one is to use the partitions unrefined. With BOP this makes a greater difference than with OPP, but it also makes the partitions less balanced. Another solution is to limit the number of iterations that the refinement can take. A third solution is to have the Center, when available, calculating the partitions while the platoon agents search the map. The first two solutions have the reduce the running time, and the third just relieves the platoon agents of that load, but all three operate during a simulation. There is a last solution in case it is possible to run a map before executing a simulation. If that is the case then the partitions can be created offline and then stored in a file that will be read by the agents during the start of the simulation. Creating the partitions from a file is very fast and no cycles are lost with

this process. Limiting the number of iterations for the refinement and saving the partitions on a file were the solutions implemented. The partitions used to obtain the results shown here where created from files where the partitions were saved. Those files recorded partitions created with unlimited iterations on the refinement algorithm.

6.6.2 Communication

Communication is one of the primary tools for cooperation and coordination. It can and should be used for everything from sharing sensory data, to coordinate team efforts for rescuing civilians. In this work, which was more centered in search methods, the communication method was developed to transmit sensory data in simulations with no Ambulance Centers available.

During search the agents travel to unvisited locations. If another teamate has already search a given location that another agent hasn't visited, then it is a waste of time for the second agent to travel to that location, specially if there are no civilians trapped there. Another information of great relevance is the location of civilians, and blockades. When searching, these are the fundamental informations that the agents require.

To recall the constraints that the communication between RCR agents currently has, a field agent can only read four messages in a single cycle. It can send as many as it wants. Ambulance Centers have a greater communication capacity, and can read up to $2 \times n$ messages, where n is the number of Ambulances present in the simulation. The other constraint is the capacity of each message, 256 bytes. This means that it may not be possible to send all the intended data in a single message, and it may be necessary to select carefully what data to send.

With the strategy explained in 6.4 the number of messages available is not a big problem. The same cannot be said for teams with more than four agents each, or of strategies that use information regarding the entire map.

The messages that the agents send are not guaranteed to reach their intended destination(s). For that reason it would be better to, at any moment, send the complete WM of each agent. Unfortunately this is not possible because there is not enough space available in each message. The number of maximum messages the agents can receive also prohibit the segmentation of the WM. The solution is to send only new data collected at each cycle. If the information gathered in a cycle exceeds the maximum amount of data, then the agent will send what it can, and save the remaining data to send in the following cycle. Another option was to send another message, but it is ill advised, as it would increase the total number of messages being sent, and cause the agents to drop some of the messages because their limits would have been reached. Another reason not to send a second message is because it would hardly carry as much data as a single message from another agent, which might not be read because of the two part message.

With a solution for the problem of message space, we are still left with the problem of lost messages, which can be solved in two ways: the agents assume that a message allways reaches its destination; each agent expects a message from each of its teammates per cycle and if it does not receive one of the messages then it will send a new message to the agent that failed to send a message reporting the error and asking for the message to be resent. After several simulations, experimenting radio communication between the agents, no message was lost. For that reason the first strategy was chosen for implementation. The possibility of a message being lost is still real, but its effects had little impact.

Chapter 7

Conclusion

For this work several methods of graph partitioning and refinement were implemented, together with a search method that was constantly improved, a rescue method for single agent scenarios, a communication method for exchanging sensory data, and a new tool for agent development that could be of great use for rescue developers. The search method and the partitioning methods were presented at the 2008 edition of the I-ROBOT workshop, an international workshop on robotics and artificial intelligence, included in the Ibero-American conference, IBERAMIA [7].

The different strategies and methods presented here provided good results and allowed to obtain an extensive knowledge of the inner workings of the RCR simulator, and also rescue strategies, multiagent development and debugging, multiagent communication and cooperation, graph algorithms for creation, partitioning and refinement. Even though the work was focused on the needs of Ambulances, the methods developed for searching, map partitioning and communication can be used with any of different types of RCR agents, which represents an advantage for further development of Policemen or Firefighters. Specially because having agents performing similar methods helps the development of cooperation between the agents.

The rescue method also returned some positive results, and good pointers for future versions, more complex and evolved. Nevertheless, the results were taken in simulations deprived of roadblocks and burning buildings, which means that in simulations complete with the adversities that come from blockades and fires the casualties would be higher. Although in such simulations there would be Firefighters and Policemen to help overcome those obstacles.

Regarding the initial objectives set for this work, not all were achieved. The work focused mainly on developing an efficient search method, and explored several methods of partitioning the map for multi agent scenarios. After obtaining satisfying methods that provided good results for searching and map division, the development of cooperation, communication and rescue strategies began. Because of the lack of time these methods did not evolve from simple and effective methods, but not very efficient. Time was not the only reason for the lack of development in those areas. Initially the idea was to improve pre-existing strategies. Unfortunately this was not the case due to several factors. The first one was the lack of existing code for the current simulator of that time (0.49+), despite the efforts of contacting other developer teams. Another problem encountered was the lack of code in C++, the chosen programming language for this work. Most of the code found at the time was for the previous version of the simulator, and it was all in Java. For that reason the Ambulance agent was

developed from a Civilian agent, which features the most basic of actions, moving.

7.1 Future Work

The results obtained from this work were focused on the development of an efficient search method. This means that any future work should be more concentrated on developing rescue strategies and cooperation techniques, without completely overlooking the search method. A useful modification would be to make the search method more aware of unvisited map areas close to the agent's location, a problem described in section 6.5.2.2. Another item for future work is to add to the search method the means to deal with roadblocks. A feature discarded at the time since roadblocks were a problem for Policemen, and should be dealt when this type of agents was added to the development. The algorithms for partitioning and refinement should also be optimized to make them less time consuming in order for them to be used during a simulation without any loss of cycles, or with an acceptable number of lost cycles.

The rescue method was developed for a single agent scenario, where there is little or no cooperation included. Also, since this work only envolved Ambulances it was not designed for the possibility of burning buildings. For that reason a more efficient and reliable method should be implemented. One that uses the agent's capacity for cooperation to its full extent. Such a method should also include a lifetime prediction mechanism, and a rescue schedulling algorithm.

At the core of both search and rescue strategies should be a cooperation method that maximizes the resources available, which must include a communication protocol that deals with the limitations in the best way possible. Although there is a cooperation and communication method developed for the Ambulances, it is still very raw, and is oriented for search tasks alone.

The cooperation is more successfull if there is a Center available. In this work no Centers were developed, but the partitioning and communication methods can be used by the Center without the need for alterations. Nevertheless, the Center can be very useful and it is a very important agent for cooperation and communication, besides beeing the key agent for communication between agents of different types. For these reasons it should be considered a priority to include an Ambulance Center in future developments.

Any new developments should also be oriented to the two scenarios described, single and multiagent. Even including the means for the Ambulances to overcome obstacles like roadblocks, and burning buildings. Although the RCR world is mainly a MAS, there are still simulations without communication, and since it intends to mimic real life, such inconveniences are likely to occur, and because of that should also be taken seriously when developing RCR agents.

As was detailed in section 6.1.1.1 the Agent Viewer can, and should, have more features added to it, making it an even more valuable tool for agent development.

To finish, another consideration regarding future work. All major teams have their agents developed in Java. Even though the languages Java and C++ are conceptually the same, the libraries provided for one and the other, and the general structure of the agents are very different. Since there should be collaboration between the developer teams, developing the agents in a language not very used by other teams only makes the collaboration harder. More so considering the differences between the libraries. For this reason it is my opinion that further developments should be made in Java.

Bibliography

[1] SDL - Simple Directmedia Layer. online available at: http://www.libsdl.org/.

[2] FIPA ACL message structure specification. online available at: http://www.fipa.org/specs/fipa00061/index.html, 2002.

[3] FIPA communicative act library specification. online available at: http://www.fipa.org/specs/fipa00037/, 2002.

[4] Robocup rescue wiki: 2007 competition results. online available at: http://www.robocuprescue.org/wiki/index.php?title=Agent2007results, 2007.

[5] Robocup rescue wiki: 2008 competition results. online available at: http://www.robocuprescue.org/wiki/index.php?title=RSL2008, 2008.

[6] H. Levent Akin and Ergin Ozkucur. Robocuprescue 2008 rescue simulation league team description RoboAkut (Turkey). In *RoboCup Symposium, 2008. Proceedings CD-Rom*, 2008.

[7] Pedro Alves, Nuno Lau, and Luís Paulo Reis. Map visiting in robocup rescue simulations. In *IROBOT 2008 3rd International Workshop on Intelligent Robotics*, pages 49–60, 2008.

[8] Thiago Assunção and Vasco Furtado. A heuristic method for balanced graph partitioning: An application for the demarcation of preventive police patrol areas. In *Advances in Artificial Intelligence - IBERAMIA 2008, 11th Ibero-American Conference on AI, Lisbon, Portugal, October 14-17, 2008. Proceedings*, volume 5290 of *Lecture Notes in Computer Science*, pages 62–72. Springer, 2008.

[9] Jonathan W. Berry and Mark K. Goldberg. Path optimization for graph partitioning problems. *Discrete Applied Mathematics*, 90(1-3):27–50, 1999.

[10] Dinesh Bhatia. Circuit partitioning. Partitioning Chapter Slides for the Design Automation of VLSI Systems course, at the University of Texas, Dallas, online available at: http://www.utdallas.edu/ dinesh/teaching/PDA/3-Partitioning.pdf, 2006.

[11] Carlos S. N. Brito, Daniel L. Baggio, Humberto S. Naves, Jackson P. Matsuura, Rafael A. B. Barros, Leandro M. Groisman, and Laura Bolignini. Itandroids-rs team description paper. In Brazil Instituto Tecnológico de Aeronáutica São José dos Campos, SP, editor, *RoboCup Symposium, 2006. Proceedings CD-Rom*, 2006.

[12] João Certo and Nuno Cordeiro. Search and rescue in urban catastrophes. In *Graduation Project: Faculdade de Engenharia da Universidade do Porto*, 2005.

[13] Francesc Comellas and Emili Sapena. A multiagent algorithm for graph partitioning. In *EvoWorkshops*, pages 279–285, 2006.

[14] Robocup Rescue Technical Committee. Robocup rescue simulator manual version 0 revision 4. online available at: http://www.robocuprescue.org/docs/robocup-manual-v0-r4.pdf, 2000.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2nd edition, 2001.

[16] Marco Dorigo and Thomas Stutzle. *Ant Colony Optimization*. MIT Press, 2004.

[17] Habibi et al. Impossibles robocup rescue 2006 team description paper. In Department of Computer Engineering Sharif University of Technology, editor, *RoboCup Symposium, 2006. Proceedings CD-Rom*, 2006.

[18] Habibi et al. Impossibles 2007 team description: Robocup rescue simulation. In Department of Computer Engineering Sharif University of Technology, editor, *RoboCup Symposium, 2007. Proceedings CD-Rom*, 2007.

[19] Nasrin Mostafazadeh et al. Poseidon team description paper. In Tehran Farzanegan Highschool, editor, *RoboCup Symposium, 2006. Proceedings CD-Rom*, 2006.

[20] S. D. Ramchurn et al. Aladdin - rescue team description. In School of Electronics IAM Group and UK Computer Science, University of Southampton, editors, *RoboCup Symposium, 2007. Proceedings CD-Rom*, 2007.

[21] Fabrizio Grandoni. A note on the complexity of minimum dominating set. *J. Discrete Algorithms*, 4(2):209–214, 2006.

[22] Gregory Gutin and Abraham P. Punnen. *The Travelling Salesman Problem and its Variations*. Kluwer Academic Publishers, 2002.

[23] Michel Habib, Christophe Paul, and Laurent Viennot. A synthesis on partition refinement: A useful routine for strings, graphs, boolean matrices and automata. In *STACS*, pages 25–38, 1998.

[24] Seyed Hamid Hamraz and Seyed Shams Feyzabadi. Iust robocup rescue simulation agent competition team description. In Iran University of Science Center of Scientific Innovation and Iran Technology, Tehran, editors, *RoboCup Symposium, 2006. Proceedings CD-Rom*, 2006.

[25] Bruce Hendrickson and Robert W. Leland. A multi-level algorithm for partitioning graphs. In *SC*, 1995.

[26] Michael Huhns and Larry Stephens. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter 2, pages 121–164. The MIT Press, 1999.

[27] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[28] Majid Ali Khan, Linus Luotsinen, and Ladislau Boloni. Robocup rescue 2007 - rescue simulation league team goldenknight. In School of Electrical Engineering and USA Computer Science, University of Central Florida, editors, *RoboCup Symposium, 2007. Proceedings CD-Rom*, 2007.

[29] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. Robocup: A challenge problem for ai and robotics. In *RoboCup*, pages 1–19, 1997.

[30] Hiroaki Kitano and Satoshi Tadokoro. Robocup rescue: A grand challenge for multiagent and intelligent systems. *AI Magazine*, 22(1):39–52, 2001.

[31] A. Kleiner and M. Gobelbecker. Rescue3d: Making rescue simulation attractive to the public. online available at: http://kaspar.informatik.unifreiburg.de/∼rescue3D/3dview.pdf.

[32] Alexander Kleiner, Michael Brenner, Tobias Bräuer, Christian Dornhege, Moritz Göbelbecker, Mathias Luber, Johann Prediger, Jörg Stückler, and Bernhard Nebel. Successful search and rescue in simulated disaster areas. In *RoboCup*, pages 323–334, 2005.

[33] Ramachandra Kota, Ravindranath Jampani, and Kamalakar Karlapalem. Kshitij: Team description. In *RoboCup Symposium, 2005. Proceedings CD-Rom*, 2005.

[34] Nuno Lau and Luís Paulo Reis. FCPortugal Rescue Team site. online available at: http://paginas.fe.up.pt/∼lpreis/fcportugal/rescue/, 2005.

[35] Nuno Lau, Luís Paulo Reis, and Francisco Reinaldo. FCPortugal 2005 rescue team description: Adapting simulated soccer coordination methodologies to the search and rescue domain. In *RoboCup Symposium, 2005. Proceedings CD-Rom*, 2005.

[36] Nuno Lau, Luís Paulo Reis, Francisco Reinaldo, and João Certo. FCPortugal: Development and evaluation of a new robocup rescue team. In *Proc. First IFAC Workshop on Multivehicle Systems (MVC'06)*, 2006.

[37] Takeshi Morimoto. Robocup rescue morimoto viewer. online available at: http://ne.cs.uec.ac.jp/ morimoto/rescue/viewer/.

[38] Takeshi Morimoto. vow to develop a robocup rescue agent, 1st editiov. online available at: http://ne.cs.uec.ac.jp/∼morimoto/rescue/manual/index.html, 2002.

[39] Timo A. Nüssle, Alexander Kleiner, and Michael Brenner. Approaching urban disaster reality: The resq firesimulator. In *RobuCup*, pages 474–482, 2004.

[40] Sébastien Paquet, Nicolas Bernier, and Brahimc Chaib-draa. Damas-rescue description paper. In *RoboCup Symposium, 2004. Proceedings CD-Rom*, 2004.

[41] Juan Peng, Xiaoyong Zhang, Fu Jiang, Chenyang Ding, Ya Liu, and Sheng Zhou. Csu yunlu robocup rescue team description. In China Central South University, editor, *RoboCup Symposium, 2006. Proceedings CD-Rom*, 2006.

[42] Gopal Ramchurn, Yoannis Vetsikas, and Nobuhiro Ito. Robocup 2008 rescue simulation league - agent competition - rules and setup. online available at: http://www.robocuprescue.org/wiki/images/Rules2008-draft.pdf, 2008.

[43] S. D. Ramchurn, M. Allen-Williams, I. Vetsikas, A. Rogers, R. K. Dash, P. Dutta, and N. R. Jennings. Aladdin - rescue team description. In School of Electronics IAM Group and UK Computer Science, University of Southampton, editors, *RoboCup Symposium, 2006. Proceedings CD-Rom*, 2006.

[44] Luís Paulo Reis, Nuno Lau, and Eugenio Oliveira. Situation based strategic positioning for coordinating a team of homogeneous agents. In *Balancing Reactivity and Social Deliberation in Multi-Agent Systems*, pages 175–197, 2000.

[45] S. J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[46] Christos Sioutis and Jeffrey Tweedale. Agent cooperation and collaboration. In *KES (2)*, pages 464–471, 2006.

[47] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *ICRA*, pages 3310–3317, 1994.

[48] Peter Stone and Manuela M. Veloso. Task decomposition, dynamic role assignment, and low-bandwidth communication for real-time strategic teamwork. *Artif. Intell.*, 110(2):241–273, 1999.

[49] Tomoichi Takahashi, Ikuo Takeuchi, Tetsuhiko Koto, Satoshi Tadokoro, and Itsuki Noda. *RoboCup Rescue* disaster simulator architecture. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 379–384, 2000.

[50] Kuniyoshi Toda, Yohei Kaneda, and Nobuhiro Ito. The cooperative behaviors for rescue agents. In Department of Electrical and Nagoya Institute of Technology Computer Engineering, editors, *RoboCup Symposium, 2006. Proceedings CD-Rom*, 2006.

[51] Masafumi Ueda. Robocup rescue "facelifted" viewer. online available at: http://ne.cs.uec.ac.jp/ masa-u/viewer/.

[52] David Vernon. *Machine Version: Automated Visual Inspection and Robot Vision*. Prentice Hall, 1991.

[53] José M. Vidal. *Fundamentals of Multiagent Systems: Using NetLogo Models*. online available at: http://multiagent.com/files/mas-20070824.pdf, 2007.

[54] Denis Fillipini Vitti, Jackson P. Matsuura, Rafael A. B. Barros, Marcela Sobrinho, Kely Marques Rosa, and Acrisio Domiciano Dias. Itandroids-rs team description paper. In Brazil Instituto Tecnológico de Aeronáutica São José dos Campos, SP, editor, *RoboCup Symposium, 2007. Proceedings CD-Rom*, 2007.

[55] Wikipedia. Breadth-first search. online available at: http://en.wikipedia.org/wiki/Breadth-first_search, 2008.

[56] Wikipedia. Depth-first search. online available at: http://en.wikipedia.org/wiki/Depth-first_search, 2008.