

UNIVERSITY OF ABERDEEN

Department of Computing Science

Honours Degree

Report

SENSOR ASSIGNMENT
IN A VIRTUAL ENVIRONMENT
USING CONSTRAINT PROGRAMMING

Supervisor:

Dr. ALUN PREECE

Student:

DIEGO PIZZOCARO

Co-supervisor:

Dr. STUART CHALMERS

ACADEMIC YEAR 2006-2007

Contents

Abstract	vi
Declaration	vii
Preface	viii
1 Introduction and Motivations	1
1.1 Context: the ITA project	1
1.2 Objective	2
1.3 Motivations	3
1.4 The virtual environment: “Battlefield 2”	4
2 Related Work	7
2.1 Previous works on sensors	7
2.2 The project “Plan and Play”	8
2.3 Constraint Satisfaction Problem and Constraint Programming	10
2.3.1 Definitions: CSP and CP	10
2.3.2 The eight queens problem	11
2.3.3 The Knapsack Problem	13
2.3.4 The Multiple Knapsack Problem	16
3 Concept and Design	19
3.1 System Architecture	19
3.2 Modeling the Sensor Deployment as a CSP	21

3.3	Reformulation using the multiple knapsack problem	25
3.3.1	Sensor Assignment	26
3.3.2	Sensor Deployment	32
3.4	Modeling considerations	34
3.4.1	Heuristic	34
3.4.2	Flexibilities of the “Sensor Assignment” model	36
4	Implementation and Testing	39
4.1	Implementation	39
4.1.1	Technologies Used	39
4.1.2	Webservice Implementation	40
4.1.3	Commander’s Interface Implementation	42
4.1.4	“Battlefield 2 Mod” Implementation	43
4.2	Testing and Evaluation	44
5	Conclusion and Future works	46
5.1	Conclusion	46
5.2	Future works	47
5.2.1	Improving solver: Relaxed Constraints	47
5.2.2	Objective function improvement	47
5.2.3	Dealing with multiple mission	48
5.2.4	Integration with “Plan And Play”	48
	Bibliography	49
A	User Manual	51
A.1	Starting the system	51
A.2	Using the system	52
A.2.1	Using the Commander’s Interface	53
B	Maintenance Manual	59
B.1	Installation Instructions	59
B.2	System Execution	61

B.3	Hardware and Software dependencies	62
B.3.1	Hardware dependencies	62
B.3.2	Software dependencies	63
B.4	Space and Memory requirements	63
B.5	Source File Description	64
B.5.1	Webservice source description	64
B.5.2	Commander Interface source description	66
B.5.3	Battlefield 2 Mod source description	67
B.6	Compiling and Updating the system	68
B.6.1	Compiling the Webservice	68
B.6.2	Compiling the Commander's Interface	69
B.6.3	Updating the Battlefield 2 Mod	70
B.7	Known Bugs	70
B.7.1	Battlefield 2 Mod Bug	70
B.7.2	Webservice "Solver" Bug	71

List of Figures

1.1	A graphic representation of the “Sensor Assignment” problem. . .	3
1.2	Map of BF2 on the left, an example of playing on the right. . .	5
2.1	A possible solution to the 8 queens problem.	12
2.2	The knapsack problem.	14
2.3	Multiple Knapsack Problem	17
3.1	System Architecture	20
3.2	Representation of a sensor and a zone in the first model.	22
3.3	Representation of the Sensor Assignment Problem in the final model.	27
3.4	Analogies between the multiple knapsack and the Sensor As- signment problem.	28
3.5	Representation of the Sensor Deployment Problem in the final model.	32
3.6	An example of application for the Sensor Deployment Algorithm.	35
3.7	The “Sensor Assignment” model without the heuristic.	36
A.1	Locating webservice.	53
A.2	Inserting zone side.	54
A.3	The side used by the system.	55
A.4	Selecting zones.	56
A.5	Creating zones.	56
A.6	Creating Sensors.	57

A.7	Solution sent by the webservice.	57
A.8	The message that appears entering a sensor area in BF2.	58

Abstract

This project is part of the ITA project that is an IBM international project which addresses researches in the area of network-centric coalition operations. In particular the aim of our project is to satisfy the commander's needs of information in a certain zone of a map, for a military/rescue mission, finding a method to decide which is the optimal deployment of a set of sensors with certain capabilities in some zones selected in a map by the commander. Our attention will be directed to find an efficient mathematical model for the problem, so that we will be able to apply the Constraint Programming paradigm to find a solution to the problem.

Declaration

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Diego Pizzocaro

Date _____

Signed _____

Preface

In **Chapter One** we give a brief introduction on the context of the project and also on the motivations that brought us to begin such a project. We also try to give a motivation behind the choice of “Battlefield 2” as the virtual environment where to test things.

In **Chapter Two** we present the results of our research of previously published relevant document about the problem that we are going to face, that is an optimal assignment of sensors in a map using Constraint Programming. We also give a brief introduction on Constraint Satisfaction Problems and Constraint Programming by presenting also some relevant examples.

In **Chapter Three** we give an overview of the system architecture and then, after the description of a not-working first model developed to solve the problem, we describe carefully the final innovative model created to find an optimal deployment of sensors inside a map.

In **Chapter Four** we briefly described how we implemented the system and we try also to evaluate how good is our solution.

In **Chapter Five** we conclude our project description with some comments on how everything was expected to go and how effectively went. Finally we also present some opportunities for future works.

Chapter 1

Introduction and Motivations

Reading this chapter should give you a sufficient idea of the context of this project giving a brief overview of the project, including also motivation behind it and its objectives.

1.1 Context: the ITA project

This project is part of the ITA project which is an international project led by IBM¹ that addresses researches in the area of network-centric coalition operations. ITA stands for International Technology Alliance in Network and Information Sciences, and it involves many university in the USA and in the UK but also the defense departments of USA and UK, since the result of these studies should be applied to military/rescue missions.

There are four main research areas defined by the alliance: (i) network theory, (ii) security across system of systems, (iii) sensor information processing and delivery, and finally (iv) distributed coalition planning and decision making. The department of Computing Science of the University of Aberdeen² is contributing mainly in (iii) and (iv), researching respectively:

¹website: www.ibm.com

²ITA project at Aberdeen: <http://www.csd.abdn.ac.uk/research/ita/webpages/projects.html>
(last checked 06/05/2007)

1. Knowledge technologies for sensor information processing and delivery
2. Agent technologies for distributed coalition planning and decision making

This project is located inside the research field (1) that is in the area of sensor researches, in particular its aim is to help the commander of a mission to define needs of intelligence in certain zones of a map and then to provide the commander with the optimal deployment of the available sensors inside these zones. With regards to the ITA project, the most important aspect faced by this "Sensor Assignment" project is the scheduling of the capabilities of a sensor depending on which are the requirements given by the commander. So for example if there is need of VIDEO information in a certain zone, but the remaining sensor are all AUDIO and VIDEO sensors, then it will be possible to enable only the VIDEO ability of the sensor, in this way the autonomy of the sensor and also its performance can be increased. Indeed with only VIDEO enabled the sensor can decide to improve video quality, wasting the same energy spent with A/V enabled, or it could decide to maintain the same VIDEO quality, so that it will spend less energy and in this way the battery will last longer.

Reading this report you will also note that while developing an automated reasoning technique that can generate a "strategic" assignment of sensors to cover the requirements of a mission, we also created a model of the sensor and of the zone of a map, that in the future could be used as a base to create a sensor and a zone ontology.

1.2 Objective

Our project aims, as stated above, to develop an application to help the commander to define needs of information in certain zones and then that gives in output an optimal deployment of the available sensors, each with a schedule of their capabilities depending on the zone to which their are assigned to. So the problem, which is represented in Figure 1.1, is that we are given a set of

sensors each one with its own capabilities and a set of zones, each with its own type of information needed and we have to find an optimal deployment. With "optimal sensor deployment" we mean that the total area covered by the sensors has to be maximized, anyway we will see later that the concept "optimal" can be very flexible in our solution, since our theoretical approach to the problem is very flexible too.

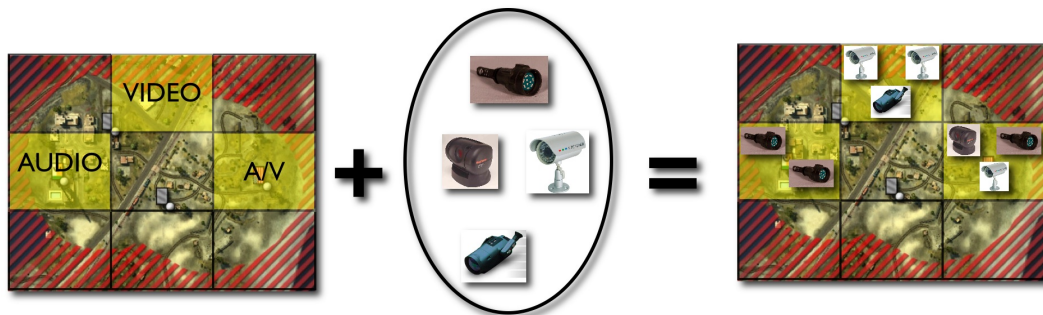


Figure 1.1: A graphic representation of the "Sensor Assignment" problem.

The project aims also to simulate a real world scenario, where we can simulate to deploy the sensors for real. So we decided to use a virtual environment where we can actually deploy sensors and schedule their capabilities, and our choice for the virtual environment is the virtual world of the PC game "*Battlefield 2*". But our application, as we will see, will be completely independent from this particular game, so that in the future the virtual environment could be changed or could be substituted by the real world.

1.3 Motivations

The motivations of our project are pretty much the same that lie behind the ITA project, so overall there is the need to eliminate the pitfalls of compartmentalized research in different technical areas³ and, through these researches,

³website: http://domino.research.ibm.com/projects/titans/www_titans.nsf/pages/proj.html, checked on 06/05/2007

to provide unique capabilities to the US Army and the UK Armed Forces which could be used either in military or in rescue missions.

Note that in every military/rescue operation the first step is to place some intelligence in strategic zones (as it is stated in [1]), so that the soldiers know how they should prepare properly for the current mission. Since the first step is to deploy sensors and since the result of the mission depends strongly on the information gathered by the sensors, then it is compulsory to provide a sensor deployment as good as possible, such that it is able to maximize the possibilities of success of a mission.

Let’s consider for example the case of a rescue mission in which some citizens got stuck in a flooded town. The commander will have to decide which are the relevant zones to the mission, let’s say that these are the most densely populated areas of the town. Then the commander will decide which type of information should be available in each area, let’s say AUDIO/VIDEO in some of them and in others only VIDEO. Our application will have to determine the optimal positions where to place the sensors and which capabilities of these sensors have to be enabled. Finally some soldier could place the sensors, and the mission could start by using the information gathered by the sensors to decide which is going to be the first zone that will be rescued. So for example in a VIDEO zone the commander could see through the sensors that it is easier to reach this zone since it is not already completely flooded.

1.4 The virtual environment: “Battlefield 2”

As we said in advance, since we want to simulate a real world scenario, we used as a virtual environment the world of the PC game “*Battlefield 2*”, that you can see in Figure 1.2. The motivations behind that choice are mainly two: the tradition of the armed forces to use this kind of games to train soldiers, and the architecture of the game.



Figure 1.2: Map of BF2 on the left, an example of playing on the right.

With regards to the first motivation, the US Army uses shooting online games (obviously with a private server) to train soldiers, i.e. they put a bunch of soldiers in a situation where they have to accomplish a mission and the commander monitors them observing the way in which they behave while playing. So this is not a “body” or a “reflexes” training but it is a way to improve the strategy applied by each team of soldiers to accomplish a mission. In conclusion we decided to use *“Battlefield 2”* since there is an high probability that soldiers already played a similar game and so if the system, developed during this project, should result really interesting with regards to the ITA project, then it could be also easily tested by the army.

The second motivation is the architecture of *“Battlefield 2”*. Indeed it is a shooting online game, that has the typical architecture of an online game: it has a server and a client. The players will play by starting the client of the game in their machine and then by connecting to the server that manages all the interactions with the environment and with other players. Furthermore it is “open”, which means that it is easy to gather data and events from the game, so we can actually know what is happening inside the game. This can be done by writing some code in Python that you can easily add to the game server. In fact this is another reason that led us to the choice of using *“Battlefield*

2”, because it is easy to create ”plug-in’s” for the server writing your own Python code. Such a ”plug-in” is called ”Mod” (that probably stands for ”modified” game) and as an evidence that it should have been quite easy to create a ”Mod” for this game there were already many ”Mod” available on the Internet, like *Project Reality*⁴ which was actually a complete revised version of ”*Battlefield 2*” (like a new different game) showing the infinite possibilities of editing this game.

⁴website: <http://www.realitymod.com/>

Chapter 2

Related Work

Here we give a brief description of the most relevant works that had been produced on the sensor deployment research area, even if as you will see only one paper is really relevant. We will also introduce a project, also developed in the Computing Science Department of the University of Aberdeen, which is closely related to this "Sensor assignment" project.

2.1 Previous works on sensors

Probably the most relevant work to this project is a paper published by IEEE entitled "*Decision-Theoretic Cooperative Sensor Planning*"[2]. This paper describes a decision-theoretic approach to cooperative sensor planning between autonomous vehicles executing a military mission. In other words there is a set of Unmanned Ground Vehicles (UGV) each with sensors installed, and they use intelligent cooperative reasoning to select optimal vehicle viewing locations and select optimal camera pan and tilt angles throughout the mission. This is a problem of optimization too, like our "sensor deployment" problem, indeed the decisions are made in such a way to maximize the value of information gained by the sensors while maintaining vehicle stealth.

As it can be noticed there are many common things with our project but also

many different aspects. For example the fact that sensors can move could bring someone to think that this is a big difference with our project, instead also our approach can be extended to the case where the sensors move, as we will see in the Chapter 3.

The most important thing in common with our project is that they try to compute an optimal location, camera pan and split that can maximize the covered area for each vehicle, so in this way they are trying to solve a problem that is similar to our problem. The difference is that they do not have a commander that specifies which zones have to be covered and which type of information is required, instead their commander will define the value of information gathered, for example a commander could decide that the value of information gathered about the kind of enemy forces that is in the area (tank, trucks, etc,) have to be the highest valued information, and so the UGV's will try to find a location from where they can maximize this type of information.

At the beginning of our project we also thought to use a similar way of giving a value to the information and then to deploy sensors maximizing this value. Anyway such a solution turned out to be too complex and it also involved some aspects that were irrelevant to this project.

Other papers on sensors are available, like in [3] or [4], but none of them face a problem similar to the one of deploying sensors in an optimal way. But it is to notice that [4] can be very relevant to the branch of the ITA project that involves sensor networks (which is more or less the same branch to which this project belongs to).

2.2 The project “Plan and Play”

The “Plan and Play” [5] is the single honours project of Daniele Masato who developed this project at the Computing Science Department of the Univer-

sity of Aberdeen. ”Plan and Play” is closely related to this project, since our ”Sensor assignment” project uses some concept and technologies developed in Masato’s project.

Plan and Play finds its collocation in the domain of e-Response, a group of network technologies designed to support emergency response operations as humanitarian relief and civilian population control. The project focuses on human and software agent interactions in a virtual environment where the collaboration between them is required in order to carry out a plan. So this project aims to track the progresses achieved in a plan by each player by mapping the plan and its various steps to states and objects within the virtual universe.

The links between ”Plan and Play” are the architecture of the system and the virtual environment used. Indeed the architecture of Masato’s project is based on a central component that is represented by a webservice written in Java, and as you will see in Section 3.1 also the ”Sensor Assignment” project is based on such a webservice.

Another important link between these two projects is the use of the same virtual environment that is *”Battlefield 2”*. The project ”Plan and Play” has implemented another webservice inside the game server that exchanges messages with the webservice written in Java but the ”Sensor Assignment” project didn’t need such a complicated ”Mod”¹ of the game, so as we will see in the Section 3.1 we only took the same structure of the ”Mod” (that is a standard for every Mod) and we eliminated the webservice inside the ”Mod” keeping only some useful methods to interact with the remaining part of the system.

¹See Section 1.4

2.3 Constraint Satisfaction Problem and Constraint Programming

This section describes what is a Constraint Satisfaction Problem (CSP) and what is Constraint Programming (CP) which are two concepts that we widely used throughout all this project. We also describe three different Constraint Satisfaction Problem that are really important with regards to the theoretical part described in this document.

2.3.1 Definitions: CSP and CP

The "sensor assignment" problem that we stated above in 1.2 can be seen as a *Constraint Satisfaction Problem (CSP)* that is a mathematical problem where one must find values for variables that satisfy a number of constraints. CSPs are the subject of intense research in both artificial intelligence and operations research. Many CSPs require a combination of heuristics and combinatorial search methods to be solved in a reasonable time².

The techniques used to solve a CSP depend on the kind of constraints being considered. Often used are constraints on a finite domain, to the point that constraint satisfaction problems are typically identified with problems based on constraints on a finite domain. Such problems are usually solved via search, in particular a form of backtracking or local search. Constraint propagation are other methods used on such problems; most of them are incomplete in general, that is, they may solve the problem or prove it unsatisfiable, but not always. Constraint propagation methods are also used in conjunction with search to make a given problem simpler to solve³.

Constraint programming is the programming paradigm where relations be-

²See http://en.wikipedia.org/wiki/Constraint_satisfaction_problem, last checked on 08/05/07

³http://en.wikipedia.org/wiki/Constraint_satisfaction, last checked on 08/05/07

tween variables can be stated in the form of constraints. Constraints differ from the common primitives of other programming languages in that they do not specify a step or sequence of steps to execute but rather the properties of a solution to be found.

The constraints used in constraint programming are of various kinds: those used in constraint satisfaction problems, those solved by the simplex algorithm, and others. Constraints are usually embedded within a programming language or provided via separate software libraries⁴. In our case we will use a Java software library called “CHOCO”⁵ that allows us to specify constraints that will be used by CHOCO itself to solve the “Sensor assignment” problem with its own methods used to solve CSP problems.

2.3.2 The eight queens problem

From what stated above, we can easily understand that to solve a CSP it is enough to specify its constraints with a programming language using some separate software libraries. So it is clear that the tricky part is not to solve the problem but to create a mathematical model for the problem, that is to define variables, the domain of the variables and the constraints for the problem.

We can understand this by analyzing a CSP problem called “the eight queens problem”, which is quite related to the “Sensor assignment” problem since it is a placement problem too. In the eight queens problem we have to place eight chess queens on a 8×8 chessboard in such a way that each queen does not attack another queen using the standard chess queen’s moves. This problem, as many CSPs, has more than one solution, and you can see one of this possible solution in Figure 2.1. So it is clear that a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general “n queens problem” of placing n queens on an $n \times n$ chessboard⁶.

⁴See http://en.wikipedia.org/wiki/Constraint_programming, last checked on 08/05/07

⁵For more information see Chapter 4.

⁶See http://en.wikipedia.org/wiki/N_queens_problem, last checked on 08/05/07

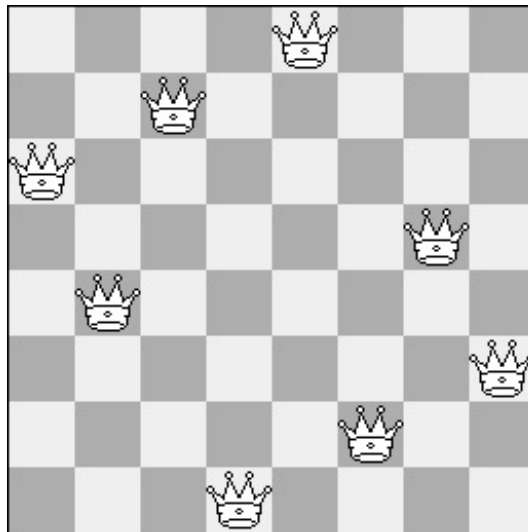


Figure 2.1: A possible solution to the 8 queens problem.

So now that we carefully defined an “*n-queens problem*”, we can try to find a mathematical model for this problem. The components of a model are always three:

1. *Variables* whose final values have to respect the constraints.
2. *Domain* of the values that variables can assume.
3. *Constraints* on the variables.

And in the case of the “*n-queens problem*” as presented in the book [6]:

Variables : x_1, \dots, x_n . where x_i denotes the position of the queen placed in the i th column of the chess board.

Domain for each variable : the integer values $[1, \dots, n]$.

- So for example, the solution presented in Figure 2.1 corresponds to the sequence of values (6,4,7,1,8,2,5,3), since the first queen from the left is placed in the 6th row counting from the bottom, and similarly with the other queens.

Constraints : They can be formulated as the following disequalities for

$i \in [1 \dots n - 1]$ and $j \in [i + 1 \dots n]$

- No two queens in the same row:

$$x_i \neq x_j \tag{2.1}$$

- No two queens in each South-West – North-East diagonal:

$$x_i - x_j \neq i - j \tag{2.2}$$

- No two queens in each North-West – South-East diagonal:

$$x_i - x_j \neq j - i \tag{2.3}$$

As you can see there is a big difference between an informal description of the problem and the mathematical model which represents it, indeed we have to reach a very high level of abstraction.

2.3.3 The Knapsack Problem

The knapsack problem is a CSP that derives its name from the maximization problem of choosing some items that can fit into one bag (of maximum weight) to be carried on a trip, you can see its graphic representation in Figure 2.2. An informal description⁷ of it could be that we are given a set of items, each with a cost and a value, and a knapsack with a given cost, then we have to determine which items to insert in the knapsack so that:

- the *total cost* of the chosen items is less than or equal to the knapsack's cost,

⁷Here we are actually considering a particular type of knapsack problem called 0-1 knapsack problem, since the items can be chosen ($x_i = 1$) or not chosen ($x_i = 0$). Instead a more general knapsack problem can also decide to take more than only one instance of the same object to insert inside the knapsack.

- the *total value* of the chosen items is as large as possible.

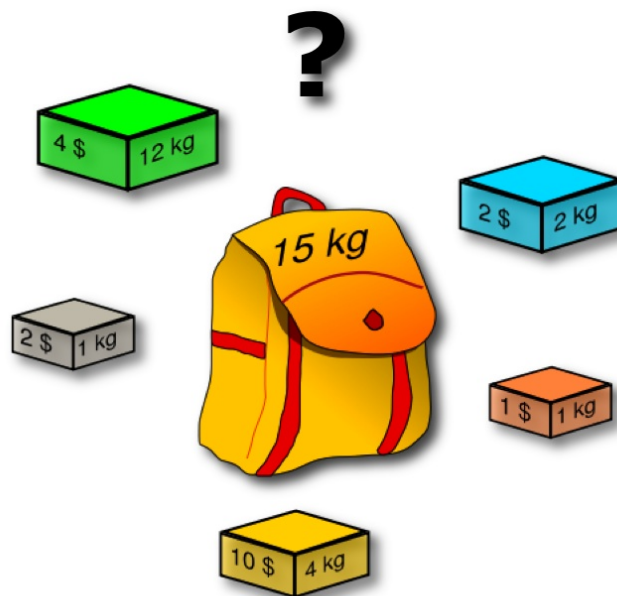


Figure 2.2: The knapsack problem.

In this case it is easier to derive a formal model than in the case of the eight queens problem, in fact it comes straight forward that the features of the model are:

Variables : x_1, \dots, x_n , where:

$$x_i = \begin{cases} 1 & \text{if item } i \text{ is inserted in the knapsack} \\ 0 & \text{otherwise} \end{cases}$$

For each item i we define also two constants:

$$w_i := \text{the cost (or weight) of the item } i$$

$p_i :=$ the value (or profit) of the item i

And one more constant for the knapsack:

$c :=$ knapsack's capacity (or cost)

In the following description of the model we will assume that:

$$w_i \leq c \tag{2.4}$$

so that each item can be (individually) inserted inside the knapsack, and also that:

$$\sum_{i=1}^n w_i \geq c \tag{2.5}$$

otherwise all the items could be inserted inside the knapsack and this would be the optimal solution.

Domain: the integer values $[0,1]$.

Constraints :

$$\sum_{i=1}^n w_i \cdot x_i \leq c \tag{2.6}$$

$$\max \sum_{i=1}^n p_i \cdot x_i \tag{2.7}$$

The first constraint says that the total cost of the chosen items has to be less than or equal to the knapsack's cost, and instead the second constraint states that the *total value* of the chosen items has to be as large as possible.

The last constraint that maximizes a function is also usually called "*Objective Function*" and it allows to find the optimal solution between the many

possible solutions to the problem.

The solution of the problem will be a set of values each assigned to a variable: each variable x_i will have a value 0 or 1, where if $x_i = 1$ the item i had been chosen to be inserted inside the knapsack. This solution will be also optimal, since the constraint 2.9 states that the total value has to be the highest.

It is to note that a problem similar to the “*knapsack problem*” often appears in business, cryptography and applied mathematics⁸.

2.3.4 The Multiple Knapsack Problem

The *Multiple Knapsack Problem*⁹ an NP-complete Constraint Satisfaction Problem, and so it is very hard to solve. It is substantially a generalization of the *Knapsack problem* explained in the above Section 2.3.3, indeed the problem is the same except for the fact that instead of only one knapsack there are many knapsacks, each with a different cost, where to insert the items. You can see a graphic representation of the problem in Figure 2.3.

Let’s analyze the formal CSP model in this case:

Variables:

$$x_{i,j} = \begin{cases} 1 & \text{if item } i \text{ is in knapsack } j \\ 0 & \text{otherwise} \end{cases}$$

$$\forall i \in \mathbf{N} = \{e_1, \dots, e_n\} \quad \text{Set of items}$$

$$\forall j \in \mathbf{M} = \{K_1, \dots, K_m\} \quad \text{Set of knapsacks}$$

⁸See http://en.wikipedia.org/wiki/Knapsack_problem , last checked on 09/05/07

⁹Also in this case we are actually considering a particular type of multiple knapsack problem called 0-1 multiple knapsack problem, since the items can be chosen ($x_{i,j} = 1$) or not chosen ($x_{i,j} = 0$) . Instead in the more general knapsack problem we can also decide to take more than only one instance of the same item i to insert inside the j th knapsack, so for example $x_{i,j} = 3$ means that we are taking three instances of the item i and inserting them inside the j th knapsack.

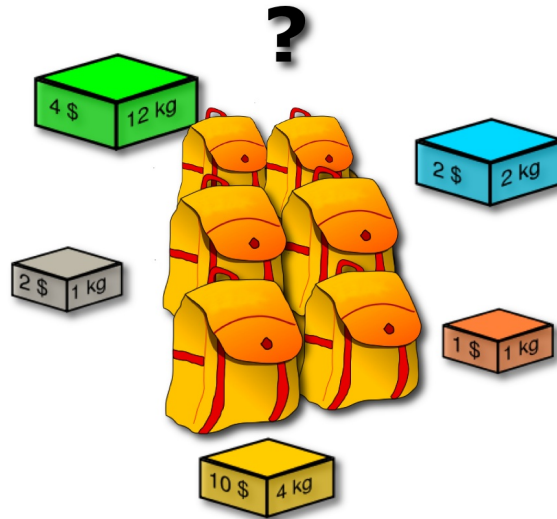


Figure 2.3: Multiple Knapsack Problem

For each item i we define two constants:

$w_i :=$ the cost (or weight) of the item i

$p_i :=$ the value (or profit) of the item i

And we also define a constant for each knapsack:

$c_j :=$ capacity (or cost) of knapsack j

In the following description of the model we will assume that:

$$\max_{i \in N} \{w_i\} \leq \max_{j \in M} \{c_j\} \quad (2.8)$$

that means that each item can be inserted in at least one knapsack, and

we assume also that:

$$\min_{j \in M} \{c_j\} \leq \min_{i \in N} \{w_i\} \quad (2.9)$$

that means that each knapsack can contain at least one item.

Domain: the integer values $[0,1]$.

Constraints:

- To ensures that the total cost of items inserted in the j th knapsack is less than the cost of the j th knapsack:

$$\sum_{i \in N} w_i \cdot x_{i,j} \leq c_j \quad \forall j \in M \quad (2.10)$$

- To ensures that each item will be inserted in no more than one knapsack:

$$\sum_{j \in M} x_{i,j} \leq 1 \quad \forall i \in N \quad (2.11)$$

- And finally we have the “*Objective Function*” that states that we have to maximize the total value of items inserted in each knapsack:

$$\max \sum_{i \in N} \sum_{j \in M} p_i \cdot x_{i,j} \quad (2.12)$$

Note that we could also adapt the “*Objective Function*” to our needs, by changing it. Furthermore as we will see in Section 3.3, this model will be the basis over which we will create our model for the problem of the “*Sensor Assignment*”. Indeed we will take this model and we will expand it with additional constant and other constraints.

Chapter 3

Concept and Design

This is the most important chapter of this document since it describes our solution to the problem, but it also gives an overview of the system and an idea of the first attempt that we did to try to solve the problem. In particular it is very interesting how we managed to learn from our mistakes, and how this brought us to a very good and innovative solution.

3.1 System Architecture

The architecture of the system is an architecture composed by three components, as you can see in figure 3.1.

The most important component is the solver, which is an application that solves the problem of finding an optimal deployment of the sensors inside the zones of a map given by the commander of a mission. This is implemented as a webservice and it represents actually the core of the system, since the remaining two components will interact each other using this webservice as computation and communication node. It is to note that as implementation of the solver we chose a webservice since in this way the other two components can be easily substituted by some different ones. So for example we could replace the commander's interface with a more user-friendly one and the game

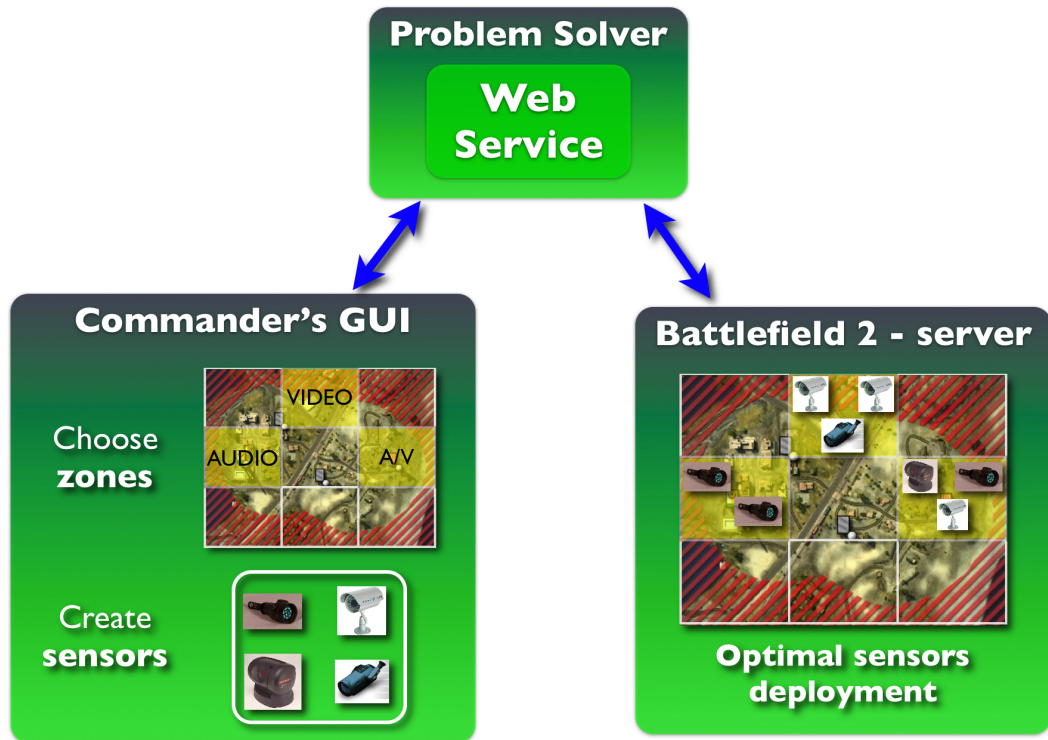


Figure 3.1: System Architecture

"*Battlefield 2*" with another virtual environment or also with the real world.

The second component is the commander's interface, which allows the commander to choose the zones from a map and to define which type of information it is required from each of them, and at the same time it allows the commander to insert data about the available sensors that is the number of sensors and a set of properties for each of them. The Commander's Interface will send to the solver the information about the sensors and about the zones selected, so later the solver will solve the problem. It is to note that the solution of the problem remains inside the solver as a static variable, so that at any time the webservice will contain the current optimal sensor deployment if there is one.

The last component is the *mod*¹ developed for *Battlefield 2* that actually modifies the behavior of the game server so that when some players connect to it to play a match, this will ask for the current optimal sensor deployment stored in the webservice, and then it creates the sensors inside the map. Anyway we will see in the Chapter 4 that the game is quite limited since the only way in which you can simulate sensors inside the game is creating a sensitive invisible area that let you know only what enters and what exits from it.

3.2 Modeling the Sensor Deployment as a CSP

Let's begin to describe the real challenge that we faced, that is to create a model for the “*Sensor Deployment*” problem. In fact the model for that problem will be implemented later as a webservice described in the previous Section 3.1 and it will actually constitute the core of the system.

In this Section we are going to present the first model that we created for the problem, and as we will notice later we will see that it is not the proper model for this problem. Anyway this model is described in the same way that we will use to present the right model since we want to describe each step that led us to develop the final model.

The first model that we developed takes inspiration by the eight queens problem, in particular it idealizes the map as a big chessboard where each possible position in the map corresponds to a cell in the chessboard. Furthermore between the constraints that we will use there will be one that is very similar to one of the constraint of the eight queens problem.

In this first model we worked out a simplified representation for sensors and zones, that actually could represent an ontology for sensor and zone, indeed this idealization will be reused in the correct model. So in particular we have:

Sensor: it is idealized as a circular area having: a radius (r_i), a center (x_i, y_i),

¹See Section 1.4

the type of information that the sensor can gather (e.g. AUDIO). You can see in the Figure 3.2 the schema of the representation of a sensor.

Zone: it is represented by: a set of four coordinates that determines the boundaries of a rectangular zone, and the type of information needed (e.g. AUDIO required). In Figure 3.2 there is the schema of this idealization of the zone.

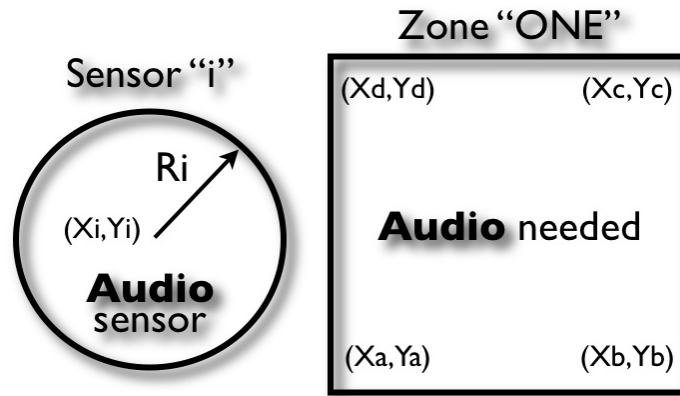


Figure 3.2: Representation of a sensor and a zone in the first model.

Now we can define the formal mathematical model that uses the idealization showed above:

Variables: $\{(x_1, y_1), \dots, (x_n, y_n)\}$

that is a list of (x_i, y_i) coordinates of the center of each sensor.

Domain: each (x, y) inside the boundary of the map

Constraints: The constraints can be divided into two types:

1. *System Constraints* that are actually the constraints proper of the hardware system of sensors, so:

- All the sensors have to be assigned to a different position in the map:

$$(x_i, y_i) \neq (x_j, y_j) \quad \forall i, j \in \{1, \dots, n\}, i \neq j \quad (3.1)$$

Note that this constraint is really similar to the constraint (2.1) used in the eight queens problem model in Section 2.1.

- The area of a sensor must not intersect the area covered by another sensor, and we implemented this constraint by using the fact that two circles intersect when the distance between their centers is less than the sum of their two radii:

$$\text{distance}((x_i, y_i), (x_j, y_j)) \geq |r_i + r_j| \quad \forall i, j \in \{1, \dots, n\}, i \neq j \quad (3.2)$$

2. *Commander's Selection Constraints* that are the constraints imposed by the commander when he selects zones in a map and he decides which type of information it is needed from these zones:

- First of all we divide the sensors in classes considering the type of information gathered by sensors, so we will have three different classes:

AUDIO sensors let's say that they are $i \in \{1, \dots, l\}$

VIDEO sensors let's say that they are $i \in \{l + 1, \dots, m\}$

A/V sensors let's say that they are $i \in \{m + 1, \dots, n\}$

- Now the following constraint states that there have to be only AUDIO sensors in the AUDIO zones selected by the commander. So *for each AUDIO zone* we have:

$$x_a \leq x_i \leq x_b \quad \forall i \in \{1, \dots, l\}, \quad (3.3)$$

$$y_a \leq y_i \leq y_d \quad \forall i \in \{1, \dots, l\}, \quad (3.4)$$

where the zone has coordinates $\{(x_a, y_a), (x_b, y_b), (x_c, y_c), (x_d, y_d)\}$ like in Figure 3.2.

- In the same way, *for each VIDEO zone* we have:

$$x_a \leq x_i \leq x_b \quad \forall i \in \{l, \dots, m\}, \quad (3.5)$$

$$y_a \leq y_i \leq y_d \quad \forall i \in \{l, \dots, m\}, \quad (3.6)$$

- And finally, for each A/V zone we have:

$$x_a \leq x_i \leq x_b \quad \forall i \in \{m, \dots, n\}, \quad (3.7)$$

$$y_a \leq y_i \leq y_d \quad \forall i \in \{m, \dots, n\}, \quad (3.8)$$

Note that the information about the capabilities of each sensor must be stored outside the mathematical model, in some data structure, viceversa you can see that the data about the type of information required from each zone is included in the model thanks to the subdivision of the zones into classes. In the last model that we will present in Section 3.3, we will see how we managed to include inside the model also the information about the capabilities of each sensor without using an external data structure.

You probably noticed also that we didn't used any "*Objective Function*" and this is why we wanted before to test how the model was working without having to optimize a solution, but only having to find a solution even not optimal. After an implementation phase and a test we found out that the solver which was using this model was of a very poor quality, since in many cases it was not able to find a (non optimal) solution in a reasonable time.

Furthermore we also realized that we were not considering the problem of doing a schedule of the capabilities of a sensor which had many capabilities. So for example we were assigning only VIDEO sensors to a VIDEO zone and we were not allowing the case in which an A/V sensor could be assigned to a VIDEO zone having only the VIDEO capability enabled.

To finish this description of the model we would like to analyze the reasons for what the model did not have good performance. The main cause is that the domain of this model is too big, in fact taking a look at the domain of the CSPs described in the previous Section 2, you will notice that those domains are really very small, sometimes they are composed by only two elements

(e.g. $\{0,1\}$). So it is comprehensible that our model does not work well since our domain includes each possible position inside a map, that is if we have a square map with a side of 512 meters, then we will have a domain with $512 \times 512 = 262144$ elements. Here we are considering that the unit of the map is one meter, and so the smallest distance between two sensors will be one meter. At the beginning we thought that reducing the domain by taking a unit inside the map that was bigger than one meter, so for example by taking 8 meters as unit in a map we would have reduced the domain of a factor 8^2 in fact the number of positions allowed in the map now becomes $64 \times 64 = 4096$. Although in this way the domain became smaller than before, it is still too big compared to the domains which are usually used in the classic CSPs, and in addition since we used a bigger unit for the map we lost precision, indeed now the smallest allowed distance between two sensors is 8 meters. Finally we decided that this was not the right model and we began to work on a new model.

3.3 Reformulation using the multiple knapsack problem

In this section we will describe the final version of the model developed for the “Sensor Assignment” problem. This model is actually the most important part of this project and in a matter of fact its development took also the most part of the time assigned to this project.

The intuition behind this model is to think at the “*Sensor Assignment*” problem as a “*Multiple Knapsack Problem*” described in Section 2.3.4, and to use a similar model for it. This new conception of the problem lead us to divide the main problem, described in Section 1.2, into two separate subproblems:

- The “*Sensor Assignment*” problem that is to assign sensors to zones considering only the zones selected by the commander and the type of

info required from them.

- The “*Sensor Deployment*” problem that is for each zone we deploy only sensors assigned to that particular zone.

This subdivision is very important since it allows also to introduce the scheduling of the capabilities of a sensor which has many capabilities. In fact, just after having found a solution to the “Sensor Assignment” problem, for a certain sensor with more than one capability we can choose which capabilities have to be enabled depending on the assigned zone. So for example if an A/V sensor is assigned to an AUDIO zone, then we will enable only the AUDIO resource of that A/V sensor.

In the following paragraphs we will describe more in detail the two different subproblems that will be solved in sequence one after the other and will give us the optimal sensor deployment.

3.3.1 Sensor Assignment

As we stated above this problem consists in the sensor assignment considering only the zones selected by the commander and the type of information that we need from these zones, this problem takes also into account the fact that sensors with many capabilities can be assigned to a zone that does not require that all the capabilities of sensor has to be enabled.

So a formal description of the problem, represented in Figure 3.3, could be: given a set of zones selected by the commander each with its own type of information required, and a set of sensors each with its own capabilities, then we will have to assign each sensor to a zone maximizing the total area covered by sensors. Note that at the end of this problem we will not have as a result the exact positions in which we have to deploy sensors, but we will get only the zone inside which each sensor has to be deployed.

You are probably also understanding that thanks to this new point of view on the problem definition we are actually reducing the domain with regards to the model presented in the Section 3.2. Indeed we are no more considering every possible position inside the map, instead we are now taking into account only the zones selected by the commander and not even including the positions that are not allowed in the definition of the problem . In Figure 3.3 we put some red crosses on the zones that are not selected by the commander to represent the domain reduction.

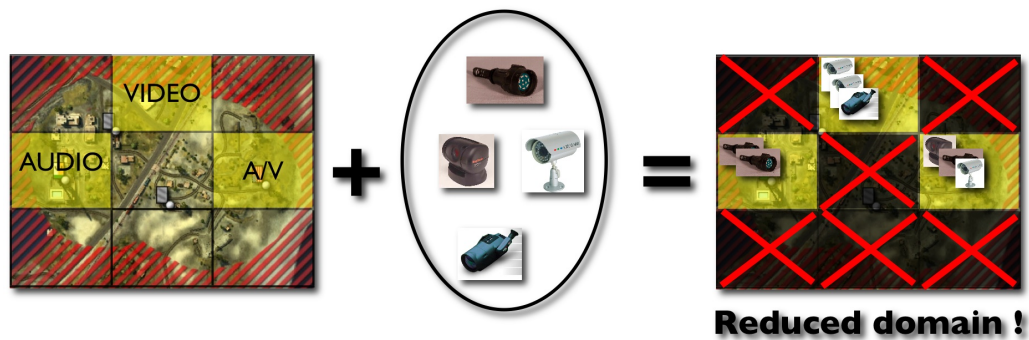


Figure 3.3: Representation of the Sensor Assignment Problem in the final model.

As we said above the model developed for this subproblem called “Sensor Assignment”, is substantially the model of the multiple knapsack extended with other constraints and other constants to include inside the model the information about the capabilities of the sensors and the data about the type of information required from each zone. To begin with let’s try to point out which are the *analogies*, represented in Figure 3.4, between the multiple knapsack problem and the ”Sensor Assignment” subproblem:

- Knapsacks \iff Zones selected,
- Knapsack’s cost \iff Area of a selected zone,
- Items \iff Sensors,
- Item’s cost \iff Area covered by the sensor,

- item's value \iff Area covered by the sensor

In particular note the last two analogies that say that in this case we are using a specific case of the multiple knapsack problem where $p_i = w_i$ for each sensor (or item), so since now we will use only the symbol w_i to indicate either the item's cost or the item's value.

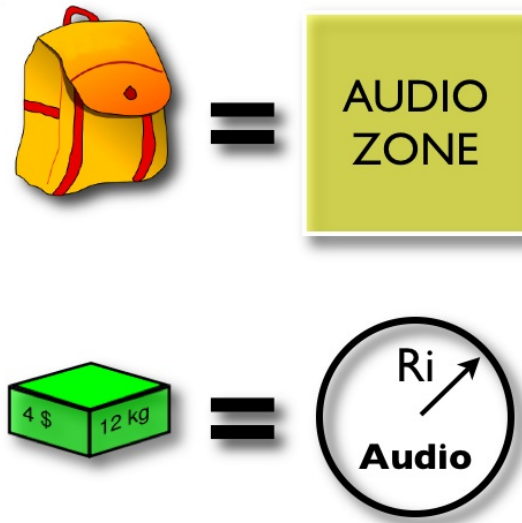


Figure 3.4: Analogies between the multiple knapsack and the Sensor Assignment problem.

Here we present the mathematical model for the problem:

Variables: We use a two-dimensional variable, to resolve the problem:

$$x_{i,j} = \begin{cases} 1 & \text{if sensor } i \text{ is in zone } j \\ 0 & \text{otherwise} \end{cases}$$

$$\forall i \in \mathbf{N} = \{s_1, \dots, s_n\} \quad \text{Set of sensors}$$

$$\forall j \in \mathbf{M} = \{Z_1, \dots, Z_m\} \quad \text{Set of zones}$$

And then we use some constant terms for each sensor and for each zone:

$$t_{a_i} = \begin{cases} 1 & \text{if sensor } i \text{ has AUDIO} \\ 0 & \text{otherwise} \end{cases}$$

$$t_{b_i} = \begin{cases} 1 & \text{if sensor } i \text{ has VIDEO} \\ 0 & \text{otherwise} \end{cases}$$

w_i = area covered by sensor i

c_j = area of the zone j

Below, we also subdivide the set of zones into subsets, each subset is composed by zones from which it is required the same type of information by the commander:

$\mathbf{M}_a = \{Z_1, \dots, Z_l\}$ Set of zones from which AUDIO is required.

$\mathbf{M}_b = \{Z_{l+1}, \dots, Z_f\}$ Set of zones from which VIDEO is required.

$\mathbf{M}_{a,b} = \{Z_{f+1}, \dots, Z_m\}$ Set of zones from which AUDIO and VIDEO are required.

Domain: the integer values $[0,1]$.

Constraints:

- The following constraints are the proper constraints of the multiple knapsack problem, like the constraints described in the Section 2.3.4:

$$\sum_{i \in N} w_i \cdot x_{i,j} \leq c_j \quad \forall j \in M \quad (3.9)$$

$$\sum_{j \in M} x_{i,j} \leq 1 \quad \forall i \in N \quad (3.10)$$

- The constraints below had been added to respect the commander's choices, in terms of type of information needed in each zone, this is one of the most important part of the model since it extends the basic model of the multiple knapsack into a more complex one:

The following constraint is to have only sensors with AUDIO enabled in AUDIO zones:

$$\sum_{i \in N} t_{a_i} \cdot x_{i,j} = \sum_{i \in N} x_{i,j} \quad \forall j \in M_a \quad (3.11)$$

The following constraint is to have only sensors with VIDEO enabled in VIDEO zones:

$$\sum_{i \in N} t_{b_i} \cdot x_{i,j} = \sum_{i \in N} x_{i,j} \quad \forall j \in M_b \quad (3.12)$$

The following constraint is to have only sensors with A/V enabled in A/V zones:

$$\sum_{i \in N} t_{a_i} \cdot t_{b_i} \cdot x_{i,j} = \sum_{i \in N} x_{i,j} \quad \forall j \in M_{a,b} \quad (3.13)$$

- And, as the last part of the extension of the multiple knapsack model, we add the following constraint to ensure that there is at least one sensor in each zone:

$$\sum_{i \in N} x_{i,j} \geq 1 \quad \forall j \in M \quad (3.14)$$

Objective function: In this case we preferred to treat the objective function in a separate paragraph, even if it is always a constraint. We developed two different objective function of which only one can be used inside the model, and we decided to use the second one.

- The first possibility for the objective function maximizes the total area covered by the sensors, considering the zones altogether:

$$\max \sum_{i \in N} \sum_{j \in M} w_i \cdot x_{i,j} \quad (3.15)$$

- Now the second possibility for the objective function, that is the function that we chose in the implementation, minimizes the number of sensors used and at the same time maximizes the total area covered by the sensors:

$$\max \sum_{i \in N} \sum_{j \in M} w_i \cdot x_{i,j} - \sum_{j \in M} \sum_{i \in N} x_{i,j} \quad (3.16)$$

In the future we could also change this objective function into a more complex one.

Note another time that in the context of this subproblem the coordinates of the boundaries of each zone do not matter, we will pay attention to them

only in the next subproblem called “Sensor Deployment”.

Finally, we would like to point out that this model is an innovative model since there is no evidence of other researches that applied an extension of the multiple knapsack model to the field of sensor assignment, so hopefully this report could be used as a basis for a future research paper about the CSP model carefully described above.

3.3.2 Sensor Deployment

This problem has to be solved separately for each zone, and it can be solved only after having found a solution for the subproblem called “Sensor Assignment”. A formal description for this second subproblem, represented in Figure 3.5, could be: given a zone and given the subset of sensor assigned to this zone, then we will have to deploy each sensor inside the zone with the constraint that the areas covered by any two sensors do not overlap.

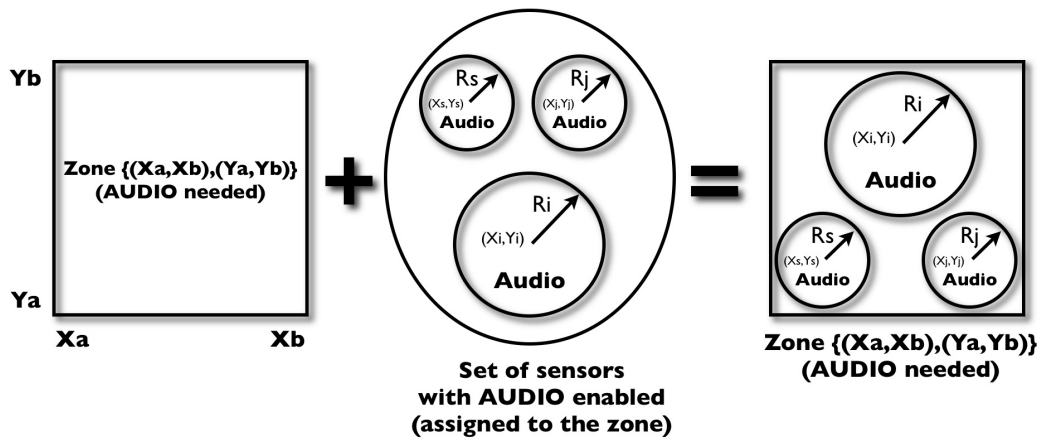


Figure 3.5: Representation of the Sensor Deployment Problem in the final model.

First of all we have to point out that this is not properly a CSP, but it is resolved by applying recursively the model of the multiple knapsack, so in other words we had to create an algorithm that is going to be described in details

into this Section. To understand what we mean with “applying recursively the multiple knapsack model” we will just explain the steps of the implemented algorithm.

Algorithm:

1. Subdivide the sensors in classes with the same radius.
2. Order the sensors for decreasing radius and take the first class (so the class with the biggest radius).
3. Subdivide the zone into subzones with side equals to the diameter of the sensors belonging to the class that we are now considering. In this way we will have that the area of a subzone is exactly equal to the area covered by the sensors belonging to the class that we are now considering (i.e. $c_j = w_i \quad \forall i \in N, \forall j \in M$)
4. Solve the multiple knapsack problem with:
 - Knapsacks \iff Subzones
 - Knapsack’s cost \iff Area of a subzone
 - Items \iff Sensors of the first class
 - Item’s cost \iff Area covered by sensor
 - Item’s value \iff Area covered by sensor

Note that the last two analogies with the multiple knapsack mean that also in this case we are using a particular case of it where $p_i = w_i$ for each sensor (or item).

5. Deploy sensors inside the subzones as states the solution of the multiple knapsack problem just resolved. In particular it is clear that all the sensors of the class will be deployed and none of them will be left out, since the set of sensors on which we are working is the set of sensors assigned to this zone from the “Sensor Assignment” model (which will

check that the total area covered by sensors is less than or equal to the area of the zone).

6. Start from the beginning of the algorithm considering the next class of sensor and always the same zone, but this time the zone will be subdivided into smaller subzones having side equals to the diameter of the next class of sensors (which have a smaller radius than the previous class). Moreover we will have to exclude the subzones that are already covered by each of the bigger sensors deployed during the previous cycle.

Let's see an example that is also represented in Figure 3.6, so if we have two classes of sensors then the algorithm will cycle twice: the first time it will assign the sensors that belongs to the class with the biggest radius to the subzones having as side the diameter of these sensors, the second time it will exclude the subzones already occupied by the sensors of the first class, and then it will assign the sensor to the second class to the new smaller subzones.

3.4 Modeling considerations

As you can easily understand the first subproblem called "Sensor Assignment" is the most hard to solve and also the most important. Indeed only if we have a solution for the first subproblem, that is a "global problem" since involves the whole map, we can go on to solve the second subproblem that is more a "local problem", since it involves only one zone and a subset of sensors per time.

3.4.1 Heuristic

Since both the subproblems are quite hard to solve we used an heuristic, which was anyway necessary so that the "Sensor Assignment" model described in 3.3.1 could work well.

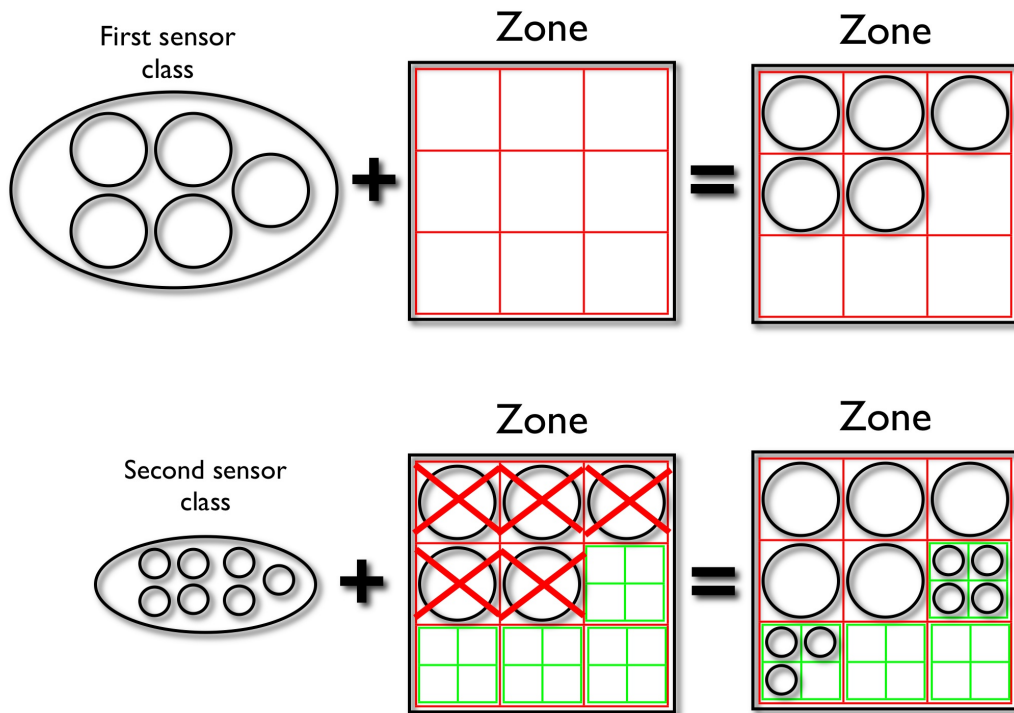


Figure 3.6: An example of application for the Sensor Deployment Algorithm.

Heuristic:

“The length of the side of each zone and the length of the radius of each sensor have to be a power of two.”

This is because otherwise there could be the case in which the “Sensor Assignment” model could insert a sensor in a zone putting it out of shape. Let’s see an example represented also in Figure 3.7, here we suppose that we are trying to insert the last sensor, with area equal to 7, inside a zone where there are already some sensors deployed and where the remaining area is greater than or equal to 7. Then if we do not use the heuristic we will have that the constraint 3.9 is respected and all the other constraints would be respected as well, and the solver will solve the problem assigning this sensor to that zone, even if it cannot be inserted, since we cannot change the shape of the area which it can cover. Instead using this heuristic will not allow to have such a situation.

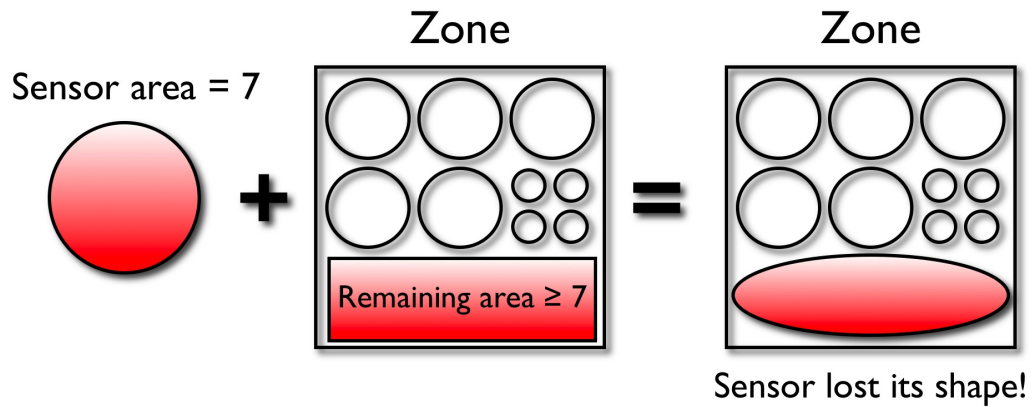


Figure 3.7: The “Sensor Assignment” model without the heuristic.

We apply this heuristic inside the system by using the power of two that is the nearest to the length of the zone side or the length of the sensor radius inserted by the commander using the commander interface.

3.4.2 Flexibilities of the “Sensor Assignment” model

The “Sensor Assignment” model is very flexible, meaning that it can adapt well to many different situations.

The first flexibility that we would like to point out is the fact that it is really easy to add to the “Sensor Assignment” model many different type of information. For example if you want to use also sensors that can have the capability INFRARED, you just need to add some constraint and some constant to the model described in Section 3.3.1.

So let’s consider the case of “INFRARED”, then we will define this constant for each sensor:

$$t_{c_i} = \begin{cases} 1 & \text{if sensor } i \text{ has INFRARED} \\ 0 & \text{otherwise} \end{cases}$$

Now, we will use these convention: A = AUDIO, V = VIDEO, I = INFRARED; so for example an AUDIO and INFRARED sensor or zone will be denoted as A/I, instead an AUDIO, VIDEO and INFRARED sensor or zone will be denoted as A/V/I.

So the types of zones that we can have now become the following:

$\mathbf{M}_a = \{Z_1, \dots, Z_e\}$	Set of zones from which “A” is requested.
$\mathbf{M}_b = \{Z_{e+1}, \dots, Z_f\}$	Set of zones from which “V” is requested.
$\mathbf{M}_c = \{Z_{f+1}, \dots, Z_g\}$	Set of zones from which “I” is requested.
$\mathbf{M}_{a,b} = \{Z_{f+1}, \dots, Z_g\}$	Set of zones from which “A/V” are requested.
$\mathbf{M}_{a,c} = \{Z_{g+1}, \dots, Z_h\}$	Set of zones from which “A/I” are requested.
$\mathbf{M}_{b,c} = \{Z_{h+1}, \dots, Z_l\}$	Set of zones from which “V/I” are requested.
$\mathbf{M}_{a,b,c} = \{Z_{l+1}, \dots, Z_m\}$	Set of zones from which “A/V/I” are requested.

The constraints that we will have to add will be:

$$\sum_{i \in N} t_{c_i} \cdot x_{i,j} = \sum_{i \in N} x_{i,j} \quad \forall j \in M_c \quad (3.17)$$

$$\sum_{i \in N} t_{a_i} \cdot t_{c_i} \cdot x_{i,j} = \sum_{i \in N} x_{i,j} \quad \forall j \in M_{a,c} \quad (3.18)$$

$$\sum_{i \in N} t_{b_i} \cdot t_{c_i} \cdot x_{i,j} = \sum_{i \in N} x_{i,j} \quad \forall j \in M_{b,c} \quad (3.19)$$

$$\sum_{i \in N} t_{a_i} \cdot t_{b_i} \cdot t_{c_i} \cdot x_{i,j} = \sum_{i \in N} x_{i,j} \quad \forall j \in M_{a,b,c} \quad (3.20)$$

From this we can notice that it is very easy to add another capability to the model, and this could be done also in an automatic way. This means that

in the future we could implement a function to allow the commander to create new capabilities for the sensors and new information requirements for the zones; and in this way the model could be adapted on the fly to the needs of the commander.

Another consideration is that the “Sensor Assignment” model could be also applied to sensors that can move, in fact we could integrate this new information by taking as radius of area covered by the sensor, the radius of the actual area in which the sensor can move. So we will simply use a bigger radius that takes into account also the case in which the sensor can move from its position. Since the sensor can move, it will have a different position after a while and so it should be necessary to resolve again the problem of deploying sensors in an optimal way. This could be a future expansion of this project.

Chapter 4

Implementation and Testing

This chapter describes the implementation of this system by looking at the technologies used and at the implementation of the three components of the system. Later we also explain which tests were done and which are the performances of our solution.

4.1 Implementation

4.1.1 Technologies Used

Here there is a brief description of the technologies used to implement this project, anyway keep always in mind the system architecture described in Section 3.1 so you could easily understand where these components are located inside the system:

Java 1.6 Since the commander interface and the webservice are written in java they require the Java VM installed on the machines in which they will run.

Apache Tomcat 6.0.2 This is an application server that allows application written in Java to be executed on the server by a client. This application server represent the platform on which we will install Axis that we need to create the webservice.

Axis 1.4 This is a platform for developing and deploying webservices written in Java. It is itself a web application that has to be installed inside Tomcat.

choco-1.2.03 This is an open source Java library that is used by the webservice to solve the problem of deploying the sensors inside the zones in an optimal way. This type of library let you apply the Constraint Programming paradigm, by defining variables, domain, constraints and objective function.

4.1.2 Webservice Implementation

As we stated above we used Axis to create the Webservice, we chose it because it is very easy to deploy new webservices. As a matter of fact you just only need to write your own Java code and insert your packages inside Axis, then you need to create two files of configuration to let Axis know that it has to deploy a new webservice and you're done. Furthermore the installation of Axis which runs inside the Apache Tomcat application server is quite easy to accomplish as well in a limited amount of time.

Let's now take an overview of the Java classes that implement the Webservice. This is composed by one package "deploySensorsService" which contains a file "MyService.java" which actually implements the webservice, and a subpackage "deploySensorsService.solver" that contains all the classes which implement the CSP solver using CHOCO library.

- Let's consider the package "deploySensorsService":

MyService.java This class implements the webservice: the method "computeDeployment" performs the deployment of the sensors inside the zones, the other methods are used to return to the client the actual sensor deployment.

The thing that is very important is that once the CSP solver has solved the problem, then the solution will be stored inside the webservice. So when the BF2 server will ask for the current optimal deployment, the webservice will return the value of the “static” variables which contains the data about the optimal deployment.

- Let’s consider the subpackage ”`deploySensorsService.solver`”, where the class that actually solve the problem is ”`DeploySensors.java`” which uses ”`ZoneDeploy.java`” as an auxiliary class. The other classes are data structures and auxiliary methods used by these two main classes:

DeploySensors.java This class solves in sequence, before the *Sensor Assignment* problem (Section 3.3.1) and then the *Sensor Deployment* problem (Section 3.3.2). This last subproblem is solved using the class ”`ZoneDeploy`”.

This class reflects the global structure of the main problem that is divided into two subproblems: the Sensor Assignment and then the Sensor Deployment problem. The implementation of the model, since it is directly the translation of the model into CHOCO constraints, also benefits of the flexibilities described in Section 3.4.2.

ZoneDeploy.java This class solve the Sensor Deployment problem for each zone considering only the sensors assigned to the zone; this problem is solved by applying the algorithm described in Section 3.3.2.

Sensor.java This is a data structure that represents the sensor and its properties. It could be considered as a primitive ontology for the object sensor.

CoveredArea.java This is a data structure that represents the zone selected by the commander and the information that is required from it. It could be considered as a primitive ontology for the object Zone.

SubArea.java This class represents a subzone created by division of a zone, this class is used in the algorithm that solves the Sensor Deployment problem.

MyList.java This class implements a list using an hashtable and it is used as a utility class by the others.

PairInt.java This class implements an object composed by a pair of integers

Utilities.java This class contains some utility function used by the classes "DeploySensors" and "ZoneDeploy".

4.1.3 Commander's Interface Implementation

As we said before the commander's interface is actually a command line interface written in Java, and it sends parameters to the webservice via SOAP messages using some Java libraries provided by Axis. The commander interface is composed by one package "deploySensorsClient" which contains a file "MyClient.java" and a subpackage "deploySensorsClient.structures". The first is the main class of the application and it implements the command-line interface, the second contains all the data structures used by the interface to perform its tasks (i.e. the same data structures used by the Webservice solver such as Sensor.java).

- The package "deploySensorsClient" contains:

MyClient.java This class implements the Commander's interface: it asks to the webservice for the solution of the problem whose parameters are set by the commander. This class uses the classes in "deploySensorClient.structures" to set the input parameters (sensors and zones) of the method "computeDeployment" of the webservice.

Note that inside this class it is implemented the heuristic that we described in the Section 3.4.2, since when the commander inserts a length for the zone side or for a sensor radius, the interface automatically computes the power of two nearest to the number inserted by the commander and then the system takes this power of two to solve the problem.

- The subpackage “deploySensorsClient.structures” contains data structures used by the commander interface to set the parameters of the problem. These classes are the same data structures used by the Webservice solver: `Sensor.java`, `CoveredArea.java`, `SubArea.java`, `MyList.java`.

4.1.4 “Battlefield 2 Mod” Implementation

As explained in Section 1.4, Battlefield 2 allows to develop your own plug-ins for the server, these plug-ins are called “mod” and they are written in Python. The real core of the Mod is implemented inside the file “scoringCommon.py” that is entirely written by Diego Pizzocaro. We used also other utility modules that are “Utils.py” and “Defines.py” which were taken from the mod “PlanAndPlay”.

scoringCommon.py This file is the core of the mod: It asks to the webservice for the current sensor deployment (that had been set before by the commander) and then it creates Sensitive Area inside the map simulating the behaviour of real sensors.

The issue is that once we created sensors inside the map, we cannot delete them on the fly while there are still other players inside the map since the game clients ask for the sensitive area that they have to create inside the local map only when they join the game. So we had to work out a mechanism that could delete old sensors and deploy the new sensors belonging to the solution of the new problem set by the commander:

Creating sensors: Each time that the first player join the server loading the map, the Battlefield 2 server will send a request to the webservice for the current optimal deployment, and then the Battlefield 2 server will create the sensors on the map

Removing sensors: When the last player disconnects from the current game, then the map and all the python code will be reloaded so that the old sensors created inside the map will be removed. So when the first player will connect to the server and join a game, there will be again the same sequence of actions described in **Creating sensors** and the sensors will be deployed on a clean map.

4.2 Testing and Evaluation

Since the most part of the time was spent to develop the model of the problem and then to implement the whole system, there was not so much time left to perform tests and evaluate its performances.

Anyway in the commander interface we implemented some commands that allow to test the solver performances with some default parameters (i.e. a fixed number and type of sensors and zones). The most interesting command is “default3” that actually uses as parameters 15 sensors of which 5 AUDIO sensors, 5 VIDEO sensors and 5 A/V sensors with different radii, and also 6 zones with different types of information needed. It takes more or less 30 seconds to the solver to solve this problem.

Note that the time spent to solve a problem will not depend from the size of the map thanks to the model, since we are only considering how many zone the commander selects. So the time to solve a problem increases with the number of sensors used and the number of zones selected. Other tests, carried on still during the development phase, showed that the time also depends on the side of the zones and on the radius of the sensors. In particular it is directly pro-

portional to the side of zones (the bigger is the side the bigger is the time), and inversely proportional to the radius of sensors (the bigger is the radius the smaller is the time).

An important issue rises when there is no solution to the problem and the solver has to understand that there is no solution. If the problem is “easy” the time to wait will be very short and it will answer that there is no solution, but if there are many sensors and zones, with also small radius the first and big side the second, then it will take a lot of time, and it could also go on indefinitely trying to find a solution. This is understandable since we are coping with an NP-complete problem.

Chapter 5

Conclusion and Future works

This chapter presents some conclusions to our project, discussing which were our expectations at the beginning of the project and how things actually went. We also present possible extensions and improvements to our system as future work.

5.1 Conclusion

At the beginning of this project we were not really concerned about the development of a model but we were more worried about the implementation of everything, in particular about how we could represent sensors in Battlefield 2 and how to integrate everything inside this virtual environment. At the end, instead, it turned out that the hardest part of the project was to develop a very efficient model for the problem that we were facing, and after this step the implementation was quite straightforward, at least for the solver inside the webservice. In any case we had to spend much time over the webservice design and the commander's interface, instead with regards to Battlefield 2 Mod server, we have been very lucky since the parallel project "Plan and Play" introduced in Section 2.2 had worked a lot with Battlefield 2 and so we could re-use some of the knowledge developed in that project.

In conclusion the goal of the project was successfully met and the CSP model that we developed for the problem seems to be a real innovation in this field.

5.2 Future works

There are many future works that could be done starting from this project, but always keeping as basis the same model described in this document without changing it very much, since as we showed it is very flexible.

5.2.1 Improving solver: Relaxed Constraints

As we discussed in Section 4.2 the solver could spend a lot of time to find the solution for a problem with many sensors and zones, or it could go on indefinitely trying to understand if it exists a solution. To improve the solver performances, we could apply a technique that keep the same model but relaxes the constraints defined in the problem. This could be really an interesting future work.

5.2.2 Objective function improvement

In the future we could also try to develop a better objective function, by considering for each item $i : p_i \neq w_i$, that are constants that we met in Section 3.3.1, and so we could maximize the total value of the sensors assigned to the zones. In this way a new objective function could maximize the total area covered by sensors, minimize the number of sensors used and at the same time maximize the value of the sensors chosen.

We could also create many different objective functions and then let the decision of which to use to the commander, depending on the requirements of the particular mission.

5.2.3 Dealing with multiple mission

An interesting feature of this model is that it can already deal with multiple missions. In fact, for example if there are two different commanders for two different missions, then they will have the same map and the same set of sensors available to share between the two missions, but they will mainly select different zones. It is enough to make a join of the two set of zones and to pass them together with the set of common sensor to the problem solver, and it will solve the problem. Actually there could be the need to adjust some little constraint of the model and some constants but it should be really straightforward.

Such a type of future work could be very interesting with regards to the ITA project.

5.2.4 Integration with “Plan And Play”

This project can be integrated with “Plan and Play” (PnP), described in Section 2.2. In fact PnP, as we stated previously, integrates a planner with Battlefield 2, and this planner receive in input a plan. Such a plan could be to realize the optimal deployment of the sensors by sending soldiers to place sensors in the optimal locations chosen by the solver. So in this case the planner will have as objective the optimal deployment and it will say to the soldiers how to reach safely the optimal locations to place sensors.

Bibliography

- [1] US Army,
“*Doctrine for Intelligence Support to Joint Operations*”.
http://www.dtic.mil/doctrine/jel/new_pubs/jp2-0.pdf,
Checked on 06/05/2007.

- [2] Diane J. Cook, Piotr Gmytrasiewicz, and Lawrence B. Holder, Member,
IEEE.
“*Decision-Theoretic Cooperative Sensor Planning*”.
IEEE Transactions on pattern analysis and machine intelligence, Vol. 18,
NO. 10, October 1996.

- [3] Goce Trajceviski, Peter Scheuermann, Herve Bronnimann
“*Mission-Critical Management of Mobile Sensors (or, How to Guide a
Flock of Sensors)*”

- [4] Hsing-Jung Huang, Ting-Hao Chang, Shu-Yu Hu, Polly Huang
“*Magnetic Diffusion: Disseminating Mission-Critical Data for Dynamic
Sensor Networks*”.

- [5] Daniele Masato.
“*Plan and Play: Interfacing an HTN Planner with a virtual environ-
ment*”.
Honours Project Report,
University of Aberdeen, UK, May 2007.

-
- [6] Krzysztof R. Apt.
“Principles of Constraint Programming”.
Cambridge University Press, 2003.
- [7] Michele Monaci.
“Modelli di Programmazione Lineare Intera”.
Notes about the course of “Operational Research 2”,
http://www.dei.unipd.it/~monaci/modelli_rev21.pdf
University of Padova, 2006.
- [8] Sami Khuri, Thomas Back, Jorg Heitkotter.
“The Zero/One Multiple Knapsack Problem and Genetic Algorithms (1993)”
Proc. of the 1994 ACM Symposium of Applied Computation proceedings.

Appendix A

User Manual

This manual documents how to run our system – it should be helpful to a commander who wants to know which are the best positions where to place sensors, given a set of available sensors and of selected zones of a map.

Note: This manual assumes that you have already installed the system correctly, if you didn't please see the Maintenance Manual and follow the Installation Instructions.

A.1 Starting the system

There are three main components which require to be executed at the same time - the webservice, the Battlefield 2 mod server and the Commander's Interface.

The webservice can be executed by simply starting the Apache tomcat service, that under WindowsXP can be done by selecting "Start→Configure Tomcat" and then by pushing the "Start" button. This will automatically start also the Axis Platform that is located inside Apache Tomcat (after a proper installation of the system), and so it will also start the webservice called "DeploySensorsService".

The Battlefield 2 mod server can be executed by browsing to the folder

"Battlefield2/ServerConfig" and then executing the file "bf2server.exe". Note that before executing the server, it has to be configured with the correct IP, as specified in the README file inside the installation folder "Software/Battlefield 2 Mod".

The Commander Interface can be simply executed by browsing into its folder and then by clicking on the file "ComInterface.bat" if you are in a WindowsXP environment. If you are not under WindowsXP you have to open a command prompt, browse to the folder "CommanderInterface" and then type:

```
java -jar ComInterface.jar
```

Finally all the BF2 players that wants to use this infrastructure can start their game client modified with the Mod that is supposed to be correctly installed in their machine. To start the game mod installed in a client machine you just need to browse to the folder of the game "Battlefield2" and then click on "RunBF2Client Debug.bat" or on "RunBF2Client Release.bat", depending if you want to run the game in debug¹ or in release². Note that before allowing the first player to connect to the BF2 server you should be sure to have stored the deployment of the sensors through the Commander Interface.

A.2 Using the system

Let's take a look at the typical sequence of operations in such a system:

1. Start the webservice, the BF2 mod server and the commander interface.
2. Use the commander interface to create sensors and select zones and then wait for the answer of the webservice, that will solve the problem (if there is a solution). At the same time, the Webservice will store the sensor deployment into a static variable.

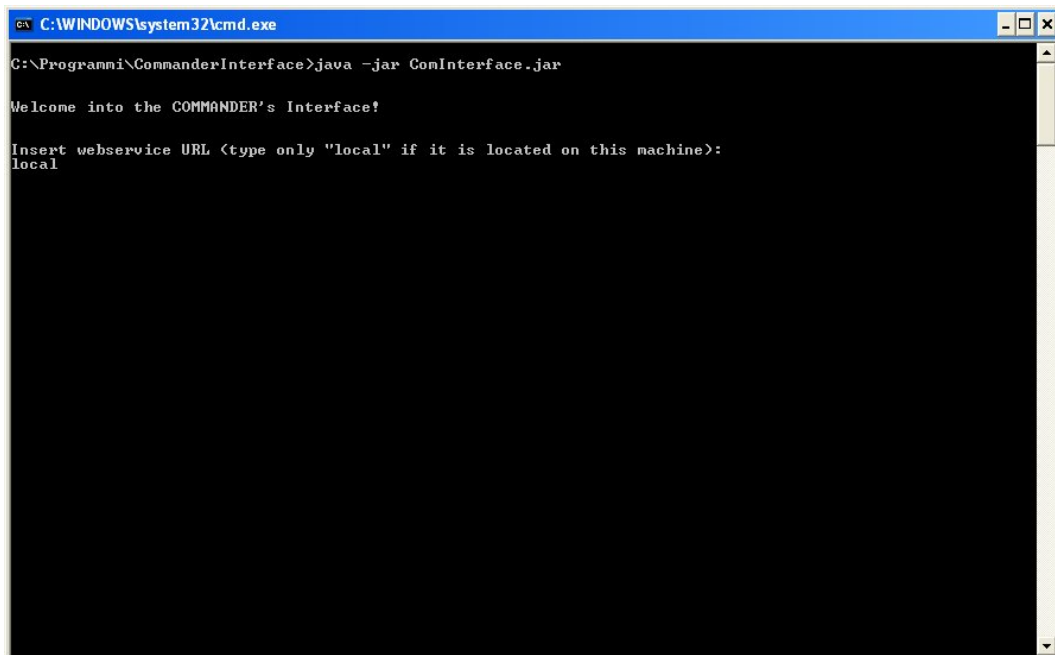
¹that means in a window at a lower resolution, with the possibility to see debug/error messages

²that means at higher resolution on full screen

3. The players can now start their client of the game with the "Mod" installed and then connect to the server "PlanAndPlay Test Server"
4. When the first player will connect to the BF2 server, this will ask to the webservice for the stored sensor deployment (previously set by the commander interface).

A.2.1 Using the Commander's Interface

When first executing the commander's interface you will be prompted to input the URL of the webservice. If it is in the same machine where you are running the Commander's Interface, then you will have only to write "local", as you can see in Figure A.1.



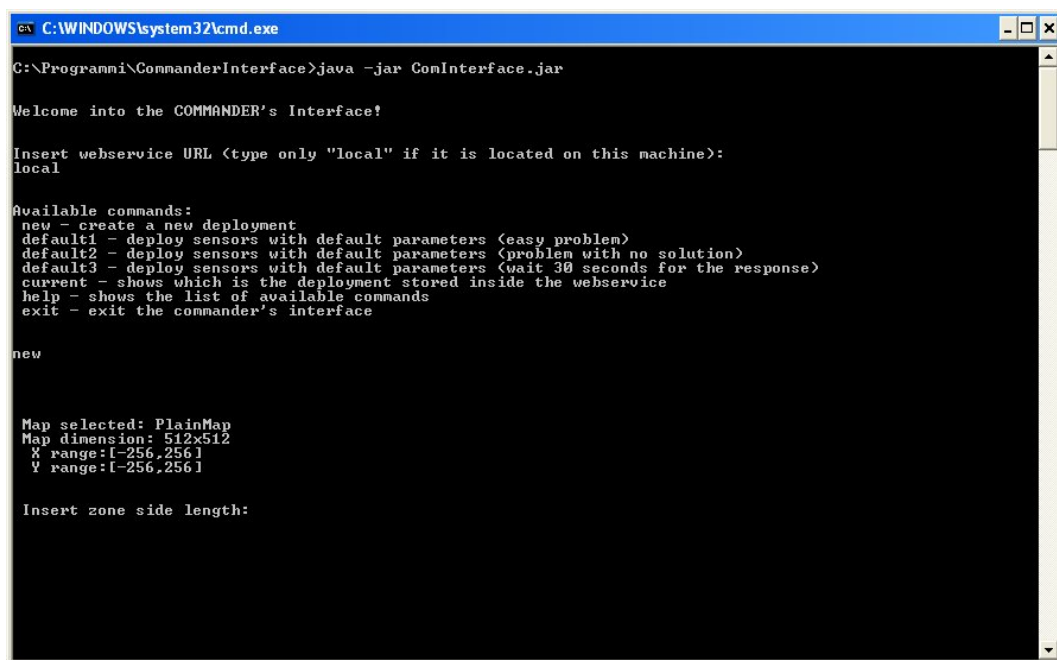
```
C:\WINDOWS\system32\cmd.exe
C:\Programmi\ComanderInterface>java -jar ComInterface.jar
Welcome into the COMMANDER's Interface!
Insert webservice URL <type only "local" if it is located on this machine>:
local
```

Figure A.1: Locating webservice.

Then you will see a list of commands with a little description of what they will do. The most important between these commands is "new", which actually

allows the commander to set the parameters such as sensors and zones and then to send this parameters to the webservice that will solve the problem and will store the solution. Let's analyze each step of the "new" command:

1. You can see in Figure A.2, that the deployment will be done on the map "PlainMap" and it says also which is the extension of the map. It asks to insert the side of zone that you want to use.



```
C:\WINDOWS\system32\cmd.exe
C:\Programmi\CommanderInterface>java -jar ComInterface.jar

Welcome into the COMMANDER's Interface!

Insert webservice URL <type only "local" if it is located on this machine>:
local

Available commands:
new - create a new deployment
default1 - deploy sensors with default parameters <easy problem>
default2 - deploy sensors with default parameters <problem with no solution>
default3 - deploy sensors with default parameters <wait 30 seconds for the response>
current - shows which is the deployment stored inside the webservice
help - shows the list of available commands
exit - exit the commander's interface

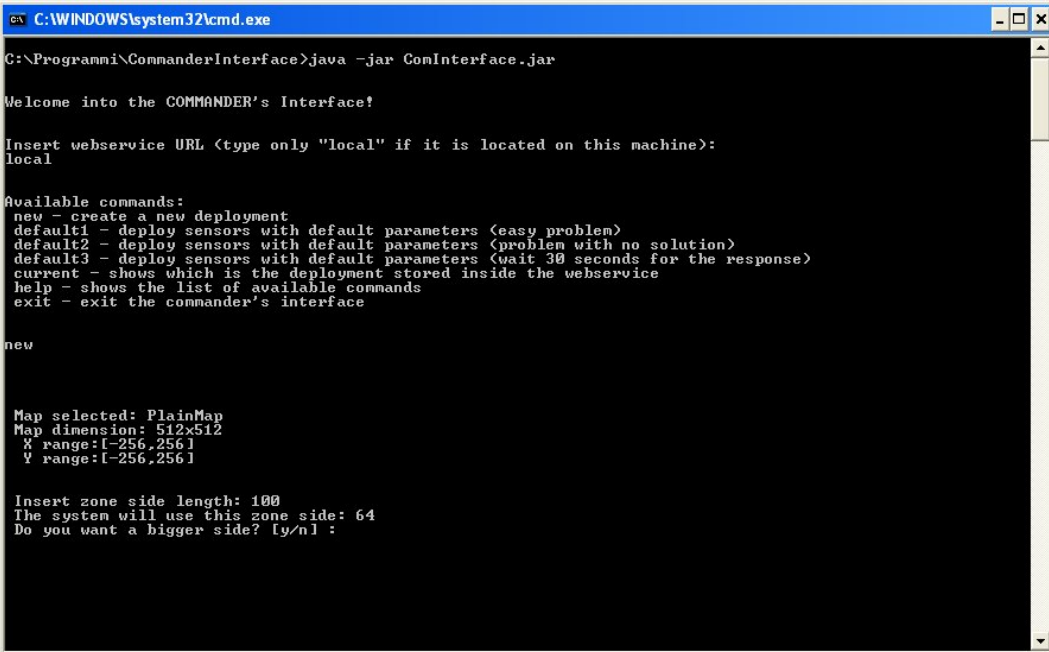
new

Map selected: PlainMap
Map dimension: 512x512
X range: [-256,256]
Y range: [-256,256]

Insert zone side length:
```

Figure A.2: Inserting zone side.

2. Once you inserted the side it will write, as in Figure A.3, that the system will use a different number since the system will actually solve a simplified version of the problem that you are setting up.
3. Now it asks you how many zones you want to select, as shown in Figure A.4 and it goes through the details of each zone that you will select, like in Figure A.5.
4. Then it asks how many sensors are available and it lets you define all the parameters for each sensor, as shown in Figure A.6.



```
C:\WINDOWS\system32\cmd.exe
C:\Programmi\ComanderInterface>java -jar ComInterface.jar

Welcome into the COMMANDER's Interface!

Insert webservice URL <type only "local" if it is located on this machine>:
local

Available commands:
new - create a new deployment
default1 - deploy sensors with default parameters <easy problem>
default2 - deploy sensors with default parameters <problem with no solution>
default3 - deploy sensors with default parameters <wait 30 seconds for the response>
current - shows which is the deployment stored inside the webservice
help - shows the list of available commands
exit - exit the commander's interface

new

Map selected: PlainMap
Map dimension: 512x512
X range: [-256, 256]
Y range: [-256, 256]

Insert zone side length: 100
The system will use this zone side: 64
Do you want a bigger side? [y/n] :
```

Figure A.3: The side used by the system.

5. Finally it sends the parameters to the webservice and it waits for the solution, that will be in the format shown in Figure A.7.

Finally the players can now connect to the Battlefield 2 “Plan And Play Test Server” and there will have invisible sensors deployed inside the map. The player knows when he is accessing or exiting a certain sensor area since on the screen of the player inside the game it will appear a message, as shown in Figure A.8.


```
C:\WINDOWS\system32\cmd.exe
Insert zone side length: 100
The system will use this zone side: 64
Do you want a bigger side? [y/n] : n

----- ZONE IDs ON THE MAP -----
  0   1   2   3   4   5   6   7
  8   9  10  11  12  13  14  15
 16  17  18  19  20  21  22  23
 24  25  26  27  28  29  30  31
 32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47
 48  49  50  51  52  53  54  55
 56  57  58  59  60  61  62  63

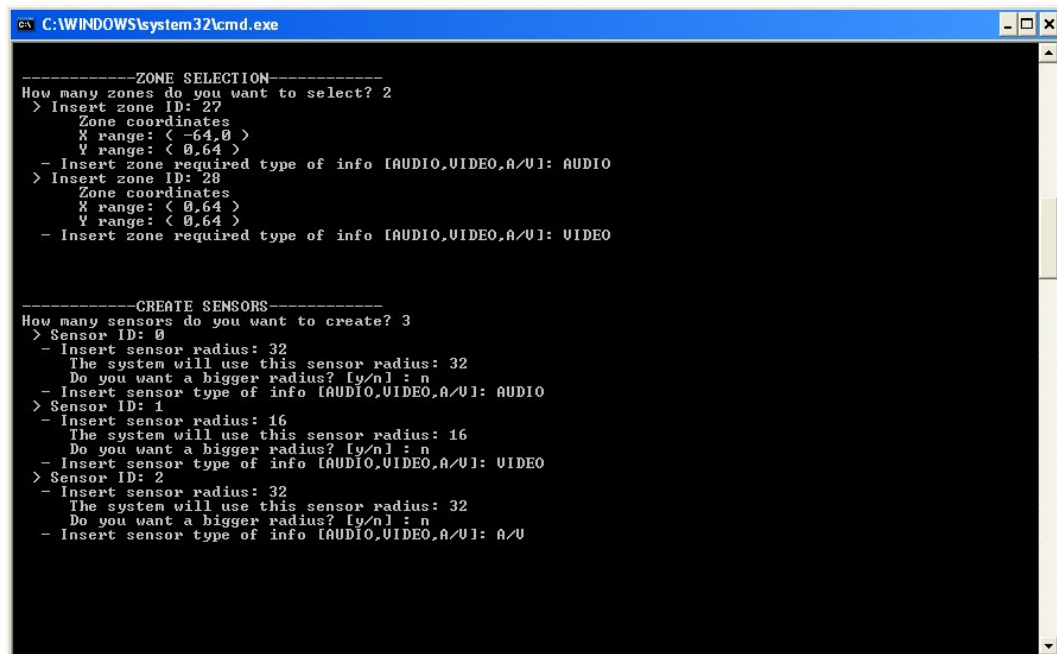
-----ZONE SELECTION-----
How many zones do you want to select?
```

Figure A.4: Selecting zones.

```
C:\WINDOWS\system32\cmd.exe

-----ZONE SELECTION-----
How many zones do you want to select? 2
> Insert zone ID: 27
  Zone coordinates
  X range: < -64,0 >
  Y range: < 0,64 >
- Insert zone required type of info [AUDIO,VIDEO,A/U]: AUDIO
> Insert zone ID: 28
  Zone coordinates
  X range: < 0,64 >
  Y range: < 0,64 >
- Insert zone required type of info [AUDIO,VIDEO,A/U]: VIDEO
```

Figure A.5: Creating zones.

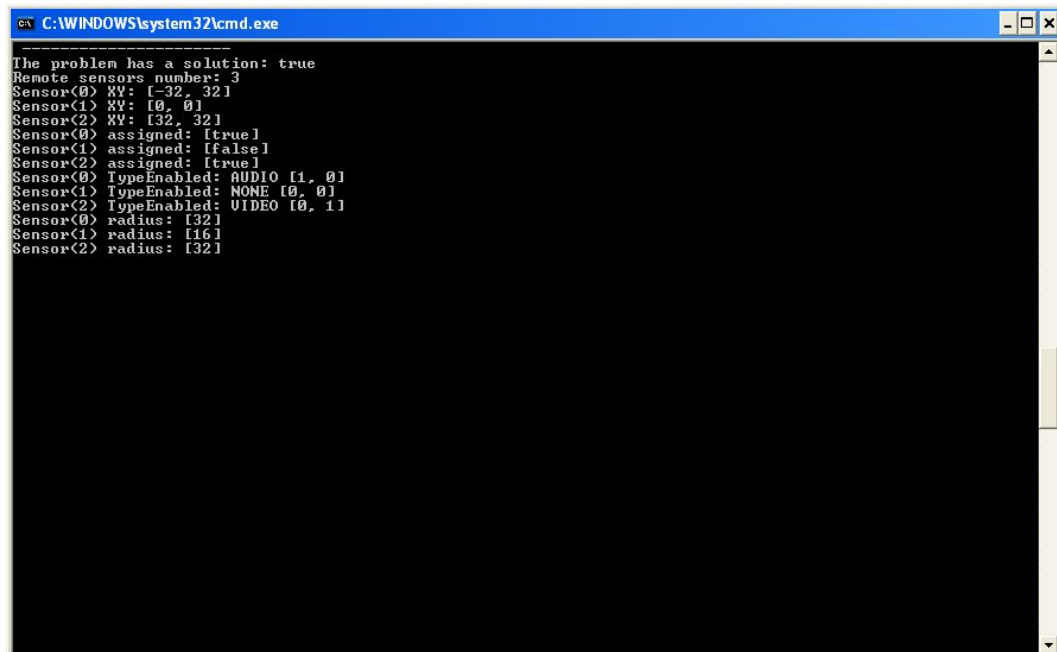


```
C:\WINDOWS\system32\cmd.exe

-----ZONE SELECTION-----
How many zones do you want to select? 2
> Insert zone ID: 27
  Zone coordinates
  X range: < -64,0 >
  Y range: < 0,64 >
- Insert zone required type of info [AUDIO,VIDEO,A/V]: AUDIO
> Insert zone ID: 28
  Zone coordinates
  X range: < 0,64 >
  Y range: < 0,64 >
- Insert zone required type of info [AUDIO,VIDEO,A/V]: VIDEO

-----CREATE SENSORS-----
How many sensors do you want to create? 3
> Sensor ID: 0
- Insert sensor radius: 32
  The system will use this sensor radius: 32
  Do you want a bigger radius? [y/n] : n
- Insert sensor type of info [AUDIO,VIDEO,A/V]: AUDIO
> Sensor ID: 1
- Insert sensor radius: 16
  The system will use this sensor radius: 16
  Do you want a bigger radius? [y/n] : n
- Insert sensor type of info [AUDIO,VIDEO,A/V]: VIDEO
> Sensor ID: 2
- Insert sensor radius: 32
  The system will use this sensor radius: 32
  Do you want a bigger radius? [y/n] : n
- Insert sensor type of info [AUDIO,VIDEO,A/V]: A/V
```

Figure A.6: Creating Sensors.



```
C:\WINDOWS\system32\cmd.exe

-----
The problem has a solution: true
Remote sensors number: 3
Sensor(0) XY: [-32, 32]
Sensor(1) XY: [0, 0]
Sensor(2) XY: [32, 32]
Sensor(0) assigned: [true]
Sensor(1) assigned: [false]
Sensor(2) assigned: [true]
Sensor(0) TypeEnabled: AUDIO [1, 0]
Sensor(1) TypeEnabled: NONE [0, 0]
Sensor(2) TypeEnabled: VIDEO [0, 1]
Sensor(0) radius: [32]
Sensor(1) radius: [16]
Sensor(2) radius: [32]
```

Figure A.7: Solution sent by the webservice.



Figure A.8: The message that appears entering a sensor area in BF2.

Appendix B

Maintenance Manual

This manual should be helpful to people who want to install the program, modify the program, extend the program, or be aware of which are the known bugs. In this document we give also a brief description of each of the source files.

B.1 Installation Instructions

The system is composed by three main component, that are actually three pice of software, and each one could be considered as an independent application. Let's analyze the folder structure in the CD that contains the software and the source code:

- **Software** - which contains everything you have to install to run the system
- **src** - which contains the source code

The folder **Software** contains three directories, which reflect the system architecture:

- **CommanderInterface** - which contains the Interface for the commander
- **WebService** - which contains the platform where you have to insert the webservice, and the webservice itself

- **Battlefield 2 Mod** - which contains the plug-in developed for BF2 and other needed game patches

Each of these folders contains a README.TXT file that explains how to install each component. The Commander's interface can be installed on every OS since it is written in java and it is a jar file. The webservice is also written in java so it could be installed on every OS, but the installer for Apache tomcat included in the CD is only for WindowsXP, so if you want to install the webservice in a OS that is not WindowsXP you will have to download the proper version of Apache Tomcat. The BF2 "mod" (i.e. modified) game server is exclusively compiled for WindowsXP, so it has to be installed in a WindowsXP environment.

To check if the webservice is well installed you could start the Apache Tomcat service, and then you could open a browser and write the following URL:

"<http://localhost:8080/axis>" and then click on the link "List" and verify that "DeploySensorsService" is in the list.

Note that it is better to install all the three components on the same machine since this is the default configuration. You can also decide to install the Commander Interface in a different PC and leave the webservice and the BF2 mod game server on another machine; in this case you will have only to enter the proper URL of the webservice in the commander interface when it will require the address of the webservice. You could also decide to install all the three components on three different machines, but to do this you will have to change the code inside the file "Battlefield2/mods/PlanAndPlay/scoringCommon.py" and in particular inside the function "OnPlayerConnect(player)" and substitute the correct URL of the webservice in the line:

```
deploy = ServiceProxy("http://localhost:8080/axis/services  
/DeploySensorsService?wsdl")
```

And finally you just need to save the file and to restart the game server.

B.2 System Execution

There are three main components which require to be executed at the same time - the webservice, the Battlefield 2 modified game server and the Commander's Interface.

The webservice can be executed by simply starting the Apache tomcat service, that under WindowsXP can be done by selecting "Start→Configure Tomcat" and then by pushing the "Start" button. This will automatically start also the Axis Platform that is located inside Apache Tomcat (after a proper installation of the system), and so it will also start the webservice called "DeploySensorsService".

The Battlefield 2 mod game server can be executed by browsing to the folder "Battlefield2/ServerConfig" and then executing the file "bf2server.exe". Note that before executing the server, it has to be configured with the correct IP, as specified in the README file inside the installation folder "Software/Battlefield 2 Mod".

The Commander Interface can be simply executed by browsing into its folder and then by clicking on the file "ComInterface.bat" if you are in a WindowsXP environment. If you are not under WindowsXP you have to open a command prompt, browse to the folder "CommanderInterface" and then type:

```
java -jar ComInterface.jar
```

Finally all the BF2 players that wants to use this infrastructure can start their game client modified with the Mod that is supposed to be correctly installed in their machine, as specified in the README file in the folder "Software/Battlefield 2 Mod". To start the game mod installed in a client machine you just need to browse to the folder of the game "Battlefield2" and then click on "RunBF2Client Debug.bat" or on "RunBF2Client Release.bat", depending if you want to run the game in debug¹ or in release². Note that before allowing

¹that means in a window at a lower resolution, with the possibility to see debug/error messages

²that means at higher resolution on full screen

the first player to connect to the BF2 server you should be sure to have stored the deployment of the sensors through the Commander Interface.

So, just to understand better the last sentence, let's see which is the typical sequence of operations in such a system:

1. Start the webservice, the BF2 mod server and the commander interface.
2. Use the commander interface to create sensors and select zones and then wait for the answer of the webservice, that will solve the problem (if there is a solution). At the same time, the WebService will store the sensor deployment into a static variable.
3. The players can now start their client of the game with the "Mod" installed and then connect to the server "PlanAndPlay Test Server" ³.
4. When the first player will connect to the BF2 server, this will ask to the webservice for the stored sensor deployment (previously set by the commander interface).

B.3 Hardware and Software dependencies

This section summarizes the hardware and software dependencies of the entire system.

B.3.1 Hardware dependencies

There are no particular hardware dependencies, except for Battlefield 2 on the client side. Indeed it is required a graphic card that has to be in the list of the graphic cards which are compatible with the game, this list is in the game requirements. Obviously also all the other requirements of the official game have to be respected.

³You will have noticed that somewhere we use the name "PlanAndPlay", this is because the "SensorDeployMod" developed for BF2 has adapted and expanded the code of the Mod developed by Daniele Masato in the project "Plan And Play".

Instead for the commander interface, the webservice and the BF2 server there are no particular hardware requirements.

B.3.2 Software dependencies

Java 1.6 Since the commander interface and the webservice are written in java they require the Java VM installed on the machines in which they will run. This is not included inside the CD, but you can download it from <http://java.sun.com>.

Apache Tomcat 6.0.2 This is an application server that allows application written in Java to be executed on the server by a client. This software is included in the CD.

Axis 1.4 This is a platform for developing and deploying webservices written in Java. It is itself a web application that has to be installed inside Tomcat. Also this software is included in the CD

Battlefield 2 and Patches The original Battlefield 2 game is for obvious reasons not included in the Installation CD. But the other patches that you will have to install are all attached.

choco-1.2.03 This is a Java library that is used by the webservice to solve the problem of deploying the sensors inside the zones in an optimal way. This type of solver is called a CSP⁴ solver, and so "choco" is a library to solve CSP's. This library is included in the installation CD.

B.4 Space and Memory requirements

Installing the Battlefield 2 server or client will require 2.3 GB of hard disk space, and this is the same also for the patches. Anyway if you will install the server, even if during the installation process they say that it is required

⁴Constraint Satisfaction Problem

to have 2.3 GB of free space, at the end of the day the space occupied by the server will be only 530 MB, already including the patches.

With regards to memory requirements, it is not recommended to run the Battlefield 2 server and the client on the same machine since if there is not enough memory the performances of the game could be compromised. Another important thing is that if the problem that the webservice has to solve is very hard (i.e. if there is a high number of zones and of sensors with different capabilities), then the webservice could start to use a lot of memory space, but also of CPU percentage work, to solve the problem.

B.5 Source File Description

As we stated previously, there are three main components and the source code of them is grouped inside the folder `src` inside the installation CD. The directory `src` contains three subdirectories that reflect the system architecture.

- `Webservice` - which contains the Java source code of the developed webservice
- `CommanderInterface` - which contains the Java source code of the Interface for the commander
- `Battlefield 2 Mod` - which contains the Python source code of the BF2 mod

B.5.1 Webservice source description

The Webservice is composed by one package "deploySensorsService" which contains a file "MyService.java" which implements the webservice, and a sub-package "deploySensorsService.solver" which contains all the classes that implement the solver of the problem to deploy sensors in the selected zones in an optimal way.

You will note that inside the folder `src/webservice` there are also other folders and installation files that need to be installed before modifying the webservice, since they are the platform that allows the webservice to work. Inside the folder `src/ConfigWebService` there are two files called `deploy.wsdd` and `undeploy.wsdd`, the first is the most important since you will have to use it to deploy the webservice inside the Axis platform (as it is well explained in the README file inside that folder), the second can be used in the case that you have to undeploy the webservice.

- Let's consider the package `deploySensorsService`:

MyService.java This class implements the webservice: the method `computeDeployment` performs the deployment of the sensors inside the zones, the other methods are used to return to the client the actual sensor deployment.

- Let's consider the subpackage `deploySensorsService.solver`, where the class that actually solve the problem is `DeploySensors.java` which uses `ZoneDeploy.java` as an auxiliary class. The other classes are data structures and auxiliary methods used by these two main classes:

DeploySensors.java This class performs the Sensor Assignment and then the Sensor Deployment of the sensors using the class `ZoneDeploy`. This reflects the actual model that divides the main problem into two subproblems: the Sensor Assignment and then the Sensor Deployment inside each zone of the sensors assigned to that zone.

ZoneDeploy.java This class solve the Sensor Deployment problem for each zone considering only the sensors assigned to the zone.

Sensor.java This is a data structure that represents the sensor and its properties.

CoveredArea.java This is a data structure that represents the zone selected by the commander and the information that is required from it.

SubArea.java This class represents a subzone created by division of a zone, this class is used in the algorithm that implements the Sensor Deployment problem solver.

MyList.java This class implements a list using an hashtable and it is used as a utility class by the others.

PairInt.java This class implements an object composed by a pair of integers

Utilities.java This class contains some utility function used by the classes "DeploySensors" and "ZoneDeploy".

B.5.2 Commander Interface source description

The commander interface is composed by one package "deploySensorsClient" which contains a file "MyClient.java" and a subpackage "deploySensorsClient.structures". The first is the main class of the application and it implements the command-line interface, the second contains all the data structures used by the interface to perform its tasks (i.e. to send the request to the webservice).

- Let's consider the package "deploySensorsClient":

MyClient.java This class implements the Commander's interface: it asks to the webservice for the solution of the problem whose parameters are set by the commander. This class uses the classes in "deploySensorClient.structures" to set the input parameters (sensors and zones) of the method "computeDeployment" of the webservice. It uses Axis libraries to communicate with the webservice.

- Let's consider the subpackage "deploySensorsClient.structures", where there are the data structures used by the commander interface to set the parameters of the problem. This classes are the same data structures used by the Webservice solver:

Sensor.java This is a data structure that represents the sensor and its properties.

CoveredArea.java This is a data structure that represents the zone selected by the commander and the information that is required from it.

SubArea.java This class represents a subzone created by division of a zone.

MyList.java This class implements a list using an hashtable and it is used as a utility class by the others.

B.5.3 Battlefield 2 Mod source description

Battlefield 2 allows to develop your own plug-ins for the server, this plug-ins are called "mod" and they are written in Python and inserted into the folder "mods/[YourModName]". Each "mod" has to respect a proper structure, so it will have to include certain folders and files; in this structure you can insert your own Python code inside the folder "mods/[YourModName]/Python/game" that has to have a fixed structure too, but you can add also your own Python modules.

So the source of Battlefield 2 Mod is contained in the folder "src/Battlefield 2 Mod/game" inside the installation CD and it has a fixed structure. The real core of the Mod is implemented inside the file "scoringCommon.py" that is entirely written by me, Diego Pizzocarò. We used also other utility modules that are "Utils.py" and "Defines.py" which were taken from the Honors Project of Daniele Masato, whose name is PlanAndPlay⁵.

- Let's consider the folder "game" it contains a folder "gamemodes" and other files of which the most important is "scoringCommon.py":

scoringCommon.py This file is the core of the mod: It asks to the webservice for the current sensor deployment (that had been set

⁵Indeed we took the same structure of the mod "PlanAndPlay" developed by Daniele Masato and, after having removed some part of the mod that we would not have used, we modified the file "scoringCommon.py".

before by the commander) and then it creates Sensitive Area inside the map simulating the behaviour of real sensors.

Utils.py it contains some of the utility methods used inside the "mod" of BF2. This file is more or less the same of the one written by Daniele Masato, except that we deleted some functions that were useless for our own "mod".

Defines.py it contains all the constants used inside the "mod" of BF2. This file is exactly the same of the one written by Daniele Masato except that we only use certain constants and not all of them.

B.6 Compiling and Updating the system

Let's explain how to compile/update the different components of the system.

B.6.1 Compiling the Webservice

To begin with, you could use an environment such as eclipse to edit the source code of the webservice. In this case you could create a new Java project, then import all the source files respecting the structure in packages (and so you have to create the packages inside eclipse), and finally you have to add to the project the needed libraries. For the last step you have to consider that to compile the project you have to add all the Jar files of Axis (contained in the folder where you installed Axis, inside the directory "axis-1.4/lib") and also the Jar file "choco-1.2.03".

Once you compiled the webservice you have to copy the compiled code inside Axis and then deploy the service. So the steps are:

1. Copy the folder "deploySensorsService" (you can leave into it the source code too) inside the folder where you installed Tomcat and in particular inside the directory "Tomcat 6.0/webapps/axis/WEB-INF/classes".

You will have to delete the old folder "deploySensorsService" contained inside this directory and then you will have to paste the new one.

2. Start the Apache Tomcat service
3. Open a command prompt and browse to the folder of the project where you have also the files "deploy.wsdd" and "undeploy.wsdd". Now you have to undeploy the service with this command:

```
java -cp %CLASSPATH% org.apache.axis.client.AdminClient undeploy.wsdd
```

Where it is supposed that you set properly the variable "%CLASSPATH%" as explained in the README.TXT file inside the folder "src/WebService".

4. Always using the command prompt, you have now to deploy the service. And you can do this by writing the following command:

```
java -cp %CLASSPATH% org.apache.axis.client.AdminClient deploy.wsdd
```

5. Now restart the Apache Tomcat service and the new modified service should work.

B.6.2 Compiling the Commander's Interface

Also in this case you could use Eclipse to edit the source code and compile it. Like previously explained you have to create a new project in Eclipse respecting the same package structure, and then to include the necessary libraries. In this case you have only to include the Jar file "choco-1.2.03". Once you compiled it you can also create a Jar File alway using Eclipse (or Netbeans if you prefer it).

B.6.3 Updating the Battlefield 2 Mod

As you probably know, Python does not need to be compiled, it is usually an interpreted language and in this case the server will automatically execute the Python source code. So once you modified the Python source code contained in the folder "src/Battlefield 2 Mod/game" inside the installation CD, you just need to browse inside the directory where the Mod is actually installed (i.e. "Battlefield 2/mods/PlanAndPlay/python"), then delete the folder "game" and replace it with the new modified one.

B.7 Known Bugs

Even if we tested the system quite a lot, it is likely that there are bugs in the system, which cannot be resolved because of the limited time assigned to this project. Here we document some known bugs in our system.

B.7.1 Battlefield 2 Mod Bug

The BF2 server asks to the webservice for the stored deployment only when the first player connects to the server. In the meanwhile the commander could use the commander's interface to set another sensor deployment, but this deployment will not be used until all the players disconnect from the server. When the last remaining player disconnect from the server, it will reboot itself⁶, so that all the sensors that were created inside the map are deleted. Now when a player connects again to the server, if it is the first who is going to connect, the server will ask again for the stored sensor deployment, and finally it will deploy the sensors inside the map.

The problem is located in the slice of time that the server spend to reboot itself. During this reboot time, a player can still connect to the server, but it

⁶In this case with the term "reboot" we mean the operations that the server carries on to reload all the Python modules.

will not have any deployment available, so in the map there will not be any sensor. In the future this bug could be resolved by not allowing players to connect to the server while it is rebooting.

B.7.2 Webservice "Solver" Bug

The solver inside the webservice cannot always understand when a problem (whose parameters are set by the commander) has no solution. When the problem is quite "easy" it manages to answer that there is no solution, but when there are too many variables it could go on forever trying to find a solution. In the future this could be resolved by using a time limit, so that after this fixed time the solver will stop to look for a solution and will answer that there is no available solution.