Comparing Winner Determination Algorithms for Mixed Multi-Unit Combinatorial Auctions

MSc Thesis (*Afstudeerscriptie*)

written by

Brammert Ottens
(born 29-08-1982 in Amsterdam)

under the supervision of **dr. Ulle Endriss** and **prof. dr. Krzysztof Apt**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
27-09-2007	prof. dr. Krzysztof Apt
	prof. dr. Peter van Emde Boas
	dr. Ulle Endriss
	dr. Evangelos Markakis
	Joel Uckelman

Institute for Logic, Language and Computation

abstract

In this thesis three different approaches to the Winner Determination Problem (WDP) for Mixed Multi-Unit Combinatorial Auctions (MMUCA) are explored. The first, due to [Cerquides *et al.*, 2007] is based on a traditional integer programming approach. The second, due to [Uckelman and Endriss, 2007] is based on constraint programing and the third is based on a division of the original problem in two subproblems.

These two subproblems are the problem of finding an allocation that produces enough goods, and the problem of finding a sequence of the transformations in this allocation. It is shown that both problems are NP-complete. However, in the case of an acyclic goods graph the problem of finding a sequence becomes trivial. This is also the case when any cycles present limit themselves to just one good, i.e. the good occurs both in the input and in the output of a transformations.

All three algorithms are tested using two different datasets. The first is created using the generator described in [Vinyals *et al.*, 2007] and discerns three different types of transformations, input, output and input-output transformations. The second is created using a generator that discerns four different types of transformations, namely input, output, structured and unstructured transformations.

In terms of performance the results are not conclusive. Although the constraint programming approach does not perform very well, the performance of both the integer programming approach and the division approach vary. In several situations the latter is better and in other situations the former is better. Furthermore, the influence of the number of transformations on the complexity of the auction varies for the different datasets.

However, the results do show that structured auctions are easier to solve than unstructured and hybrid auctions and that the division approach performs much better on structured auctions.

Acknowledgements

The first two people that I want to thank are Krzysztof Apt and Ulle Endriss. Although they did not know me very well they agreed to supervise my MoL thesis. Without them this thesis would have never seen the light of day.

I also want to thank Sebastian Brand for providing me with the code for his array constraints library, and Jesús Cerquides Bueno for providing me with the code of their generator.

This thesis marks the end of my studies at the University of Amsterdam. For the past seven years I have studied, with great pleasure, at this university. It has provided me with a paramount of opportunities to develop myself. However, all this would not have been possible without the support and help I have gotten from my parents. For the past seven years they have supported me in all my decisions.

Finally, my great thanks and admiration goes to my girlfriend, Mirjam, for putting up with me in these last hectic weeks of moving to another country, starting a new job, finding a house and finishing this thesis.

Contents

Chapter 1

Introduction

Auctions play an important part in our everyday life. They are used by the government to distribute the available radio frequencies. Most of the fruit and vegetables we eat and the plants we look at are bought at auctions at some point in the processing chain. On a smaller scale many goods are traded using auctions on sites like ebay.com or marktplaats.nl.

All these types of auctions have in common that there is an auctioneer that wants to sell something and a set of bidders that want to buy it from the auctioneer. The first extension of this situation is to allow several goods to be auctioned at the same time, in such a way that the bidders can bid on combinations of goods. Such an auction is called a combinatorial auction [Cramton *et al.*, 2006]. The second generalisation one can make is to allow goods to travel both from the auctioneer to the bidder and vice versa. A bidder now no longer bids on a set of goods, but on a set of "transformations", where a transformation consists of a set of input goods and a set of output goods. The input goods are given to the bidder by the auctioneer. The output goods are given to the auctioneer by the bidder. An auction consisting of such transformations is called a Mixed Multi-Unit Combinatorial Auction (MMUCA), and has been introduced in [Cerquides *et al.*, 2007].

In the end, everybody participating in an auction is interested in who wins the auction. That is, everybody is interested in who obtains the goods offered by the auctioneer or who is allowed to perform certain transformations. The problem of finding this winner is called the Winner Determination Problem (WDP). For standard auctions, the problem of finding a winner is relatively easy. However, for combinatorial auctions the problem becomes very hard to solve. This is due to the fact that the number of possible allocations increases exponentially with the number of goods that are up for sale. The WDP for MMUCAs has not yet been studied very thoroughly. In [Vinyals *et al.*, 2007] some experiments are performed on the WDP using an integer programming approach. The tests are performed on a testset, generated with a generator introduced in the same paper. Their results show that this approach is not scalable.

The purpose of this thesis is to learn more about the WDP for MMUCAs. To that end, two more methods of solving are introduced on top of the approach introduced in [Cerquides *et al.*, 2007]. A comparison of these three approaches is made using both the testset generator introduced by Vinyals *et al.* [2007] and

a new generator which is introduced in this thesis.

As has already been mentioned, the approach used by Vinyals *et al.* is based on integer programming. Integer programming is a tool that is widely used in the auctions community to solve the WDP for combinatorial auctions. The problem of using this approach in the MMUCA case, is that it introduces a quadratic number of variables in terms of the number of transformations in the auction.

To remedy this two new approaches are introduced that are designed in such a way that they reduce the amount of variables used. The first of these approaches is based on the idea of constraint programming. Constraint programming is a general technique in which one is able to define constraints over sets of variables. The advantage of constraint programming over integer programming is that the language that can be used to define constraints is much more flexible. Using this flexibility, the constraint programming uses a linear number of variables in terms of transformations present in the auction.

The second new approach uses the observation that the WDP for MMUCAs can be divided into two different subproblems. This division of the WDP into two subproblems is interesting in that it teaches something about what makes the WDP for MMUCAs hard to solve. Furthermore, it is shown that both subproblems are NP-complete in their own respect.

Although the WDP for MMUCAs is NP-complete [Cerquides *et al.*, 2007], it is still interesting to see what the performances of several solutions are in the case of realistic auctions. It might be the case that the problem is more easily solvable for such transformations. To this end, a dataset must be created. Two approaches to this are discussed in this thesis, both based on a different taxonomy of transformations. The first, taken from [Vinyals *et al.*, 2007], discerns three different types of transformations. The second discerns four different types of transformations.

The rest of this thesis is structured as follows. In Chapter 2 the different methods that can be used to solve the WDP are discussed. In Chapter 3 more detail is given on auction theory. Chapter 4 discusses the WDP for MMUCAs. In Chapter 5 the implementation of the different approaches are given and in Chapter 6 these algorithms are evaluated experimentally. Finally, Chapter 7 contains the conclusions.

Chapter 2

Methods

A wide range of problems can be described using constraints. The purpose of this Chapter is to introduce two different methods that can use constraints to solve a problem. The first method that is discussed, integer programming, can be used to solve a system of linear equations. The second method, constraint programming, is a much more general approach that can handle a wide array of constraints (including linear constraints).

Both methods have in common that the constraints must range over a set of variables. However, the inner workings differ very much. For integer programming fast and dedicated algorithms exists, whereas due to the generality of constraint programming the performance can be slower. On the other hand, constraint programming can be used to solve a wider array of problems and has more freedom in the kind of constraints that are used.

In this thesis, both methods are used to analyze the winner determination problem of mixed multi-unit combinatorial auctions. The WDP consists of a set of constraints and the way these constraints are represented are of great influence on the methods that can be used to solve the problem.

Integer programming is the method that is traditionally used to solve the WDP for all types of auctions. Above that, it is a widely used tool in many other fields. The use of constraint programming, on the other hand, is not as wide spread. However, the greater flexibility in available constraints makes it an interesting candidate for problems like the WDP.

The rest of this Chapter is structured as follows. In Section 1 integer programming is introduced, and in Section 2 constraint programming is introduced.

2.1 Integer Programming

Integer programming is a modification of linear programming. Linear programming, also called mathematical programming, involves the optimization of a linear objective function, who's variables are subject to linear equality and inequality constraints. Integer programming deals with the case where the variables are constraint to be integer.

In terms of complexity, there is a huge gap between linear and integer programming. Where the former is solvable in polynomial time, the latter is known to be NP-complete [Papadimitriou, 1981]. This complexity gap occurs because

in solving an integer programming problem one is required to search, whereas the solving of a linear programming problem can be done analytically.

The purpose of this section is to give some general information about integer programming. Readers interested in the method should read [Eiselt *et al.*, 2000] for a more informal introduction. The more mathematically inclined reader is referred to [Schrijver, 1986].

2.1.1 Linear programming

Remember that in high school, during math courses one had to solve questions like

- calculate where $5x + 6y = 10$ and $4x + 8y = 10$ coincide.
- look at the graph depicted in Figure 2.1 and determine where the two functions cross each other.

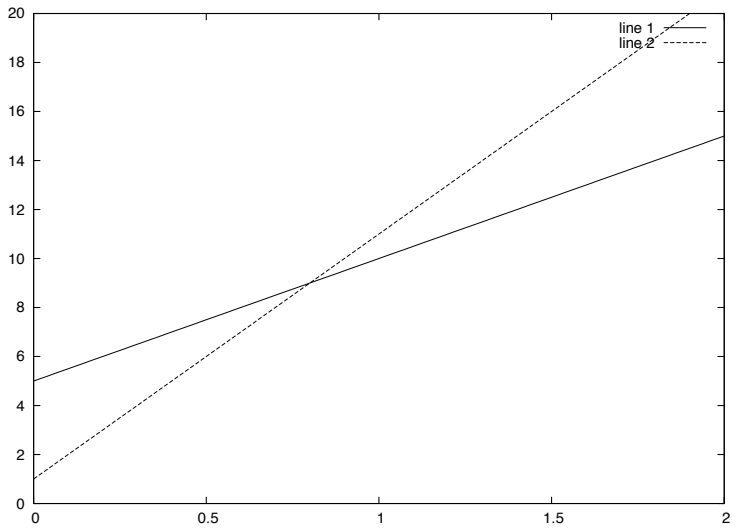


Figure 2.1: A high-school example

Both questions deal with linear equations. The first in a symbolic way and the latter in a geometrical way, but they are both sides of the same coin. In general, a linear equation follows the following definition.

Definition 1 (Linear equations) *A* linear equation *is of the form*

$$a_1x_1 + \ldots + a_nx_n \, OP \, b$$

Where $a_1, \ldots, a_n, b \in \mathbb{R}$, $x_1, \ldots, x_n$ *are variables ranging over the reals and* $OP \in \{=, \leq\}$

In terms of geometry, a linear equation must be seen as either a hyperplane or a halfplane in the $\mathbb{R}^n$. That is, $a_1x_1 + \ldots + a_nx_n = b$ defines a hyperplane

which cuts the $\mathbb{R}^2$ into two separate pieces. These pieces are called halfplanes and are defined by $a_1x_1 + \ldots + a_nx_n \leq b$ and $a_1x_1 + \ldots + a_nx_n \geq b$.

A more succinct representation of linear equations can be found using vector notation. For example, the equation $5x + 6y = 5$ can be written as

$$(56)\begin{pmatrix} x \\ y \end{pmatrix} = 5$$

and represents a line in the $\mathbb{R}^2$. In general, a linear equation in vector notation looks like

$$ax\,OP\,b$$

where a is a row vector, x a column vector and b a real number. A set of linear equations can now be written as a matrix equation. That is, the set of linear equations

$$\begin{array}{c} a_{1,1}x_1 + \ldots + a_{1,n}x_n \leq b_1 \\ a_{2,1}x_1 + \ldots + a_{2,n}x_n \leq b_2 \\ \vdots \\ a_{m,1}x_1 + \ldots + a_{m,n}x_n \leq b_m \end{array}$$

can be represented as the matrix equation

$$\begin{pmatrix} a_{1,1} & \ldots & a_{1,n} \\ a_{2,1} & \ldots & a_{2,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \ldots & a_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Given a matrix equation $Ax \leq b$, the set of solutions to this equation is defined as $\{x|Ax \leq b\}$, i.e. the set of solutions determines a set of vectors in $\mathbb{R}^n$. Moreover, one can interpret $\{x|Ax \leq b\}$ as the intersection of a finite number of half spaces. The resulting region is called a polyhedron.

Definition 2 (Polyhedron) *A set of vectors P in $\mathbb{R}^n$ is called a* polyhedron *if $P = \{x|Ax \leq b\}$ for some matrix A and some vector b.*

The set of solutions to a system of linear equations thus constitutes a polyhedron. Geometrically, a polyhedron is a region in space. For example, in the $\mathbb{R}^2$ a polyhedron may look like the region in Figure 2.2

An important property of a polyhedron is that it is convex. A region P is convex if, for every two vectors $x, y \in P$, the line segment defined by these two vectors lies inside P. The proof of the following proposition can be found in [Schrijver, 1986].

Proposition 1 *A polyhedron P is convex.*

Each point that lies inside the polyhedron is a solution to the original set of linear equations. However, linear programming is not concerned with finding just one solution. It wants to find the optimal solution given some vector c. That is, one tries to find $\max_x\{cx|Ax \leq b\}$.

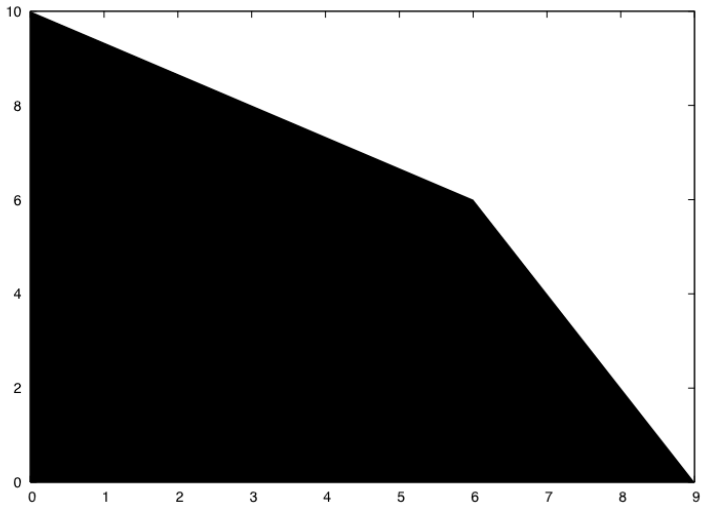


Figure 2.2: A polyhedron in $\mathbb{R}^2$

Let's go back to the geometrical interpretation for a second. Given some scalar δ, $\{x|cx = \delta\}$ determines a hyperplane. Finding the optimal solution now is equivalent to finding the hyperplane with the highest δ that intersects the polyhedron. In the case of the polyhedron in Figure 2.2, let

$$c = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

then $\{x|cx = \delta\}$ determines a line in the $\mathbb{R}^2$ and finding an optimal solution amounts to shifting the line over the region in the direction of improvement until the line with the highest δ has been found. Note that there will always be only one direction of improvement.

In the two-dimensional case, it is easy to see that the 'optimal' line must intersect with a corner point. For a generalization to the higher dimensional case, the reader is referred to §7.2 in [Dantzig, 1963].

Thus, in order to find an optimal point, one only has to visit the vertices. Even more, one can show that the number of vertices is finite. The first to have this insight was Fourier [Fourier, 1827]. It was algebrized by Dantzig [Dantzig, 1963], who gave the basis for the Simplex algorithm. The Simplex method is still the most popular algorithm used to solve linear programming problems.

2.1.2 Integer programming

The Simplex algorithm can be used to solve a system of equations, but it cannot be used to solve a system of equations that demands an integer solution. This is due to the fact that, in general, a vertex is not integer. Therefore, finding such an optimal integer solution requires searching. Several methods have been designed to do this. In this treatment one of them, the branch and bound algorithm, is explored in some detail.

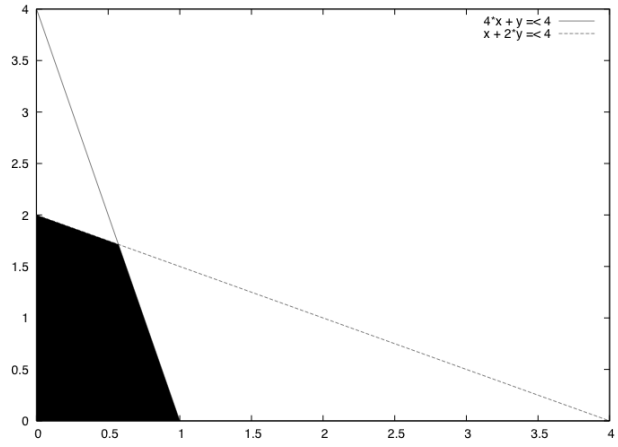


Figure 2.3: An integer programming problem

The basic idea behind the branch and bound algorithm is to first find a solution to the problem without the integrality constraints, and use this solution as a starting point to find an integer solution. How this actually works is best explained using an example. Suppose one has the following problem

$$\begin{aligned} z = \max_{x_1,x_2}\{x_1 + x_2\} & \\ \text{s.t.} \quad 4x + y & \leq 4 \\ x + 2y & \leq 4 \\ x, y & \in \mathbb{N} \end{aligned}$$

Geometrically, the solution must lie in the black area depicted in Figure 2.3

The first step is to relax the problem. That is, the problem is adjusted by replacing the integrality constraint for x and y with $x \geq 0$ and $y \geq 0$. Next, the solution to this relaxed problem is calculated. In this case, the optimal solution is $\{x = 0.571\,, y = 1.714, z = 2.286\}$. If one looks at Figure 2.3, one can see that this point lies on a vertex of the polyhedron.

The next step is to choose a variable that does not have an integer solution, and use the optimal value of this variable to split the problem in two new problems by adding extra constraints. In our example, $x = 0.571$. Since x must be an integer, it is either the case that $x \leq 0$ or $x \geq 1$. Adding the first constraint results in the optimal solution $\{x = 0, y = 2, z = 2\}$. Adding the second constraint results in the optimal solution $\{x = 1, y = 0, z = 1\}$. Note that both solutions are integer solutions. The solution with the highest value for z is $\{x = 0, y = 2\}$ and constitutes the optimal solution to the original problem. In most cases, one does not find an integer solution in the first step, and one has to repeat the previous steps several times.

This splitting operation results in a tree that has to be searched. Each node in the tree constitutes a linear problem, and the splitting constitutes the branching part of the branch and bound algorithm. The bounding part operates on the following two observations. First, a solution that does not produce integer values for the variables gives an upper bound on the final optimal value. That

is, since the non-integer solution is optimal, an integer solution cannot produce a higher optimal value. Second, an integer solution produces a lower bound for the optimal value, because one is not interested in z values that are lower than the value already found.

Using this bounding, parts of the search tree can be pruned away. That is, if at some node the optimal value is lower than what has already been found, all the nodes below this node can be discarded for the simple reason that none of his children can produce a higher z value.

As a final note, the way in which the tree is actually expanded can vary and depends on the problem at hand.

2.2 Constraint programming

This section is based on [Apt, 2003]. Constraint programming is a method one can use to solve a wide range of problems. The sole requirement is that they must be representable in the form of constraints over a set of variables. Good examples are scheduling problems, construction problems but also crossword puzzles or sudokus.

In general, constraint programming consists of two different tasks. First, the problem must be modelled. This means that one has to determine which variables are to be used, what their respective domains are and in what language the constraints must be specified. The second task is to solve the problem using the model created in the previous step. Obviously, the solving stage is strongly influenced by the modeling phase. That is, the types of domains and the constraint language used determine what kind of solving methods can be used.

In solving a CSP one can choose to use a specific solving method that is designed for a specific type of constraint problem, or one can use a more general approach. A good example of a specific type of solver is a solver based on the Simplex algorithm. In this Section, the more general approach is introduced.

2.2.1 Modelling phase

The first step in solving a problem using constraint programming (CP) is to determine the variables, the domains of these variables and the language in which the constraints over the variables are specified. For example, look at the sudoku puzzle shown in Figure 2.4. A traditional sudoku consists of nine large squares that are all subdivided into nine smaller squares. Each small square can be filled with a number ranging from 1 to 9. However, the way these squares can be filled is restricted. That is, collections of nine small squares that together form one large square must all contain different numbers. Furthermore, a single number is not allowed to occur more than once in a column or a row.

To model such a sudoku, each small square is assigned a variable $x_{i,j}$ where i is the horizontal coordinate, j the vertical coordinate and $x_{0,0}$ points to the lower left small square. The domain of each variable is $[1 \dots 9]$. Thus, each variable is an integer variable. As far as constraints are concerned, one needs a way to define that a sequence of variables all have a different value. This can be done by either using the $\neq$, or by a special constraint called *alldifferent*. In

		6			7			2
	3	4	8		9			
8			2	1		3		
	1		9			7	2	
					3			
6		9	5					8
		2						
					8			4
				5	1			

Figure 2.4: Sudoku

case of the Sudoku in Figure 2.4, this results in the following model consisting of the variables

$$x_{0,0} \in [1 \dots 9], \dots, x_{8,8} \in [1 \dots 9]$$

And the constraints

$$
\begin{aligned}
&alldifferent(x_{0,0}, x_{0,1}, x_{0,2}, x_{1,0}, x_{1,1}, x_{1,2}, x_{2,0}, x_{2,1}, x_{2,2})\\
&alldifferent(x_{0,3}, x_{0,4}, x_{0,5}, x_{1,3}, x_{1,4}, x_{1,5}, x_{2,3}, x_{2,4}, x_{2,5})\\
&\vdots\\
&alldifferent(x_{6,6}, x_{6,7}, x_{6,8}, x_{7,6}, x_{7,7}, x_{7,8}, x_{8,6}, x_{8,7}, x_{8,8})
\end{aligned}
$$

$$
\begin{aligned}
&alldifferent(x_{0,0}, x_{0,1}, \ldots, x_{0,8})\\
&alldifferent(x_{1,0}, x_{1,1}, \ldots, x_{1,8})\\
&\vdots\\
&alldifferent(x_{8,0}, x_{8,1}, \ldots, x_{8,8})
\end{aligned}
$$

$$
\begin{aligned}
&alldifferent(x_{0,0}, x_{1,0}, \ldots, x_{8,0})\\
&alldifferent(x_{0,1}, x_{1,1}, \ldots, x_{8,1})\\
&\vdots\\
&alldifferent(x_{0,8}, x_{1,8}, \ldots, x_{8,8})
\end{aligned}
$$

$$
\begin{aligned}
&x_{0,4} = 5\\
&x_{0,5} = 1\\
&\vdots\\
&x_{8,8} = 2
\end{aligned}
$$

In general, a model is called a Constraint Satisfaction Problem (CSP), and has the following form.

Definition 3 (Constraint Satisfaction problem) *Given a sequence of variables $X = x_1, \ldots, x_n$ and their respective domains D_i, a CSP is of the form $\langle \mathcal{C}, \mathcal{BE} \rangle$. $\mathcal{BE} = \{x_1 \in D_1, \ldots, x_n \in D_1\}$ and $\mathcal{C}$ is a set of constraints, where a constraint C over variables $x_{i_1}, \ldots, x_{i_m}$ is of the form $C \subseteq D_{i_1} \times \ldots \times D_{i_m}$.*

In the end, one is interested in a solution for the problem. Before the notion of a solution can be discussed, the following concepts need to be introduced. First, a specific constraint C over the variables $x_{i_1}, \ldots, x_{i_m}$ is called a *solved* constraint if $C = D_{i_1} \times \ldots \times D_{i_m}$, with $m > 0$ and each D_{i_k} is non-empty. That is, the constraint and the domains coincide In the case of $m = 0$, the only two constraints that are allowed are $\bot$ and $\top$. The former is always false and the latter is always solved. A CSP is called *solved* if all its constraints are solved. If, given some CSP, either one of its constraints is $\bot$ or one of the variables has an empty domain, this CSP is called *failed*.

A solved CSP, however, is not equal to a solution. In short, a solution is an assignment of values to variables such that each constraint is satisfied. More formally.

Definition 4 (Solution) *Take a CSP $\langle \mathcal{C}, \mathcal{BE} \rangle$, where $\mathcal{BE} = \{x_1 \in D_1, \ldots, x_n \in D_n\}$. Then an* assignment *is a tuple $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$. A constraint C over the variables $x_{i_1} \ldots, x_{i_m}$ is satisfied by an assignment if $(d_{i_1}, \ldots, d_{i_m}) \in C$.*

A solution *is an assignment $(d_1, \ldots d_n)$ that satisfies all constraints in $\mathcal{C}$*

If it is possible to find a solution to a CSP, it is called *consistent* . A CSP that does not have any solution is an *inconsistent* CSP.

2.2.2 General method

After the modelling phase is completed, the next step is to find a solution. Given a CSP $\langle \mathcal{C}, x_1 \in D_1, \ldots, x_n \in D_n \rangle$, a solution merely is an element in $D_1 \times \ldots \times D_n$. A naive way of finding a solution is to just enumerate all the possible assignments and check whether they satisfy all the constraints. However, the number of possible assignments grows exponentially in the size of the domains, which makes this method highly inefficient. In order to increase efficiency, one has to find a way to reduce the search space. That is, one has to find a way in which one can reduce the size of the domains without loosing any solutions.

The key idea now is that one can transform a CSP to a form that is easier to solve. In general, there are two types of transformations. First, the information contained in the constraints can be used to create a new CSP. That is, one transforms the CSP to another, equivalent CSP. Second, one can take a CSP and cut it up in several smaller CSPs, under the condition that the union of these smaller CSPs is equivalent to the original CSP. That is, every solution to the original CSP is a solution to one of the smaller CSPs, and every solution to one of the smaller CSPs is a solution to the original CSP.

The former method is called constraint propagation and is performed by constraint solvers. The latter method is called splitting. In constraint programming, both methods are applied in an alternating method, resulting a tree like structure.

2.2.3 Constraint solvers

The purpose of a constraint solver is to transform a CSP to an equivalent CSP that is easier to solve. Such a transformation is performed by a proof rule, which has the following form

$$\frac{\phi}{\psi}$$

Here both ϕ and ψ are CSPs. If ϕ and ψ are equivalent, such a proof rule is called *equivalence preserving*.

A CSP can be changed in two different ways. One can use the information contained in the constraints to reduce the domains of certain variables. Rules that perform this kind of transformation are called *domain reduction rules*.

The second type of rule, called a *transformation rule*, can be used to simplify certain constraints by adding new variables and new constraints. Finally, no rule is allowed to remove any variable because that would make it possible for a non-solution to become a solution.

In general, a proof rule deals with one type of constraint while a CSP normally contains different types of constraints. An application of a proof rule must therefore be defined in such a way that it only affects the variables in the constraint on which the rule must be applied. Do note that affecting some variable domains means that other constraints over these variables must be adapted as well.

It is not always the case that the application of a rule has an effect, i.e. sometimes the CSP in the premise and the conclusion are not only equivalent, but also identical. If this is the case or a rule R cannot be applied on a CSP, this CSP is *closed under the application of R*. If the application of a rule R

does have an effect, i.e. the resulting CSP differs from the original CSP, the application of rule R is dubbed *relevant*.

A constraint solver consists of a finite set of rules. A sequence of applications of rules in a constraint solver produces a sequence of CSPs. This sequence is called a *derivation*. A derivation is successful if the last CSP is the first CSP that is solved, *failed* if the last CSP is the first failed CSP and *stabilizing* if the last CSP is closed under all the rules in the constraint solver used.

Ideally, one wants to have either a successful or a failed derivation. However, in most cases the best one can do is to create a stabilizing derivation. The last element in such a stabilizing derivation is not necessarily consistent. Therefore, most constraint solvers do not aim for consistency. Instead, one usually tries to design a constraint solver in such a way that it always derives a CSP that satisfies a notion of local consistency. The purpose of this is that local consistency, although it does not imply overall consistency, does make it easier to find a solution.

2.2.4 Searching

In general constraint propagation is not able to produce a solved CSP. Recall that it is possible to split a CSP into several smaller CSPs. With splitting, it is meant that the domain of a certain variable is chopped into little pieces. Each new subdomain can then be used to replace the old domain, creating a new CSP. Constraint propagation can be applied on these smaller CSPs, which on their turn can be split. This can continue until a solution is found or all possibilities are exhausted.

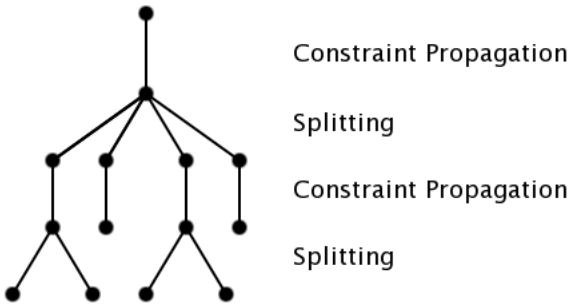


Figure 2.5: Search tree

A CSP can be split in various ways, but no matter what splitting method is used, the result is a tree like structure. In this tree, the original CSP is the root, each node at an uneven depth is the result of constraint propagation and each node at an even level is the result of splitting, see Figure 2.5. With the tree structure comes the problem of finding a leaf that contains a solved CSP. In other words, the tree must be searched. The usual search methods can be applied to navigate the tree. The interested reader is referred to [Apt, 2003] for more information on the role of searching in constraint programming.

2.2.5 Constraint optimization

Above, it is discussed how to find a solution to a CSP. However, in this thesis constraint programming is not used to find just any solution to a CSP, but an optimal solution. In general, such problems are called Constraint Optimization Problems (COP). Which solution is the best is decided using some objective function $f : D_1 \times \ldots \times D_n \mapsto \mathbb{R}$. Again, a naive method is to find all the solutions, calculate their respective f-values and determine the solution with the highest f-value. However, in traversing the tree certain parts of the tree are not interesting because they do not contain any solution with a f-value higher than the value already found. Such parts can thus be discarded in search for the optimal solution.

A search algorithm that uses this observation in order to prune the tree is the branch and bound algorithm. The variation of the algorithm that is used in this thesis traverses the search tree in a depth first manner. Each time it finds a solution it calculates its f-value and updates the bound for the optimal value. It only enters a branch that has the possibility of producing a solution with a higher f-value than has been found before.

2.3 Summary

Two methods have been introduced, integer programming and constraint programming. Both methods can be used to solve problems that consist of a collection of constraints. In integer programming, these constraints come in the form of linear equation. In constraint programming, the constraints can take on any from.

Integer programming is a modification of linear programming in which one adds integrality constraints. This turns it from a polynomial time problem to an NP-complete problem. Linear programming is used to find an optimal solution to a system of linear equations w.r.t. an objective linear function. One can show that an optimal solution must lie on one of the vertices of the polyhedron, defined by the system of linear equations. To find such a vertex, the Simplex algorithm has been developed. However, the Simplex algorithm only works if all the variables are allowed to take a non-integer value. In the case of integer constraints, the Simplex algorithm is used to find an initial solution that can be used as a starting point to search for an integer solution. The search algorithm that is used in this thesis is the branch and bound algorithm.

Constraint programming can handle a wide array of constraints. Where in integer programming the constraint language is fixed, choosing it constitutes an important decision in the constraint programming setting. This decision is part of the modeling process of constraint programming. Modeling a problem consists of deciding the variables, their domains and the constraint language over these variables. The end product is a constraint satisfaction problem (CSP).

The idea behind constraint programming is that one can use the information contained in the constraints to reduce the domains of the variables without searching. This reduction is performed by a constraint solver. A constraint solver consists of a set of transformation rules. However, in general a constraint

solver is not able to find a solution. Therefore, at some point a CSP must be split into several smaller CSPs on which the constraint solver can again be applied. This process of propagation and splitting results in a search tree that must be search for a solution.

If one is interested in the optimal solution to some problem, one can use several different search methods. The method used in this thesis is the branch and bound algorithm.

Chapter 3

Auctions

Auctions have been around for quite some time. According to the history books, the old Greeks used auctions to sell their women and Roman soldiers used auctions to sell their loot. During the French revolution, the common people used auctions to sell goods in taverns and bars, and today everybody can start an auction selling almost everything. Just think about eBay.com. This site enables everybody that has access to internet to either start or participate in an auction.

But it is not just ordinary people that use auctions. Even governments use auctions to sell radio or UMTS frequencies. Airports use them to allocate gates to airlines and some cities uses auctions to allocate bus-routes to bus companies.

As illustrated by the above examples, in today's world auctions play an important role. Therefore, it is important to understand what makes auctions tick. As a reaction to this need, the field of auction theory developed in the economic community in the 1960-1970 and has developed since. The goal of this chapter is to provide an introduction to auction theory, more specifically, to provide some details on combinatorial auctions and its generalisations.

The rest of this chapter is structured as follows. In Section 1, auction theory is introduced. In Section 2, the notion of a combinatorial auction is discussed. In Section 3, the winner determination problem for combinatorial auctions is treated. Finally, in Section 4 Mixed-Multi-Unit Combinatorial Auctions (MMUCA's) are introduced.

3.1 Auction theory

This section is based on Chapter 1 of [Klemperer, 2004]. Many different types of auctions are in use today. The most notable being the first-price sealed bid auction, the second-price sealed bid auction, the ascending bid or English auction and the descending bid or Dutch auction. All these auctions deal with a set of bidders, bidding for a single item.

The first-price sealed bid auction is an auction where all the bidders make their bid simultaneously and anonymously. The winner is the bidder with the highest bid and he must pay the value of its bid. The second-price auction is almost similar, except that the winner pays the value of the bid of the runner

up.

The ascending bid auction is an auction where the price is successively raised until only one bidder remains. This bidder is the winner and he must pay the final price. In a descending bid auction the price is successively dropped until one bidder calls in. That is, the price is dropped until one bidder accepts the current price. This bidder then is awarded the item for the current price.

Basically, in an auction one can discern two groups of players. On the one hand there are the bidders that each have a valuation for the good on sale. This valuation determines their bidding strategy. On the other hand we have the auctioneer. The auctioneer decides on the type of auction being held. Having two groups also means having two perspectives. From the bidder's perspective, the problem of solving an auction is equivalent to determining one's bid. From the auctioneer's perspective, the problem is twofold. First, he has to decide on the type of auction, and second, he must determine the winner of the auction

3.1.1 The bidder's point of view

In any auction, the sole task of the bidder is to determine the height of his bid. In making this decision, a bidder must take into account his actual value for the item on sale, but also his chances of winning, given a certain bid.

In terms of value, one can discern two types of auctions. The bidder's value is either private, i.e. each bidder only knows her own value, or the value is in some degree common. The former is called the *private-value* model and the latter the *common-value* model. In the common-value model, each bidder's value depends, to some degree, on the information possessed by him and other bidders. In this case, it becomes important whether a bidder is allowed to know other bidders' information or not.

The term *pure common-values* is used for the case where each bidders' value-function is an identical function dependent on the information it possesses, i.e. in principle all the bidders have the same value for the object on sale. However, they do not necessarily posses the same information.

In determining ones bid a bidder does not only need to take into account its own information and, possibly, the information of other bidders. He must also determine whether it pays to form a coalition with other bidders. That is, it might be profitable for a coalition of bidders to cooperate and make agreements on the division of profit.

3.1.2 The auctioneer's point of view

From an auctioneer's point of view, three things are important. First of all, the auctioneer wants to maximize its revenue. That is, when choosing a winner he wants to maximize his income. Second, the auctioneer wants that all the bidders report their true value. Third, the auctioneer wants to prevent bidders from forming a coalition.

The last two points are related to the first one in that, if it is either the case that some bidder does not report his true valuation or that several bidders form a coalition the auctioneer might not be able to maximize his revenue.

The first type of auction that has been thoroughly investigated by economists is the Vickrey auction. In [Vickrey, 1961], W. Vickrey investigated the properties of a second-price sealed bid auction, also known as the Vickrey auction.

The biggest virtue of the Vickrey auction is that truthful reporting of ones values is a dominant strategy. To see why this is the case, note that the bidder's payoff is defined by the bidder's true valuation, and the bid made by the runner up. This means that a bidder will not be able to raise his payoff by changing its bid. Thus, since not reporting truthfully cannot improve a bidder's payoff but might harm it, he is inclined to report its true valuation.

In fact, the Vickrey auction is not only an example of an auction where truthful reporting is the dominant strategy. Under certain assumptions, one can prove that it is the unique mechanism that has truthful reporting as its dominant strategy [Green and Laffont, 1979; Holmstrom, 1979]. The distinguishing factor in a Vickrey auction is the way its prices are calculated.

As already mentioned, the main goal of the auctioneer is, usually, to maximize its own revenue. This means that he must be able to identify the bidder that, if chosen as winner, results in the highest revenue. In the case of the first-price and second price sealed bid auctions, this amounts to choosing the bidder with the highest bid. In the case of the descending or ascending price auction, the winner will simply announce itself. The process of finding the winner is incorporated in the auction design.

In general, the problem of finding an allocation that maximizes the auctioneer's revenue is called the Winner Determination Problem (WDP).

3.2 Combinatorial auctions

Where the previous section dealt with single-item auctions, the present deals with auctions ranging over several items, also called combinatorial auctions. This Section only mentions some general results concerning combinatorial auctions. For a thorough overview, see [Cramton *et al.*, 2006].

The main difference between single item auctions and combinatorial auctions, is that in the latter the bidder is allowed to bid not only on a single item, but on a bundle of items. Such an auction is useful in the case where a combination of different items is worth more to a bidder than the individual items themselves. A nice, real world example of an auction where a combination of several goods can have an added value is the auctioning of radio spectra. A licence for, say the Amsterdam area is worth much more to a company if it also has licences for adjacent regions. It would not be beneficial for a radio operator to have gaps or holes in the area he covers.

Giving the bidders the possibility to bid on bundles of items allows them to better report their valuations. However, this enrichment also has its influence on the complexity of the auction. From the bidder's point of view, it gets much harder to determine its bids due to the explosion of combinations. An auction over k items results in 2^k different bundles. For example, 10 items result in $2^{10} = 1024$ different bundles. From the auctioneer's point of view, combinatorial auctions are both harder to design and to solve. This has resulted in several different auction designs. The following elements are found in all of them.

Definition 5 (Combinatorial auction) *A* combinatorial auction *consists of a set of bidders* $N = \{1, ..., n\}$ *and a set of goods* G*. A* bundle $S \subset G$ *is a set of*

goods, and $v_i(S)$ denotes the valuation of bidder i for bundle S. Furthermore, $\mathcal{F}_i$ denotes the set of bundles over which bidder i reports a bid.

Definition 6 (Allocation) *An* allocation *x is a vector such that $x_i \subseteq G$ represents the set of items obtained by bidder i. An allocation is a* feasible allocation *if $\forall g \in G \sum_{i \in N} x_i(g) \leq 1$.*

A typical and well studied example of a combinatorial auction is the Vickrey-Clarke-Groves auction [Clarke, 1971; Groves, 1973]. As the name already hints, the VCG auction is based on the single item Vickrey auction. All the bidders simultaneously make a bid on each possible bundle of items. The price the winners have to pay is determined by looking at the same auction, but ignoring the winning bidders. This results in the following definition.

Definition 7 (VCG winning allocation) *Let $\hat{v}_i$ be bidder $i's$ bid, the winning allocation x^* is defined by*

$$x^* \in argmax_{x_1,\ldots,x_n} \sum_i \hat{v}_i(x_i)$$

Definition 8 (VCG auction) *Let $\hat{v}_i$ be bidder $i's$ bid and let x^* be the winning allocation of some combinatorial auction. Then bidder i's payment p_i is calculated as $p_i = \alpha_i - \sum_{j \neq i} \hat{v}_j(x^*_j)$, where $\alpha_i = max\{\sum_{i \neq j} \hat{v}_j(x_j) | \sum_{i \neq j} x_j \leq 1\}$.*

Note that there is a difference between v and $\hat{v}$. The former denotes a bidder's true valuations, whereas the latter denotes the bidders reported valuation.

To give a better idea of what happens in a VCG auction, suppose there are two goods, A and B. Furthermore, suppose that there are two bidders, 1 and 2 with the bids

$$\begin{aligned}
&\hat{v}_1(A) = \hat{v}_1(B) = 10\\
&\hat{v}_1(AB) = 25\\
&\hat{v}_2(A) = 5\\
&\hat{v}_2(B) = 10\\
&\hat{v}_2(AB) = 20
\end{aligned}$$

The allocation that maximizes the auctioneer's revenue results in bidder 1 winning all the items. Now suppose that bidder 1 would not participate in the auction, then giving both A and B to bidder 2 maximizes the auctioneer's revenue. This means that bidder 1 has to pay $\hat{v}_2(AB) = 20$.

However, suppose that $\hat{v}_1(A) = 13$ and $\hat{v}_2(B) = 13$. This results in bidder 1 receiving good A for a price of $\hat{v}_2(AB) - \hat{v}_2(B) = 7$ and bidder 2 receiving good B for a price of $\hat{v}_1(AB) - \hat{v}_1(A) = 12$.

Just as in the single item case, the VCG auction as defined above has truthful reporting as a dominant strategy. However, the VCG auction comes with some flaws. It can result in a low revenue for the auctioneer, the sellers revenue is non-monotonic in the set of bidders and the amount of bids, it is vulnerable to coalition forming and it is vulnerable to the use of multiple bidding identities by a single bidder. To overcome the complexity of the standard combinatorial auction and weaknesses of the VCG auction, several other auctions designs have been proposed. See Chapter 1 of [Cramton *et al.*, 2006] for an analysis of these

weaknesses.

As a final note, the combinatorial auction as defined in Definition 5 deals with single-unit CAs. That is, each unit occurs exactly once. In the multi-unit case, the available goods are not represented by a set, but by a multiset of goods. The definition of a combinatorial auction now becomes:

Definition 9 (Multi-Unit Combinatorial Auction) *A* multi-unit combinatorial auction *consists of a set of bidders* $N = \{1, ..., n\}$ *and a multi-set of goods* $M = \{G, \sigma\}$*, where* G *is a set of goods and* $\sigma : G \mapsto \mathbb{N}$ *is a function that maps goods to their multiplicity. A* bundle $S \subseteq M$ *is a multiset of goods, and* $v_i(S)$ *denotes the valuation of bidder* i *for bundle* S*. Furthermore,* $\mathcal{F}_i$ *denotes the set of bundles over which bidder* i *reports a bid.*

3.3 WDP for combinatorial auctions

The explosion of the valuation function of different bidders makes the WDP for combinatorial auctions hard to solve. In fact, in general it is NP-hard [Rothkopf *et al.*, 1998]. This is due to the combinatorial property of the set of possible bundles. However, the bidding language can have an influence in certain restricted cases.

3.3.1 Bidding languages

A bidding language is a language in which the bidder is allowed to specify its bids. For a general overview, see Chapter 9 from [Cramton *et al.*, 2006] and [Nisan, 2000]. Just as there are different types of auctions, there exist different types of bidding languages. The bidding language determines the types of valuations that are representable. In this section, three general types of bidding languages are introduced. These are the atomic bids, the OR bids and the XOR bids. Most other existing languages are combinations of these three. All the results discussed below apply to the single-unit case only.

In general, a bidding language provides a succinct representation of a valuation. That is, a bidding language does not require a bidder to report his valuation for every possible bundle. In what way a bid represents the value for a missing bundle is partially defined by whether or not a bidder is willing to accept more items than he asks. This is formalized in the notion of free disposal.

Definition 10 (Free disposal) *A bidder satisfies free disposal if it is willing to accept more goods without demanding a change in payment.*

The atomic bid language

The simplest bidding language is the atomic bid language, also called the single-minded language. Using this language, a bidder can submit a pair (S, p), where S is bundle and p is the price that the bidder that submits this bid is willing to pay for S. In the case of free disposal, the valuation for a bundle T is $\hat{v}(T) = p$ if $S \subseteq T$ and $\hat{v}(T) = 0$ otherwise. Without free disposal, the $\subseteq$ can be replaced with $=$.

The OR language

The OR bidding language is already more powerful than the atomic bid language. Using the OR language, a bidder can submit a set of atomic bids (S_i, p_i). In reporting such a set, the bidder states that he accepts any combination of disjoint atomic bids in the set for the sum of their respective prices. In terms of the valuation represented by a bid, for a bundle T, $\hat{v}(T)$ is the maximum over all possible valid combinations of bundles W, of the value $\sum_{S_i \in W} p_i$. W is a valid combination if for all $S_i, S_j \in W$ with $i \neq j$, $S_i \bigcap S_j = \emptyset$. In other words, to calculate the valuation of some bundle T, one has to find the combination of bids with the highest combined value who's bundles form a subset of T. Again, in the case without free disposal a combination W is valid if $\bigcup W = T$.

The XOR language

Although the OR language is more expressive than the atomic language, it still cannot express all the possible valuations. For full expressivity, the XOR language must be used. As in the OR language, a bid consists of a set of atomic bids (S_i, p_i). However, in this case a bidder is willing to accept only one of the atomic bids in the set. In the case of free disposal, for any bundle T, $\hat{v}(T)$ is defined as $\max_{p_i | S_i \subseteq T} p_i$. The XOR language with free disposal is able to represent all monotonic valuations.

Definition 11 (Monotonic valuation) *A valuation v is monotonic if and only if $v(S) \leq v(S')$ implies $S \subseteq S'$.*

Without free disposal, the XOR language is able to represent all possible valuations. The biggest drawback of the XOR representation, however, is that it is not very succinct. In fact, the number of atomic bids that is needed to represent a certain valuation can explode in an exponential way.

In the multi-unit case, the XOR-language is not fully expressive anymore. This is due to the fact that in this situation there are an infinite number of possible bundles, making it impossible to create one single XOR statement that contains all the bundles.

3.3.2 Problem formulations

The WDP is the problem of finding the allocation that maximizes the social welfare. That is, it tries to solve the following equation

$$x^* = \max_{x_1,\ldots,x_n} \sum_i \hat{v}_i(x_i)$$

under the condition that $\forall g \in G \sum_{i \in N} x_i(g) \leq 1$. There are several ways to solve this problem. The formulations below are valid in the single-unit case. However, they are easily generalizable to the multi-unit case.

Integer Programming

The first method is to describe the problem using a set of equations, and then solving these equations using the methods developed in the field of Integer Programming (IP). IP is discussed in Chapter 2. The exact description of the

problem depends on the bidding language used. In the case of the OR language, a different representation of the allocation must be used. Instead of using a vector, an allocation is represented as a set of variables $x_i(S)$ for each possible bundle $S \subseteq M$ and each bidder i. $x_i(S) = 1$ means that bidder i is awarded bundle S. The WDP is formalized as:

Definition 12 ($\mathbf{WDP_{OR}}$)

$$max_{x_1,\ldots,x_n} \sum_{i=1}^{n} \sum_{S \subseteq M} \hat{v}_i(S) x_i(S)$$

$$\sum_{i=1}^{n} \sum_{S \subseteq M, g \in S} x_i(S) \leq 1 \quad \forall g \in G$$

$$x_i(S) \in \{0, 1\}$$

Note that in this definition, a bidder can receive different disjoint subsets. If, instead of the OR language the XOR language is used, a third equation must be added that ensures that each bidder cannot be assigned more than one bundle. This results in the following formalization.

Definition 13 ($\mathbf{WDP_{XOR}}$)

$$max_{x_1,\ldots,x_n} \sum_{i=1}^{n} \sum_{S \subseteq M} \hat{v}_i(S) x_i(S)$$

$$\sum_{i=1}^{n} \sum_{S \subseteq M, g \in S} x_i(S) \leq 1 \quad \forall g \in G$$

$$\sum_{S \subseteq M} x_i(S) \leq 1 \quad \forall i \in N$$

$$x_i(S) \in \{0, 1\}$$

Note that each sum over all the subsets results in an exponential blowup, in terms of the size of M.

Intersection graphs

Another representation that can be used to solve the WDP is the notion of an intersection graph. Let $\mathcal{G} = (U, E)$ be a graph, with U a set of nodes and E a set of edges. For this representation to work, each item in M must be interpreted as a unique item. Thus, M becomes equal to a normal set. Each node $u \in U$ now represents a bid $\hat{v}_i(S)$ for $S \in \mathcal{F}_i$, and an edge between two nodes represents a conflict between two bids. In the case of the OR language, two bids are in conflict if they both contain the same item. In the case of the XOR language, two bids are in conflict if they either overlap, or are made by the same bidder. The value of each bid is represented with a weight. That is, each node u, linked to bid $\hat{v}_i(S)$ gets the weight $w_u = \hat{v}_i(S)$ attached.

Using this representation, an allocation is represented by a set of nodes that are not connected with each other. In graph theory, such a set of nodes is called a *stable set*. The problem of finding the winner now equals the problem of finding the maximum weighted stable set. That is, an unconnected set of nodes that maximizes the sum of their respective weights.

3.4 Mixed Multi-Unit Combinatorial Auctions

The types of auctions introduced above all dealt with an auctioneer that wanted to sell items, and a set of bidders that wanted to buy an item. The case of a reversed auction, where the bidders sell items and the auctioneer wants to buy items, follows that same reasoning. In each case, the flow of goods is one way, either from the auctioneer to the bidder, or vise versa.

Mixed Multi-Unit Combinatorial Auctions (MMUCA), introduced in [Cerquides *et al.*, 2007], are a generalization of both types of auctions. That is, in a MMUCA goods can flow both ways. A nice example is the construction of a car. Suppose that you, as an auctioneer, have all the items to build a car. However, you do not not have the faintest clue of how to actually build a car. To this end, you organize an auction where auto mechanics can bid on the assignment (or a sub assignment) of building a car. For example, one mechanic can submit a bid for building the engine, while another mechanic can opt for constructing the wheels. The main characteristic is that each bid consists of some items going to the bidder, and some items going to the auctioneer. This bring us to the definition of a transformation.

Definition 14 (Transformation) *Let G be the set of goods that is available in the auction. Then a transformation is a pair of multisets over G. That is, a transformation is a pair of multisets $(\mathcal{I}, \mathcal{O}) \in \mathbb{N}^G \times \mathbb{N}^G$.*

In the setting of a MMUCA, an agent can bid on a bundle of transformations.

Definition 15 (Bundle) *A bundle S is a multiset of transformations. Thus, a bundle S is an element of $\mathbb{N}^{(\mathbb{N}^G \times \mathbb{N}^G)}$.*

For example, the bundle $\{(\{\}, \{a\}), (\{b\}, \{c\})\}$ means that an agent that submits a bid for this bundle is able to deliver a and produce c if it is presented with b. Recall that in section 3.3.1, several bidding languages where introduced that could be used by a bidder to convey its valuation function. From the information in the bid, its entire valuation function could be reproduced using the notion of a subset. That is, the value of some bundle was determined by finding out whether it contained some bundle for which the valuation was known. In the case of transformations, it is not possible to use the normal subset relation. Therefore, the notion of *subsumption* is introduced.

Definition 16 (Subsumption) *Given two transformations $(\mathcal{I}, \mathcal{O})$ and $(\mathcal{I}', \mathcal{O}')$, $(\mathcal{I}, \mathcal{O})$ is said to be subsumed by $(\mathcal{I}', \mathcal{O}')$ iff $\mathcal{I} \subseteq \mathcal{I}'$ and $\mathcal{O} \supseteq \mathcal{O}'$. Extended to bundles, the subsumption relation becomes as follows*

Let $S, S' \in \mathbb{N}^{(\mathbb{N}^G \times \mathbb{N}^G)}$ be two bundles. S subsumes S', denoted by $S \sqsubseteq S'$ iff:

1. *S and S' have the same cardinality: $|S| = |S'|$*
2. *There exists a surjective mapping $f : S \mapsto S'$ such that, for all transformations $t \in S$, we have that $t \sqsubseteq f(t)$*

3.4.1 Bidding languages

Atomic bids

The simplest possible bidding language is the language consisting of atomic bids. Just as in the normal combinatorial auction, an atomic bid *Bid* in a MMUCA consists of a bundle of transformations S, combined with a price p, i.e. $Bid = (S, p)$. From a collection of atomic bids over transformations a bidder's valuation can be obtained.

XOR bids

In this treatment, the WDP for MMUCAs is investigated in terms of the XOR language. Therefore, only the XOR language is treated. Also, most bidding languages can be represented using an XOR bid (see [Cerquides *et al.*, 2007]), thus looking only at the XOR language does not effect the expressivity available.

A XOR bid basically is a combination of atomic bids, with the restriction that the bidder is willing to accept exactly one of the atomic bids in the bid. So, let $Bid = Bid_1$ XOR ... XOR Bid_n be a bid, with Bid_i being an atomic bid, then the valuation defined by Bid is

$$v_{Bid}(S) = \max\{v_{Bid_i}(S) | i \in \{1, ..., n\}\}$$

Just as in combinatorial auctions, the presence of free disposal has an influence on the expressivity of the language. However, since both the bidder and the auctioneer can receive goods, the definition must be adjusted to the new situation.

Definition 17 (Free disposal) *Both a bidder and an auctioneer can follow the assumption of* free disposal *. On the bidders side, free disposal amounts to the fact that the bidder is always prepared to accept more goods, and give fewer goods away without wanting a change in payment. On the auctioneer's side, free disposal means that the auctioneer is willing to accept more goods and give less goods away.*

The former influences the bidder's valuation function, whereas the latter influences the definition of a proper solution to the WDP for MMUCAs.

In the case of free disposal at the bidder's side, an atomic bid $Bid = (S, p)$ results in the following valuation

$$v_{Bid}(S') = \begin{cases} p & \text{if } S \sqsubseteq S' \\ \perp & \text{otherwise} \end{cases}$$

In the case where free disposal is not valid, the valuation is obtained by replacing the $\sqsubseteq$ in the previous definition with $=$.

In order to talk about the expressive power of the XOR language for MMUCAs, the following definitions are needed.

Definition 18 (Monotonic closure) *The monotonic closure $\tilde{v}$ of a valuation v is defined as $\tilde{v}(S) = max\{v(S') | S' \sqsubseteq S\}$*

As in the multi-unit case of standard combinatorial auctions, the existence of an infinite number of possible bundles posses a problem in terms of expressive power. The following definition is therefore needed.

Definition 19 (Finitely-peaked valuation) *A valuation is* finitely-peaked *iff v is only defined over finite multisets of pairs of finite multisets and $\{S \in \mathbb{N}^{(\mathbb{N}^G \times \mathbb{N}^G)} | v(S) \neq \bot\}$ is finite*

The following theorems, due to [Cerquides *et al.*, 2007], show the expressive power of the XOR language

Theorem 1 *The XOR language without free disposal can represent all finitely-peaked valuations, and only those.*

Theorem 2 *The XOR language with free disposal can represent all valuations that are the monotonic closure of a finitely-peaked valuation, and only those.*

3.4.2 WDP for MMUCAs

The introduction of transformations has added to the complexity of the WDP for MMUCAs. Where in the combinatorial auction an allocation was valid as long as it did not allocate more goods than available, the design of the MMUCAs makes the *applicability* of transformations important as well. A transformation is applicable if all the input goods are available. Therefore, the notion of a sequence of transformations becomes important. In other words, an allocation is only valid if, on top of assigning available goods, the assigned goods can form a sequence of applicable transformations. The next chapter is focused on algorithms for solving the WDP of a MMUCA.

3.5 Summary

In this Chapter, several notions and concepts connected with auction theory have been introduced. First, the notion of a single item auction has been discussed and it has been shown what the problems and considerations are that occur when designing an auction. A type of auction that promotes truthful reporting, the Vickrey auction, has been introduced.

As a generalization of the single item auction, the notion of a combinatorial auction has been introduced as well. The biggest difference between a single item auction and a combinatorial auction is that in the latter a bidder is allowed to bid for a combination of items. The motivation behind this enhancement is that sometimes, a bundle of items is worth more to a bidder than the individual items themselves. In that case the whole is more than the sum of its parts. As in the single item case, a generalization of the Vickrey auction is given that also promotes truthful reporting. The addition of combinatorial bids greatly enhances the complexity of the auction. From the perspective of the bidder, it gets much harder to determine ones valuations. From the perspective of the auctioneer, it becomes much harder to both design the auction and determine its winner. In fact, the problem of determining the winner of such an auction becomes NP-complete.

Finally, the notion of a combinatorial auction is generalized to a Mixed Multi-Unit Combinatorial Auction. In such an auction, a bidder does not bid on a set of items. Instead, he bids on a set of transformations. A transformation represents the process of transforming some goods to other goods. For example, a transformation could state that a bidder is able to transform dough to cookies.

To top things off, the XOR bidding language has been introduced, which is able to represent a wide array of valuations.

Chapter 4

The Winner Determination Problem

The winner determination problem for mixed multi-unit combinatorial auctions differs from the standard winner determination problem. The biggest difference is that, for an assignment to be a proper assignment, several additional conditions must hold. The purpose of this Chapter is to formalize these conditions and to suggest several algorithms to solve the WDP for MMUCAs

The rest of this Chapter is structured as follows. In Section 1 the WDP for MMUCAs is discussed. In Section 2 a division of the WDP is discussed. In Section 3 three different approaches to solving the WDP are discussed.

4.1 The WDP for MMUCAs

As in standard auctions and combinatorial auctions, solving the WDP in MMUCAs means finding a proper allocation of transformations that maximizes the auctioneer's revenue. The main difference, however, is the definition of a proper allocation. Remember that in normal auctions, an allocation is proper if it does not contain more goods than available.

The structure of the WDP for MMUCAs, however, is more complex. As input, the WDP gets a set of bids made by the bidders, a multiset of input goods $\mathcal{U}_{\text{in}}$ and a multiset of goods $\mathcal{U}_{\text{out}}$ that is requested by the auctioneer. Thus, where in a normal auction one only has to take into account the available goods, a MMUCA forces you to also take into account what the result of the transformations must be. Furthermore, the bidders do not bid on goods, but provide transformations. Thus, an allocation does not state which bidder gets which good, but which bidder is allowed to apply which transformation. The applicability of a transformation is very important.

Definition 20 (Applicable) *A transformation* $(\mathcal{I}, \mathcal{O})$ *is* applicable*, given some multiset of goods* G*, if* $\mathcal{I} \subseteq G$*.*

An allocation is *proper* if one is able to create a sequence of transformations in which each transformation is applicable and the result is what is requested by the auctioneer. In the following, the XOR bidding language is assumed.

Definition 21 (MMUCA) *A MMUCA consists of a set of input goods $\mathcal{U}_{in}$, a set of goods $\mathcal{U}_{out}$ requested by the auctioneer, and a set of XOR bids B. The bid of bidder i is represented by B_i, and $B_{ij} = (D_{ij}, p_{ij})$ is the jth atomic bid in bidder i's bid. $D_{ij} \in \mathbb{N}^{\mathbb{N}^G \times \mathbb{N}^G}$ is a multiset of transformations and p_{ij} is the price. Furthermore, t_{ijk} points to the kth transformation in D_{ij}, given some predefined ordering, and T is the set of all transformations.*

Informally, a proper allocation is a sequence of transformations that satisfies the following requirements.

- The multiset of transformations must satisfy the structure of the bids that are reported.
- The sequence must be *implementable*. That is, each transformation in the sequence must be applicable after all his predecessors have been applied. Furthermore, after all the transformations have been applied the set of goods that is delivered must be a superset of $\mathcal{U}_{\text{out}}$.

In the case of the XOR language, the first requirement restricts the auctioneer in terms of which transformations to accept. That is, if one of the atomic bids made by a bidder is accepted, no other atomic bids in the same XOR bid can be accepted.

The second requirement is captured using the following equations. Let $\mathcal{M}^m(g)$ be the amount of good g available after all the transformation up to position m have been applied. One can assume that $\mathcal{M}^0(g) = \mathcal{U}_{\text{in}}(g)$. Then $\mathcal{M}^m(g)$ is defined as

$$\mathcal{M}^m(g) = \mathcal{M}^{m-1}(g) + (\mathcal{O}^m(g) - \mathcal{I}^m(g)) \tag{4.1}$$

Here $\mathcal{O}^m(g)$ and $\mathcal{I}^m(g)$ respectively denote the output goods and the input goods of the transformation in the m^{th} position of the sequence. The requirement of applicability can be modelled using $\mathcal{M}^m$ in the following way

$$\mathcal{M}^{m-1}(g) \geq \mathcal{I}^m(g) \quad \forall g \in G \tag{4.2}$$

A proper allocation can now be defined using the function $\mathcal{M}^m(g)$ as follows.

Definition 22 (Proper allocation) *Given a set of input goods $\mathcal{U}_{in}$, a set of output goods $\mathcal{U}_{out}$ and a set of bids B, an allocation Σ is* proper *if it satisfies the following requirements*

- *If the auctioneer selects one transformation from an atomic bid, it must select all the transformations in this atomic bid*

$$t_{ijk} \in \Sigma \Rightarrow t_{ijk'} \in \Sigma$$

- *If an auctioneer selects two transformations from the same bidder, they must be in the same bid*

$$t_{ijk}, t_{ij'k'} \in \Sigma \Rightarrow j = j'$$

- *Each transformation in Σ must satisfy equation 4.2 for each good g, making each transformation applicable.*

- *The allocation* Σ *must* at least *produce what the auctioneer requested*

$$\mathcal{M}^{|\Sigma|}(g) \geq \mathcal{U}_{out}(g) \quad \forall g \in G$$

Note that the above definition assumes free disposal. In the case without free disposal, the $\geq$ in the last requirement must be replaced with $=$. The WDP for MMUCAs can now be formalized as follows.

Definition 23 (WDP for MMUCAs) *Given a set of input goods* $\mathcal{U}_{in}$*, a set of output goods* $\mathcal{U}_{out}$ *and a set of XOR bids* B*, solving the winner determination problem amounts to finding a proper allocation* Σ*, such that it maximizes* $\sum_{\exists k\, t_{ijk} \in \Sigma} p_{ij}$*.*

4.2 Division of the WDP

The complexity of the WDP for a MMUCA is NP-complete because, as stated in [Cerquides *et al.*, 2007], it can simulate a normal combinatorial auction. However, the WDP for MMUCAs does not only require to find an optimal allocation, it also requires a sequence consisting of all the transformations in the allocation.

One can thus decompose the WDP for MMUCAs in two separate problems. The first problem is to find an optimal set of transformations Σ that satisfy the requirements posed by the bidding language and combined with $\mathcal{U}_{\text{in}}$ produces enough goods to satisfy $\mathcal{U}_{\text{out}}$, i.e. to find a set of transformations that satisfies Equation 4.3.

$$\mathcal{U}_{\text{in}}(g) + \sum_i \sum_j b_{ij} \cdot (\mathcal{O}_{i,j}(g) - \mathcal{I}_{i,j}(g)) \geq \mathcal{U}_{\text{out}}(g) \quad \forall g \in \mathcal{G} \tag{4.3}$$

The second problem is the problem of finding a proper sequence using all the transformations in some set Σ. More formally, the second problem can be defined as:

Definition 24 (Finding a sequence) *Given a set of transformations* Σ *and a set of goods* $\mathcal{U}_{in}$*, let* $\mathcal{G}$ *be the set of goods that occur in* Σ*. Furthermore, let* Σ *be such that* $\forall g \in \mathcal{G} \quad \mathcal{U}_{in}(g) + \sum_{t \in \Sigma}(O_t(g) - I_t(g)) \geq 0$*. Find a sequence* $t_1, ..., t_n$ *with* $t_i \in \Sigma$*, such that it contains all the transformations in* Σ *and*

$$\forall m \quad \mathcal{U}_{in}(g) + \sum_{i=1}^{m-1} (O_i(g) - I_i(g)) \geq I_m$$

where m *ranges from 1 to* n*.*

Because this decomposition gives rise to a non-standard approach to the WDP, as shall be demonstrated in Chapter 5, it is interesting what the complexity of the two problems is.

In the case of the XOR language, it is easy to see that the WDP for a normal combinatorial auction can be simulated using the first problem. Simply replace every good g in an atomic bid with the transformation $(\{g\}, \emptyset)$. Hence, this first problem is NP-complete. Also note that the first problem resembles the winner determination for exchange auctions [Sandholm *et al.*, 2002].

Concerning the second problem one can show that using a solution to the sequence finding problem, the Hamiltonian directed path problem can be solved. The Hamiltonian directed path problem is NP-complete [Garey and Johnson, 1979] and therefore the problem defined in Definition 24 is NP-complete as well.

Formally, the Hamiltonian path problem is defined as:

Definition 25 (Hamiltonian directed path problem (HDPP)) *Given a graph $\mathbb{G} = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$, find a path through the graph that visits every vertex exactly once.*

To show that the HDPP can be solved using a solution to the sequence finding problem, a reduction from the HDPP to this problem must be found. The first step in finding such a reduction is to realize that each vertex can be represented as a transformation. To this end, label each vertex in the graph. Then a transformation representing a vertex takes as input the label of this vertex, and as output the labels of the successor vertices. More formally,

Definition 26 (From vertex to transformation) *Given a graph $\mathbb{G} = (V, E)$, associated with each vertex $v \in V$ is a transformation of the form $(\{v\}, O)$, where $O = \{v' | (v, v') \in E\}$.*

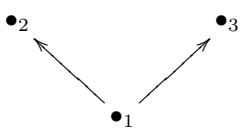
However, restricting us to just this set of transformations is not sufficient. Each transformation belonging to a vertex v produces the labels of all its successor vertices. However, there is no way to remove the remaining labels after the successor is chosen. This means that finding a sequence will not guarantee the existence of a path. Therefore, several restrictions must be built into the translation in order to make sure that after choosing a successor all the produced labels are removed.

Instead of deciding which vertex to proceed with the vertices that are not used are to be selected first. The trick now becomes to postpone choosing a successor until the choice becomes trivial, i.e. until only one label is left. To this end the goods e and d are introduced. The good e can only be produced using a label, and the d can only be produced by as many e's as there are successors, minus one. That is, if a label v has n successors, $n - 1$ e's must be used to produce a d. Now, only after the d has been produced the next successor transformation can be applied, making sure that all the other labels have been removed. The translation from vertex to transformation now becomes:

Definition 27 (From vertex to transformation no. 2) *Given a graph $\mathbb{G} = (V, E)$, associated with each vertex $v \in V$ is a set of transformations, containing only the following transformations, where $O = \{v' | (v, v') \in E\}$*

- $(\{v, d, e\}, O \cup \{v\})$
- *for each $v' \in O$,* $(\{v'\}, e)$
- $(E, \{d\})$*, with $E = \{e, ..., e\}$ such that $|E| = |O| - 1$ (Only if $|O| > 0$).*

For an example of how this procedure works, look at the following graph



Using the second definition, the above graph defines the following set of transformations

- $(\{1, d, e\}, \{1, 2, 3\})$
- $(\{1\}, \{e\})$
- $(\{2\}, \{e\})$
- $(\{3\}, \{e\})$
- $(\{e\}, \{d\})$
- $(\{2, d, e\}, \{2\})$
- $(\{3, d, e\}, \{3\})$

The only two paths through the graph are $1, 2$ and $1, 3$. This means that, given $\mathcal{U}_{\text{in}} = \{1, d, e\}$, it should be possible to make a sequence using all the transformations belonging to nodes 1 and 2, or all the transformations belonging to nodes 1 and 3, but not a sequence using all the transformations. That this is indeed the case can be seen from the fact that one must use either good 2 or good 3 for the next node transformation to be applicable, rendering the other transformation unusable.

Next, it is shown that the above translation provides a proper translation of the Hamiltonian directed path problem to the problem of finding a sequence

Proposition 2 *Given a graph $\mathbb{G} = (V, E)$, let S be the set of transformations that is created using Definition 27. $\mathbb{G}$ contains a Hamiltonian directed path iff for some $v \in V$, it is possible to create a sequence using all the transformations in $S \cup \{(\{v\}, \{e\})\}$ and $\mathcal{U}_{in} = \{v, d, e\}$.*

Proof.

⇒ Suppose that $\mathbb{G}$ contains a Hamiltonian directed path. This means that there exists a path $v_1, ..., v_n$, with v_i being the ith vertex on the path, that contains all the vertices. Using this path, a sequence of transformations can easily be created. Start by setting $\mathcal{U}_{\text{in}} = \{v_1, d, e\}$. Next, the only transformation that is applicable is $(\{v_1, d, e\}, O_1 \cup \{v_1\})$. Having applied this transformation, one can apply $(\{v_1\}, \{e\})$ because v_1 is not needed anymore. Next, by construction, it must be the case that $v_2 \in O_1$. Furthermore, let $Rem_1 = \{(\{v\}, \{e\}) | v \in O_1 \setminus v_2\} \subseteq S$. That is, Rem_1 contains the transformations belonging to all the successor vertices of v_1, except for the transformation belonging to v_2. Thus, using Rem_1, one can remove all the labels that do not correspond to v_2, producing exactly enough e to produce a d with $(E_1, \{d\})$. This leaves us with the goods $\{v_2, d, e\}$, and only $(\{v_2, d, e\}, O_2 \cup \{v_2\})$ is applicable. The rest of the sequence can be finished as follows. Suppose that the last transformation that has been applied is of the form $(\{v_i, d, e\}, O_i \cup \{v_i\})$. By construction, one can remove v_i by applying $(\{v_i\}, \{e\})$. Also, again by construction, it is the case that $v_{i+1} \in O_i$. Let $Rem_i = \{(\{v\}, \{e\}) | O_i \setminus \{v_{i+1}\}\} \subseteq S$. Using the transformations in Rem_i, all but v_{i+1} are removed from the set of available goods, and enough e is produced to create a d with $(E_i, \{d\})$.

Thus, after this only $(\{v_{i+1}, d, e\}, O_{i+1} \cup \{v_{i+1}\})$ is applicable. Note that each time the last transformation that is associated with v_i and all but one of the transformations that are associated with v_{i+1} are used. The sequence ends with $(\{v_n, d, e\}, O \cup \{v_n\})$ and after that the transformations in $\{(\{v\}, e)|v \in O\}$, $(E_n, \{d\})$ and $(\{v_{n-1}\}, \{e\})$. At each step i, $2 + |O_i|$ transformations are used, except at the last step, where $|O_n| + 3$ transformations are used. Thus in total $1+2n+\sum_{i=1}^{n} |O_i|$ transformations are used. By definition, $|S| = 2n + \sum_{i=1}^{n} |O_i|$. This, combined with the transformation $(\{v_1\}, \{Se\})$ that was added at the beginning, justifies the conclusion that all the transformations have been used in the sequence.

$\Leftarrow$ Suppose that for some $v_1 \in V$, it is possible to create a sequence $s_1, ..., s_n$, using all the transformations in $S \cup \{(\{v_1\}, e)\}$. We show that from the sequence $s_1, ..., s_n$ a Hamiltonian directed path in $\mathbb{G}$ can be constructed. By construction, $s_1 = (\{v_1, d, e\}, O_1 \cup \{v_1\})$. Let $v_1, ..., v_m$ be a sequence, with v_i the vertex such that $(\{v_i, d, e\}, O_i \cup \{v_i\})$ is the first transformation of this form since $(\{v_{i-1}, d, e\}, O_{i_1} \cup \{v_{i-1}\})$ occurred in the sequence $s_1, ..., s_n$. By construction, the number of transformations of the form $(\{v_i, d\}, O_i \cup \{v_i, d\})$ is equal to the number of vertices in $\mathbb{G}$. This, combined with the fact that a transformation can be used only once make sure that the sequence visits every vertex exactly once. What remains to be shown is that $v_1, ..., v_m$ is a path. Suppose that it is not. This would mean that at some point v_{l+1} is not a successor of v_l. By construction however, after $(\{v_i, e, d\}, O_i \cup \{v_i\})$ has been applied, $O_i \cup \{v_i\}$ are the only goods that are available. To see why this is the case, first note that $\mathcal{U}_{\text{in}} = \{v_1, e, d\}$. After application of $(\{v_1, e, d\}, O_1 \cup \{v_1\})$, only $O_1 \cup \{v_1\}$ are available. Next, assume that after application of $(\{v_i, e, d\}, O_i \cup \{v_i\})$ only $O_i \cup \{v_i\}$ are available. This means that at the moment only $\{(\{v\}, e)|v \in O_i\}$ and $(\{v_i\}, e)$ are applicable. In order for a sequence to continue, all but one of the transformations in $\{(\{v\}, e)|v \in O\}$ together with $(\{v_i\}, e)$ can be applied. After this, $(E_i, \{d\})$ and $(\{v_{i+1}\}, e)$ are applicable. Again, in order for the sequence to continue, only $(E_i, \{d\})$ can be applied. Now, $\{v_{i+1}, e, d\}$ are the only goods that are available. This means that only $(\{v_{i+1}, e, d\}, O_{i+1} \cup \{v_{i+1}\})$ is applicable, using up all the available goods. Thus, at this point only $O_{i+1} \cup \{v_{i+1}\}$ are available.

Hence, after application of $(\{v_l, e, d\}, O_l \cup \{v_l\})$ only the goods $O_l \cup \{v_l\}$ are available. Because $(\{v_{l+1}, e, d\}, O_{l+1} \cup \{v_{l+1}\})$ is the next transformation of this form, it must be the case that $v_{l+1} \in O$. This can only be the case if v_{l+1} is a successor of v_l and thus $v_1, ..., v_m$ must be a path.

QED

Proposition 2 thus gives us a way of determining whether a graph contains an Hamiltonian directed path.

Theorem 3 (NP-completeness) *The problem of finding a sequence, defined in Definition 24 is NP-complete*

Proof. It is known that the problem of determining whether a graph $\mathbb{G}$ contains a Hamiltonian directed path is NP-complete [Garey and Johnson, 1979]. Proposition 2 shows that Definition 27 can be used to translate a graph into a

set of transformations in such a way that a sequence can only be found if the graph contains a Hamiltonian path. However, one must take into account the fact that the translation assumes that the starting point of the path is known. To actually solve the path problem, one must do the following.

Given graph $\mathbb{G}$, create a set of transformations S according to Definition 27. The problem thus is that it is not known beforehand on which vertex the path starts. Therefore, each vertex must be tested as a starting point until a path has been found, or all the vertices have been checked. After choosing a vertex v to test, let $\mathcal{U}_{\text{in}_v} = \{v, d, e\}$ and $S_v = S \cup \{(\{v\}, \{e\})\}$. To determine whether S_v contains a sequence, one first tests whether $\forall g \in \mathcal{G} \quad \mathcal{U}_{\text{in}_v}(g) + \sum_{s \in S_v}(O_s(g) - I_s(G)) \geq 0$. If S_v combined with $\mathcal{U}_{\text{in}_v}$ satisfies this condition, try to find a sequence using all the transformations in S_v. If this succeeds, then by Proposition 2, $\mathcal{G}$ contains a Hamiltonian directed path. If no sequence exists, continue with the next vertex.

The translation can be performed in a polynomial amount of time, since the number of translations is only dependent on the number of vertices and the number of successors per vertex. The maximal number of times a test is performed is a equal to the total number of vertices in the graph. Therefore, finding a sequence must be NP-complete. QED

The problem of finding a sequence given a fixed number of transformations is thus NP-complete. However, it is interesting to see whether there are properties that a set of transformations can have that make it less hard to solve. To this end, the notion of a goods graph is introduced. Each transformation contains input and output goods. The goods graph is a directed graph where there is an edge between two goods, say a and b, if there is some transformation such that a occurs in this transformations as an input good and b as an output good. More formally,

Definition 28 (Goods graph) *Given a set of transformations T, let G be the set of goods that occur in T. Then the goods graph $\mathbb{G}$ belonging to T is a directed graph such that $\mathbb{G} = (G, E)$, where $(a, b) \in E$ if there is some transformation $(\mathcal{I}, \mathcal{O}) \in T$ such that $a \in \mathcal{I}$ and $b \in \mathcal{O}$.*

There is one property of the goods graph that is responsible for the complexity of the second problem. To see what this property is, first let's look at the following auction. Let $\mathcal{U}_{\text{in}} = \{a\}$ and $\mathcal{U}_{\text{out}} = \{d\}$. Furthermore, let $(\{a\}, \{b\})$, $(\{b\}, \{c\})$ and $(\{c\}, \{d\})$ be the available transformations. It is easy to see that the transformations produce enough goods to satisfy the requirements posed by $\mathcal{U}_{\text{out}}$. Furthermore, there is only one transformation that can be used in the first slot of the sequence, after that only one sequence that can be used in the second slot and only one sequence that can be used in the final slot of the sequence. In other words, the problem of finding a sequence is trivial in this instance.

Now let's look at the following transformation. Let $\mathcal{U}_{\text{in}} = \{a\}$ and $\mathcal{U}_{\text{out}} = \{c\}$. The available transformations are $(\{a\}, \{b\})$, $(\{b\}, \{a\})$ and $(\{a\}, \{c\})$. Now recall that **all** the transformations must be used in the sequence. However, when creating this sequence two transformations can be used for the first slot, namely $(\{a\}, \{b\})$ and $(\{a\}, \{c\})$. The problem, however, is that if $(\{a\}, \{c\})$ is chosen there is no other transformation that is applicable. Whilst if $(\{a\}, \{b\})$

is chosen the sequence can be finished. Thus, where in the previous example no choice has to be made in constructing the sequence there is a choice to be made in this example.

The difference between the previous two auction is the occurrence of cycles in the goods graph. More generally, one can state that if there are no cycles in the goods graph, the problem of finding a sequence is trivial. To prove this, the reverse is proved. Namely that if it is not trivial to find a sequence, the goods graph should contain a cycle.

Definition 29 (Building a sequence) *Given a set of transformations $\mathcal{S}$ and a multi-set of initial goods $\mathcal{U}_{in}$, one can build a sequence $t_1, ..., t_n$ using the following procedure.*

- *Start by picking a transformation from $t_1 \in \mathcal{S}$, with $t_1 = (I_1, O_1)$ that is applicable. That means that all the goods that are required as an input are available in $\mathcal{U}_{in}$. Furthermore, let $\mathcal{S}_1 = \mathcal{S} \setminus t_1$ and $\mathcal{U}_{in_1} = \mathcal{U}_{in} \setminus I_1 \cup O_1$.*
- *Next, apply the following operation until either $\mathcal{S}_n = \emptyset$ or no transformation is applicable.*
  - *select a transformation t_{i+1} from $\mathcal{S}_i$ such that $\mathcal{U}_{in_i}$ contains all the goods that are required.*
  - *Let $\mathcal{S}_{i+1} = \mathcal{S}_i \setminus t_{i+1}$ and $\mathcal{U}_{in_{i+1}} = \mathcal{U}_{in_i} \setminus I_{i+1} \cup O_{i+1}$.*

A sequence is called a proper sequence if it uses all the transformations in $\mathcal{S}$.

Proposition 3 *Given an allocation A that satisfies Equation 4.3, let $\mathcal{S}$ be all the transformations in A. Furthermore, let $\mathbb{G}_{\mathcal{S}}$ be the goods graph. $\mathbb{G}_{\mathcal{S}}$ contains a cycle if one can build a sequence such that at some point i one is not able to pick a transformation that is applicable while $\mathcal{S}_{i-1} \neq \emptyset$.*

Proof. Suppose that, given some set of transformations $\mathcal{S}$ that satisfies Equation 4.3, one can build a sequence in such a way that at point i no transformation is applicable, while $\mathcal{S}_{i-1} \neq \emptyset$. This means that for each transformation $t \in \mathcal{S}_{i-1}$, some required goods are not available. Let's pick out one of these goods. Because $\mathcal{S}$ satisfies Equation 4.3 it must be the case that there is some transformation $t' \in \mathcal{S}_{i-1}$ that produces this good. We now show that t' is dependent on t.

Let T_t denote the set of all the transformations that produce some good that is needed by t such that $T_t \subseteq \mathcal{S}_{i-1}$. Now assume that there is no transformation t such that some $t' \in T_t$ is dependent on t. This means that for each $t' \in T_t$, $T_{t'} \subseteq \mathcal{S}_{i-1} \setminus t$. This again means that for each $t'' \in T_{t'}$, $T_{t''} \subseteq \mathcal{S}_{i-1} \setminus \{t, t'\}$. Due to the fact that the number of transformations is finite, in the end there must be some transformation $t_e \in \mathcal{S}_{i-1}$ such that $T_{t_e} = \emptyset$. But this would mean that allocation A does not satisfy Equation 4.3. Thus, there is some transformation $t \in \mathcal{S}_{i-1}$ such that some transformation $t' \in T_t$ is dependent on t.

So, we now know that there is a path in $\mathbb{G}_{\mathcal{S}}$ from some good that is produced by t to some good that is needed by t'. By definition t' produces something that is needed by t, thus there is a cycle in $\mathbb{G}_{\mathcal{S}}$ QED

Thus, if there are no cycles in the the goods graph, the problem of finding a sequence is trivial. But suppose it does contain a cycle, will it always be the case that it is not trivial to solve the problem? I.e. are there some types of cycles for which it is still not important which applicable transformation is used in which slot? The answer to this question is positive, for suppose that the only cycles that occur are cycles of the form $(\{a\}, \{a\})$. In general, the problem with cycles is that they hold certain goods "hostage" that are needed further on in the transformation. To see what is meant by this, look at the previous example of a set of transformations with one good. The cycle consists of the goods a and b. Thus, if transformation $(\{a\}, \{b\})$ is performed, good a is taken hostage by this transformation. It is not until transformation $(\{b\}, \{a\})$ is performed that good a is released again so that it can be used by transformation $(\{a\}, \{c\})$.

If the cycle is of the form (a, a), this means that there is a transformation that has a both in the input and the output. Therefore, a is by no means "taken hostage" by the transformation. Therefore, transformations of this sort still allow the sequence to be found very quickly.

That it is not unlikely to come across such transformations can be seen from the observation that in certain transformations tools can be needed. For example, to create a cupboard from some boards and a set of nails, one is likely to need a hammer. However, after the construction the hammer is not a part of the cupboard. Thus, a transformation for a cupboard would need a hammer as input but would also deliver the hammer as output.

4.3 Solving the WDP

Traditionally, the WDP is solved using integer programming. However, in the case of MMUCAs, using integer programming amounts to using a quadratic number of variables. The purpose of this section is to introduce three different attempts to solve the WDP for MMUCAs. The first approach is a traditional approach using an integer programming solver. The second approach tries to reduce the number of variables used by switching to another constraint language. Finally, the third approach uses the fact that the WDP for MMUCAs consists of two different problems, both formulated using integer programming.

4.3.1 A first IP approach

The approach discussed in this section is based on the formalism of integer programming. The problem formulation is taken from [Cerquides *et al.*, 2007]. The final goal of the WDP is to find a proper sequence. Which transformation occurs where in the sequence is represented by a set of decision variables $x^m_{ijk} \in \{0, 1\}$. x^m_{ijk} takes on the value 1 if transformation t_{ijk} occurs at position m in the sequence, and 0 otherwise. Do note that, although the length of the sequence is not known beforehand, the maximum length is $\sum_i \max_j\{|B_{ij}|\}$. That is, the maximal length of a sequence cannot be longer then the sum of the sizes of the biggest atomic bids made by each bidder. Therefore, $1 \leq m \leq \sum_i \max_j\{|B_{ij}|\}$.

Furthermore, several other decision variables are used. For a each transformation t_{ijk}, a variable $x_{ijk} \in \{0, 1\}$ is used to store whether the transformation occurs in the sequence. That is, $x_{ijk} = 1$ if t_{ijk} occurs in the sequence and 0 otherwise.

For each atomic bid B_{ij} a variable $x_{ij} \in \{0,1\}$ denotes whether it has been accepted. And finally, for each position m in the sequence, the variable $x^m \in \{0,1\}$ denotes whether it is filled.

Using the above introduced decision variables, the following equations model the WDP

1. One of the requirements of the XOR bidding language is that if one of the transformations from an atomic bid have been chosen, all the transformations from this atomic bid must be used in the sequence. This is modelled using the following equation

$$x_{ij} = x_{ijk} \quad \forall ijk \tag{4.4}$$

2. A second requirement of the XOR language is that per bidder, at most one atomic bid can be chosen. The following equation models this

$$\sum_j x_{ij} \leq 1 \quad \forall i \tag{4.5}$$

3. The following equation makes a connection between the x_{ijk} and x^m_{ijk} variables. That is, it models the fact that if a variable is chosen, it must occur in the sequence and vice versa

$$x_{ijk} = \sum_m x^m_{ijk} \quad \forall ijk \tag{4.6}$$

4. By definition, a sequence defines a total order on a set of transformations. Therefore, only one transformation can occur at position m in the sequence. This is formalized as

$$x^m = \sum_{ijk} x^m_{ijk} \quad \forall m \tag{4.7}$$

5. The following equation makes sure that there are no gaps in the sequence. Although it is not essential that there are no gaps in the sequence, it does reduce the search space.

$$x^m \geq x^{m+1} \quad \forall m \tag{4.8}$$

6. Each transformation in the sequence must be applicable. To this end, in Section 4.1 the equations 4.1 and 4.2 have been defined. The following two equations capture the definition of $\mathcal{M}$ and enforce applicability of each transformation in the sequence.

$$\mathcal{M}^m(g) = \mathcal{U}_{\text{in}}(g) + \sum_{l=1}^{m} \sum_{ijk} x^l_{ijk} \cdot (\mathcal{O}_{ijk}(g) - \mathcal{I}_{ijk}(g)) \quad \forall g \in G\,, \forall m \tag{4.9}$$

$$\mathcal{M}^{m-1}(g) \geq \sum_{ijk} x^m_{ijk} \cdot \mathcal{I}_{ijk}(g) \quad \forall g \in G\,, \forall m \tag{4.10}$$

7. Finally, $\mathcal{U}_{\text{out}}$ resembles the goods that are requested by the auctioneer. Therefore, after applying the final transformation in the sequence, at least all the goods in $\mathcal{U}_{\text{out}}$ must be present. This is captured via the following equation
$$\mathcal{M}^{\sum_i \max_j \{|B_{ij}|\}}(g) \geq \mathcal{U}_{\text{out}}(g) \quad \forall g \in G \tag{4.11}$$

The above equations resemble the constraints that are posed on the variables. In other words, they assure that only a proper allocation is a possible solution. However, the WDP is not interested in just any solution, but in the solution that maximizes the auctioneer's revenue. To this end, the following integer program is defined

$$\max \sum_{ij} x_{ij} \cdot p_{ij} \quad \text{subject to the conditions (4.4)-(4.11)}$$

After solving the above programme, a solution to the WDP can be found by selecting the transformations t^m_{ijk} such that x^m_{ijk} equals 1.

The biggest drawback of the formulation introduced above is that it requires a quadratic number of variables, in terms of the number of transformations. This is due to the fact that for each position in the sequence, and each transformation, a variable occurs. The following sections both propose formulations that remedy this deficiency.

4.3.2 A CP approach

The problem with integer programming is that one is bound by the language of linear equations. The nice thing about constraint programming is that one is not bound by any language constraint, only the solving power of the constraint solver at hand. This means that several other constraints might be introduced that reduce the number of variables used.

Although the following formulation, which is an extension of the formulation defined in [Uckelman and Endriss, 2007], resembles the above formulation very much, several distinct features are used. First of all the *alldifferent* constraint is used, and second a variable index to an array is introduced in order reduce the number of variables.

The variables and their use differ somewhat from the previous approach. First of all, each transformation t_{ijk} is assigned a number n_{ijk}. The variable x_n now contains the position at which transformation n occurs in the sequence, and x^m contains the number of the transformation at position m in the sequence. There is an obvious link between x^m and x_n, which is represented by the following two equations

$$x^{x_n} = n \quad \forall n \tag{4.12}$$

$$x_{x^m} = m \quad \forall m \tag{4.13}$$

These two equations state that x_n and x^m agree with each other on which transformation is where in the sequence. Furthermore it must be the case that

$$\text{All the } x_n \text{ are different and all the } x^m \text{ are different} \tag{4.14}$$

The problem now is that *every* transformation must have a position somewhere in the sequence. Therefore, one is not able to disambiguate between accepted and rejected transformations. To this end, a set of dummy transformation is introduced, combined with a set of dummy positions in the sequence. That is, the first $\sum_i \max_j\{|B_{ij}|\}$ positions are real positions, and the remaining $|T|$ are dummy positions. Thus, only accepted transformations occur in a real position, and the rejected transformations occur in a dummy position. The set of dummy transformations is used to fill the real and dummy positions that have not been filled with real transformations. Note that only the real positions need to be constrained. This brings us to the following set of constraints

1. The following equation is used to record whether transformation t_{ijk} is used in the sequence. b_{ijk} equals 1 if t_{ijk} does occur in the sequence and 0 otherwise.

$$b_{ijk} = \begin{cases} 1 & \text{if } x_{n_{ijk}} \leq \sum_i \max_j\{|B_{ij}|\} \\ 0 & \text{otherwise} \end{cases} \qquad \forall \text{ "real" } ijk \tag{4.15}$$

2. The variable $b_{ij} \in \{0,1\}$ is used to encode whether an atomic bid has been accepted. The following equation ensures that if an atomic bid has been chosen, each transformation in this bid is used. Furthermore, it ensures that a bid is chosen if any of its transformations is used.

$$b_{ij} = b_{ijk} \quad \forall i \tag{4.16}$$

3. Again, one is allowed to accept at most one bid per bidder

$$\sum_j b_{ij} \leq 1 \quad \forall i \tag{4.17}$$

4. The applicability of each transformation is captured using

$$\mathcal{M}^m(g) = \mathcal{U}_{\text{in}}(g) + \sum_{l=1}^{m} b^l_{x^l} \cdot (\mathcal{O}_{x^l}(g) - \mathcal{I}_{x^l}(g)) \quad \forall g \in G\,, \forall m \tag{4.18}$$

$$\mathcal{M}^{m-1}(g) \geq \mathcal{I}_{x^m}(g) \quad \forall g \in G\,, \forall m \tag{4.19}$$

5. And finally, the sequence must at least produce the amount of goods in $\mathcal{U}_{\text{out}}$

$$\mathcal{M}^{\sum_i \max_j\{|B_{ij}|\}}(g) \geq \mathcal{U}_{\text{out}}(g) \tag{4.20}$$

In order to find the allocation that maximizes the auctioneer's revenue, the following must be maximized

$$\max \sum_{ij} b_{ij} \cdot p_{ij} \quad \text{subject to the constraints (4.12)-(4.20)}$$

Note that the above formulation allows gaps in the sequence. Again, gaps do not pose a problem in terms of finding a solution. However, in searching for the optimal solution, allowing gaps means that one has to deal with a lot of equivalent solutions. Therefore, gaps should not be allowed. To this end, a

decision variable $b^m \in \{0,1\}$ is introduced. b^m equals 1 if position m is *not* filled, and 0 if it is filled. The following constraints now enforce a sequence without gaps

$$b^m \leq b^{m+1} \forall m$$

$$b^m = \begin{cases} 1 & x^m \geq \sum_i \max_j\{|B_{ij}|\} \\ \textit{undecided} & \text{otherwise} \end{cases} \quad \forall m$$

4.3.3 A second IP approach

The final approach that is discussed treats the WDP as two separate problems. First, note that one can look at the combination of a set of transformations as one big transformation. That is, given some set of transformations Σ, the total number of input goods required is $\mathcal{I}^{\Sigma}_{total}(g) = \sum_{(\mathcal{I},\mathcal{O})\in\Sigma} \mathcal{I}(g)$, and the total number of output goods produced is $\mathcal{O}^{\Sigma}_{total}(g) = \sum_{(\mathcal{I},\mathcal{O})\in\Sigma} \mathcal{O}(g)$. The first problem now is to find, given $\mathcal{U}_{\text{in}}$, $\mathcal{U}_{\text{out}}$ and B, a set of transformations Σ that maximizes the auctioneer's revenue, and satisfies the requirements posed by the bidding language such that $\mathcal{U}_{\text{in}}(g) + \mathcal{O}^{\Sigma}_{total}(g) - \mathcal{I}^{\Sigma}_{total}(g) \geq \mathcal{U}_{\text{out}}(g)$.

The second problem is to find a proper sequence using *all* the transformations in Σ found by solving the previous problem.

Of course it is not the case that solving the WDP amounts to first solving the first problem and then the second. It can very easily be the case that a set of transformations Σ satisfying the constraints posed in the first problem, does not contain any proper sequence. Therefore, both problems need to be solved sequentially until either a solution is found, or all possible sets of transformations have been looked at.

The main advantage of this formulation is that it greatly reduces the amount of variables needed in order to solve the problem. The drawback is that in the worst case scenario, all possible valid sets of transformations must be explored. Experimentation must show how often this situation occurs.

The first problem can be formalized using the integer programming methodology as follows, where $b_{ij} \in \{0,1\}$ is a decision variable such that $b_{ij} = 1$ if bid B_{ij} is accepted and 0 otherwise. Furthermore, $\mathcal{I}B_{ij}(g) = \sum_k \mathcal{I}_{ijk}$ and $\mathcal{O}B_{ij}(g) = \sum_k \mathcal{O}_{ijk}$.

1. Again, from each bidder at most one atomic bid can be accepted

$$\sum_j b_{ij} \leq 1$$

2. All the accepted transformations must together produce at least as much goods as required by the auctioneer

$$\sum_i \sum_j b_{ij}(\mathcal{O}B_{ij}(g) - \mathcal{I}B_{ij}(g)) \geq \mathcal{U}_{\text{out}}(g)$$

3. A solution must maximize

$$\sum_i \sum_j b_{ij} p_{ij}$$

In order to formulate the second problem, fix an arbitrary total ordering over the sets of transformations Σ that has been found as a solution to the previous problem. To model which transformation occurs where in the sequence, the variable $pos_i^m \in \{0,1\}$ is used. $pos_i^m = 1$ if transformation $t_i \in \Sigma$ occurs at position m in the sequence. Finally, transformation $t_i \in \Sigma$ is of the form $(\mathcal{I}_i, \mathcal{O}_i)$.

1. Each transformation in Σ is allowed to occupy exactly one position in the sequence
$$\sum_m pos_i^m = 1 \quad \forall i$$
2. Each position in the sequence must contain exactly one transformation
$$\sum_i pos_i^m = 1 \quad \forall m$$
3. The sequence must be a proper sequence
$$\mathcal{M}^m(g) = \mathcal{M}^{m-1}(g) + \sum_i pos_i^m(\mathcal{O}_i(g) - \mathcal{I}_i(g))$$
$$\mathcal{M}^{m-1}(g) \geq \sum_i pos_i^m \mathcal{I}_i(g)$$

In case the second problem fails to find a sequence, the first problem must be solved again. However, the previous solution is no longer a valid solution, because no sequence can be created from the transformations in this solution. Therefore, a new constraint must be added that disables the previous solution. A solution is in fact just an instantiation of a set of decision variables, i.e. it is a combination of zeros and ones. Interpreting the one as true and the zero as false, the link with propositional logic is easily found. Thus, an instantiation is nothing more than a propositional formula, and disabling a formula amounts to stating that its negation must be true.

A propositional formula can be translated into a linear equation as follows. The formula $p \vee q$ being true can be translated as $p + q \geq 1$ and $\neg p$ being true can be translated as $1 - p \geq 1$. Using these translations, the following equation can be used to disable a solution.

$$\sum_i \sum_j (1 - b_{ij}) \geq 1$$

As a final note, integer programming is used to model this solution because an integer programming solver can, in general, very quickly determine whether a set of equations has a solution. This property makes that it can rather efficiently be decided whether a sequence cannot be created.

4.4 Summary

The winner determination problem for mixed multi-unit combinatorial auctions appears more complicated than the winner determination problem for traditional combinatorial auctions. Both problems are NP-complete. However, if

one looks at the WDP for MMUCAs more closely, one can see that the problem consists of two separate problems. Namely, the problem of finding an optimal allocation, and the problem of finding a sequence given this allocation. Both problems are NP-complete on their own right.

Several solutions to the WDP for MMUCAs have been discussed. The first approach uses integer programming. The problem with this approach is that it uses a quadratic number of variables in terms of the number of input transformations. Two different approaches have been taken to reduce the number of variables used. The first is to switch from integer programing to constraint programming. The difference is the use of an indexed variable, taking away the need to define a decision variable for each possible position of each transformations.

The second approach is based on the subdivision of the WDP in two different problems. It uses the assumption that, on average, it will not take too many iterations before a solution is found. Which approach works best must be shown by experimentation.

Chapter 5

Implementation

An algorithm is not very useful if one is not able to implement it. Therefore, the purpose of this chapter is to provide implementations of the algorithms defined in Chapter 4.

In implementing an algorithm, several choices have to be made. The most important of these is the choice of the programming language. This choice influences the performance of the algorithm, the ease with which it can be implemented but also the readability of the code. Furthermore, in the case of the WDP one is interested in a comparison between the different algorithms. Since a comparison is less meaningful if two algorithms are not implemented using the same language, a language must be found that is able to implement all the different suggested algorithms.

The difficulty now is that two algorithms require an integer programming approach, while the other algorithm requires a constraint programming approach. Fortunately, these two methods can be elegantly combined using ECL^{i}PS^e[1] [Apt and Wallace, 2006].

Originally, ECL^{i}PS^e is a constraint solver (or actually a combination of constraint solvers), based on the programming language Prolog. The functionality of ECL^{i}PS^e can be extended by loading different libraries, one of which is the EPLEX library. The purpose of this EPLEX library is to provide an interface between ECL^{i}PS^e and several constraint solvers. The constraint solver used in this thesis is CPLEX 10.2[2], which is an industrial integer programming solver.

The rest of this Chapter is structured as follows. In Section 1 the ECL^{i}PS^e programming language is briefly introduced and Section 2 deals with the implementation of the three different algorithms.

5.1 ECL^{i}PS^e

ECL^{i}PS^e is an extension of Prolog [Bratko, 2001] that provides full-fledged support for constraint processing. For a general introduction to constraint programming using ECL^{i}PS^e, the reader is referred to [Apt and Wallace, 2006].

[1] `http://eclipse.crosscoreop.com/`
[2] `http://www.ilog.com/products/cplex/`

To explain the idea behind the ECL^{i}PS^e approach to programming, let us briefly recall Prolog.

5.1.1 Prolog

The programming language Prolog is an instance of Logic Programming. The basic constituents of a Prolog program are predicates. A predicate is of the form $Pred(A_1, \ldots, A_n)$, where $Pred$ is the name of the predicate and $A_1, \ldots, A_n$ are its arguments. A predicate can be either true or false, and hence predicates can be combined using the boolean operators $\vee$ and $\wedge$ (Note the absence of the boolean operator $\neg$). An example of a predicate is

$$male(bill)$$

a $\wedge$ combination of predicates is represented by

$$(male(bill), female(hillary))$$

and a $\vee$ combination of predicates is represented by

$$(president(hillary); president(obama))$$

Of course, more complicated, nested combinations are possible as well.

A Prolog program can consist of a set of predicates, also called facts. Prolog now uses this database of facts to obtain the truth value of a *query*, also called a *goal*. A query is a boolean combination of predicates. For example, look at the following set of predicates

$$male(bill)$$
$$female(hillary)$$
$$married(bill, hillary)$$

The interpretation of this database is that only the predicates present in it are true, everything else is false. In other words, Prolog uses *closed world* reasoning. Thus, Prolog will answer true to the following query

$$?: -male(bill)$$

but will answer false to the query

$$?: -husband(bill, hillary)$$

Although the traditional $\neg$ is not available in Prolog, there exists a different type of negation, represented by NOT. The interpretation of this NOT is linked with the closed world assumption used by Prolog. That is, $NOT(A)$ is true if the query $?: -A$ returns false. In other words, everything that is not logically derivable from the database is not true.

In the previous examples, all the arguments to the predicates where constants. However, in Prolog it is also allowed to use variables as arguments. For example, the results of the query $male(X)$ presented to the above database of facts will result in the answer $X = bill$. What happens is that Prolog tries to

match the query with the facts in the database. In case of the query $male(X)$, there is only one possible match, namely $male(bill)$. In order to obtain a perfect match, X must be instantiated to $bill$. This process is called matching, and has strong links with first order unification.

Using the variables and matching, one is now able to perform more complicated reasoning using the facts in the database. This can be done using so called clauses. A clause is of the form $Head :- Body$, where $Head$ is a simple predicate and $Body$ is a goal.

Suppose that the following facts are added to the database presented above

$$male(george)$$
$$female(laura)$$
$$married(george, laura)$$

Furthermore, suppose that one wants to add the notion of a husband. This can be done by adding the facts

$$husband(bill, hillary)$$
$$husband(george, laura)$$

However, these two facts are already contained in the database because we know that they are married and we know who is the male and who is the female. To extract this information, all that needs to be done is to define the "meaning" of the concept husband. This can be done using the notion of a clause. The husband clause now looks like

$$husband(X, Y) :- male(X), married(X, Y)$$

Suppose that we now supply Prolog with the query $? - husband(bill, hillary)$. What then happens is that Prolog matches $husband(bill, hillary)$ with $husband(X, Y)$, where $X = bill$ and $Y = hillary$. Next, the goal $husband(bill, hillary)$ is replaced by $male(bill), married(bill, hillary)$, both of which are facts in the database.

Besides constants and variables, there is one more important type of argument to a predicate, the list. A list is nothing more than what it is supposed to be, a list of arguments. These arguments can be either constants, variables or other lists. In Prolog, a list is represented as $[a_1, \ldots, a_n]$. To retrieve an element from a list, one can use the $[Head|Tail]$ construct. Prolog automatically matches $Head$ with the first element in the list, and $Tail$ with a list containing the remaining elements. Using this $[Head|Tail]$ construct, it is very straightforward to implement a clause that searches for an element in the list.

$$member(X, [X|Tail]).$$
$$member(X, [Y|Tail]) :- \quad member(X, Tail)$$

To see what this clause does, let's look at the query $? :- member(2, [1, 2, 3])$. First, Prolog tries to match the query with the first clause in the database, which fails. Second, it tries to match it to the second clause, instantiate X to 2, Y to 1

and $Tail$ to $[2, 3]$, replacing $member(2, [1, 2, 3])$ with the goal $member(2, [2, 3])$. Prolog will again try to match the new goal to some of the facts in the database. In this instance, the match with the first predicate is successful. Prolog will have no more goal to satisfy and returns true. From this one can conclude that 2 is a member of the list $[1, 2, 3]$.

5.1.2 ECL^{i}PS^e

The main features of ECL^{i}PS^e that are relevant for us are that variables can be declared and constraints can be imposed on them. For example, setting a range for a variable, say X, is done using the command

$$X :: [RANGE]$$

Furthermore, by loading different variables different solvers can be used.

Programming with Prolog requires a different mindset than programming with an imperative language. Also, working with datastructures like arrays it can be very cumbersome to work with Prolog. To that end, certain predicates have been added to ECL^{i}PS^e that allow a program to easily iterate through an array.

5.2 Algorithms

The goal of the different algorithms is to solve the winner determination problem for mixed multi-unit combinatorial auctions. To solve this problem, one must be able to represent auctions. A natural representation seems to be the array datastructure. Each algorithm uses three arrays, $Input$, $Output$ and $Prices$ to represent the auction. ECL^{i}PS^e contains some useful predicates to walk through these arrays in an imperative manner.

The algorithms as presented in Chapter 4 do not provide a way of solving the WDP. They merely provide a set of constraints, be they integer or other, that contain enough information to solve the constraints. The actual solving is performed by solvers. Therefore, the algorithms defined below do not provide a complete solution. They are just ways to represent the information needed by the solver to solve the problem. The only exception to this is the third algorithm, in which some work concerning the solving of the problem is done in the algorithm itself.

5.2.1 The first IP approach

The IP approach uses the EPLEX interface to talk with the CPLEX solver. In order to use this interface, the EPLEX library is loaded using the command

```
:- lib(eplex).
```

Furthermore, an eplex instance must be created using the command

```
:- eplex_instance(alg1).
```

Because the actual solving is handled by CPLEX, the purpose of the eplex code is to create the constraints that encode the WDP for the auction at hand. To that end, the clause solve() is defined

```
solve(Filename, UsedTime) :-
      readInput1(Filename, Input, Output, Price, Start, End,
      Information, Tm),
      cputime(StartTime),
      createConstraintArrays(Information, Tm, Xijkm, Xijk,
      Xij, Xm, Sequence),
      greaterOrEqualThanZero(Xijkm, Xijk, Xij, Xm),
      allIntegers(Xijkm, Xijk, Xij, Xm),
      fill(Xijkm, Xijk, Xij, Tm, Input),
      !,callSolver(Xijkm, Xijk, Xij, Xm, Input, Output, Price,
      Start, End, Sequence),
      cputime(EndTime),
      UsedTime is EndTime - StartTime.
```

The readInput1 predicate is used to obtain the auction information. The array *Input* stores the input information of each transformation and the array *Output* stores the output information of each transformation. The price of each bid is stored in *Price*, $\mathcal{U}_{\text{in}}$ is stored in *Start* and $\mathcal{U}_{\text{out}}$ is stored in *End*. *Information* contains information about the number of agents, the maximum number of bids per agent and the maximum number of transformations per bid. Finally, *Tm* represents the total number of transformations in the auction.

CreateConstraintArrays is used to create the arrays that are needed to create the constraints defining the WDP. *greaterOrEqualThanZero* and *allIntegers* define the constraints that make sure that every variable is a decision variable. That is, each variable can only contain the values 0 and 1.

Each constraint array is created by using the maximum number of bids per agent and the maximum number of transformations per bid. However, the actual number of bids or transformations can vary per agent. The predicate fill is used to fill in the variables that do not belong to an actual transformation or bid in such a way that they are not considered when generating the constraints.

Finally, the predicate *callSolver* is used to create the constraints that define the WDP and call on the CPLEX solver to find a solution. The full code can be found in Appendix A

5.2.2 The CP approach

The constraint programming approach uses the solver ic, which can handle interval constraints. The fd library could have been used as well, but the ic library has more options as to different search algorithms that are supported. To this end, the ic library is loaded using the command

```
:- lib(ic).
```

Furthermore, several constraints need to use a variable index to an array. To this end the library array_constraints_ic, created by Sebastian Brand [Brand, 2001], is loaded.

The main solve() clause now has the form

```
solve_ic(Filename, UsedTime) :-
        readInput2(Filename, Input, Output, Price, Start, End,
        Information, Nijk, T),
        cputime(StartTime),
        createArrays(Information, T, Xm, Xn, Bijk, Bij, Bm,
        Sequence),
        specifyDomains(T, Xm, Xn, Bijk, Bij, Bm),
        createConstraint1and2(Xm, Xn),
        createConstraintAllDiff(Xm, Xn),
        createConstraint3(T, Bijk, Xn),
        createConstraint4(Bij, Bijk, Nijk),
        createConstraint5(Bij),
        createConstraint6(T, Start, Bijk, Input, Output, Xm),
        createConstraint7(T, Start, End, Bijk, Input, Output,
        Xm),
        createConstraint9(T, Bm, Xm),
        createConstraint8(T, Bm),
        createConstraint10(Bijk, Bm, Xm),
        optimizeExpr(Bij, Price, Opt),
        {get_elements(Xm, XmE)},
        garbage_collect,
        bb_min(labelingTest(XmE, T, Constraints), Opt,
        bb_options{strategy:restart}),!,
        cputime(EndTime),
        UsedTime is EndTime - StartTime.
```

As in the previous algorithm, the data is obtained using the *readInput2* predicate and the constraint variables are created using *createArrays*. The domains of the variables are specified by using *specifyDomains* and the constraints are created using the predicates called *createConstraint**.

bb_min, an implementation of the branch and bound algorithm, is used to obtain a solution to the WDP. The interesting part now is the predicate *labelingTest*, which is used by *bb_min* to walk through the different possible instantiations of the constraint variables.

This predicate uses information about the structure of the auction to prune away different parts of the search tree. Remember that the final goal of the algorithm is to find a sequence of transformations and that to this end a set of dummy transformations have been introduced. The situation now is that, as soon as a dummy transformation is used in the "real" sequence, one can stop searching for other transformations. This is due to the fact that no gaps are allowed in the sequence that is found. The predicate labelingTest makes sure that bb_min stops if this situation occurs.

The complete code for the CP approach can be found in Appendix B.

5.2.3 The second IP approach

The final approach again uses the eplex library. The main solve predicate now takes the following form

```
solve(Filename, UsedTime) :-
        readInput1(Filename, Input, Output, Price, Start, End,
        Information, Tm),
        cputime(StartTime),
        createArrays(Information, Bij, InputB, OutputB),
        mergeInput(Input, Output, InputB, OutputB, Information),!,
        callSolver(Information, Bij, InputB, OutputB, Input,
        Output, Start, End, Price),
        cputime(EndTime),
        UsedTime is EndTime - StartTime.
```

The second IP approach differs in two aspects from the two previous approaches. Remember that in this approach the problem is divided into two smaller problems. The first is to find a set of transformations that satisfy certain properties. The second is to find a sequence using all these transformations.

The first problem required all the transformations in one bid to be added together. This functionality is provided by the *mergeInput* predicate. To solve the second part of the problem, the *callSolver* predicate calls the *findOptimalSequence* predicate. This predicate has the form

```
findOptimalSequence(Gm, Input, Output, Start, Bij, PassedTime) :-
        AvailableTime is 3600 - PassedTime
        cputime(CurrentTime),
        alg5:eplex_set(timeout, AvailableTime),
        alg5:eplex_solve(Cost),!,
        dim(Bij, I),
        dim(BijCopy, I),
        getBij(BijCopy, Bij),!,
        (
           cputime(CurrentTime2),
           PassedTime2 is PassedTime + (CurrentTime2 - CurrentTime),
           findSequence(Input, Output, Start, BijCopy, _Sequence,
           PassedTime2)
        ;
           writeln("The allocation is not proper"),!,
           setExclusion(BijCopy, Bij),
           alg5sequence:eplex_cleanup,
           cputime(CurrentTime2),
           PassedTime2 is PassedTime + (CurrentTime2 - CurrentTime),
           findOptimalSequence(Gm, Input, Output, Start, Bij,
           PassedTime2)
        ),!.
```

alg5 is the eplex instance that solves the first problem of finding a set of transformations that, together, produce the required amount of goods. The second eplex instance, alg5Sequence, is the instance that is associated with the problem of finding a sequence of transformations.

The complete code for the second IP approach can be found in Appendix C.

5.3 Summary

All the algorithms introduced in Chapter 4 have been implemented using ECL^{i}PS^e. ECL^{i}PS^e is a constraint solver based on Prolog, which is a Logic Programming language. The integer programming solver that is used is CPLEX 10.2.

Both IP approaches use the eplex library to talk to CPLEX. The CP approach uses the ic constraint solver, in combination with the array_constraints solver by S. Brandt. All algorithms use a branch and bound algorithm to find the optimal solution to the WDP.

The experimental evaluation of these algorithms is discussed in the next chapter.

Chapter 6

Experimental evaluation

The WDP of MMUCAs is, in the end, NP-complete. Therefore one will not be able to find an efficient algorithm that solves every auction in a reasonable amount of time. However, it is still interesting to see what the performance of an algorithm is on average for "realistic" auctions.

The purpose of this chapter is to compare the three approaches to the WDP of MMUCAs and determine which algorithm performs best in certain situations. Section 1 introduces the testset generators that have been used to generate the test suites. In Section 2, the experimental setup is discussed. Section 3 presents the results and finally, in Section 4 the results are discussed.

6.1 Test data

Although the WDP for MMUCAs is NP-complete, it is still very important to properly design a test suite to test the proposed methods of solving. This is due to the fact that one is not interested in the performance for all possible auctions. One is interested in the performance on real auctions, auctions that one might encounter in real life. Therefore, in designing a test suite one must take into consideration the situations in which an auction shall be held and performed.

In the case of normal combinatorial auctions, such a test suite already exists in the form of the CATS test suite[1]. In [Leyton-Brown *et al.*, 2000] several requirements are formulated that should be met by a combinatorial auction. These are

1. Certain goods are more likely to appear together than others.
2. The number of goods in a bundle is often related to which goods compose the bundle.
3. Valuations are related to which goods appear in the bundle, and, where appropriate, valuations can be configured to be subadditive, additive or superadditive in the number of goods requested.
4. Sets of XOR'ed bids are constructed in a meaningful way, on a per-bidder basis.

[1] `http://www.cs.ubc.ca/~kevinlb/CATS`

These requirements assume that the bidders bid on goods. Unfortunately, this is not the case in a MMUCA. However, the requirements can be used as an inspiration for the creation of a set of requirements for the MMUCA case.

Just as certain goods are more likely to appear together than others, some transformations are more likely to appear together than others. A factory that creates bicycle tires is likely to also produce car tires. However, it is not very likely that such a factory would be able to manufacture milk products. The two algorithms discussed below both have a different approach to this requirement.

The second requirement is that the number of goods which occur in a bundle are often related to which goods compose the bundle. The same holds for transformations, in that a transformation that produces one car has no need of more than four wheels, and a transformation that produces 5 cars should have as input 20 wheels. Thus, in the case of transformations the relationship is not so much between transformations, but between the input and output goods of one specific transformation.

The third requirement deals with the prices of bids. It states that the price of a bid is related to the goods that appear in the bid. In the case of MMUCAs the same can be stated. However, there is one extra layer to consider, namely the transformations. Each bidder has a valuation for each good that occurs in the auction. These goods are combined in transformations. Therefore, each agent also has a valuation for each transformation that occurs in the auction. It is reasonable to belief that the value for a transformation is determined by the values for the goods that occur in this transformation. Finally, bids are made up out of bundles of transformations. Thus, there should be a relation between the valuation of the bid and the valuations of the transformations that occur in the bid.

The last requirement is not so much a requirement on the auction, as it is a requirement on the generation process. It is linked with the first requirement in that, generating all the bids for one agent and at the same time honouring requirement one is easier if one generates bids on a per bidder basis.

The two generators discussed below are both based on the CATS requirements. However, they differ slightly in the requirements and in the types of transformations that they discern.

6.1.1 Generator 1

The first generator that is used, is taken from [Vinyals *et al.*, 2007]. Their analysis has resulted in three different types of transformations.

1. Output transformations
2. Input transformations
3. Input-Output transformations

The Output transformations are of the form $(\emptyset, \mathcal{O})$, the Input transformations are of the form $(\mathcal{I}, \emptyset)$ and the Input-Output transformations are of the form $(\mathcal{I}, \mathcal{O})$. The Input and Output transformations respectively denote the offer to buy and the offer to sell a good. The Input-Output transformations denote an exchange of goods, and can be used to model both assembly, transformation and trade.

The use of these three types of transformations has led the authors to state several requirements for MMUCAs. In their analysis they replace the notion of goods with the notion of market transformations. These market transformations are the basic building blocks of an auction, i.e. each transformation that occurs in an auction is a market transformation. Each market transformation is built up from a set of atomic transformations. An atomic transformation is a minimal transformation that can be performed. For example, a closet consists of some wooden boards and some nails. A transformation that takes as input a set of wooden boards and some nails, and outputs a closet is minimal. That is, one will not be able to cut the transformation up into smaller pieces and still obtain a meaningful transformation. The definitions of an atomic transformation and a market transformation, taken from [Vinyals *et al.*, 2007], are as follows.

Definition 30 (Atomic transformation) *Given a set of transformations* $T = \{t_1, \ldots, t_n\}$*, we say that transformation* $t_i = (\mathcal{I}_i, \mathcal{O}_i)$ *is minimum in* $\mathcal{T}$ *iff* $\forall t_j \in T$ *satisfying that* $\forall g_i \in \mathcal{I}_i, g_j \in \mathcal{I}_j\ g_i, g_j \in \mathcal{I} \cap \mathcal{I}'$ *and* $\forall g_i \in \mathcal{O}_i, g_j \in \mathcal{O}_j\ g_i, g_j \in \mathcal{O} \cap \mathcal{O}'$*, the following inequalities hold: (i)* $m_{\mathcal{I}_i}(g) \leq m_{\mathcal{I}_j}(g)\, \forall g \in \mathcal{I}_j$ *and; (ii)* $m_{\mathcal{O}_i}(g) \leq m_{|out_j}(g)\, \forall g \in \mathcal{O}_j$*.*

Definition 31 (Market transformation) *We say that* $T \subseteq \mathcal{N}^G \times \mathcal{N}^G$ *is a set of market transformations iff: (i) it is finite; (ii) every transformation* $t \in T$ *is minimum; and (iii)* $(\{g\}, \{\}), (\{\}, \{g\}) \in T \quad \forall g \in G$

Since a bid now consists of a combination of market transformations, the notion of transformation multiplicity must be introduced as an analogy to goods multiplicity. A bidder might offer to perform the transformation ($\{hub, dustcap, snapring\}, \{sprocket\}$) for a certain price. However, it might also offer to perform the transformation twice for less than twice the amount of one transformation. The multiplicity of the transformation in the second bid is two, whereas the multiplicity of the transformation in the first bid is one.

Definition 32 (Transformation multiplicity) *Let* $t = (\mathcal{I}, \mathcal{O})$ *and* $t' = (\mathcal{I}', \mathcal{O}')$ *be transformations such that* $\forall g \in \mathcal{I}, g' \in \mathcal{I}'\, g, g' \in \mathcal{I} \cap \mathcal{I}'$ *and* $\forall g \in \mathcal{O}, g' \in \mathcal{O}'\, g, g' \in \mathcal{O} \cap \mathcal{O}'$*. t has multiplicity* κ *with respect to* t' *iff* $m_{\mathcal{I}}(g) = \kappa \dot{m}_{\mathcal{I}'}(g) \forall g \in \mathcal{I}$ *and* $m_{\mathcal{O}}(g) = \kappa \dot{m}_{\mathcal{O}'}(g) \forall g \in \mathcal{O}$

All that is left is to discuss the requirements on the valuations of transformations. Suppose that one has all the goods to construct a bicycle, but not the knowledge to do this. If one goes to a bicycle shop, the mechanic will probably offer to you to build the bicycle for less than the price of a brand new bicycle. In fact, he will probably offer you to build the bicycle for the difference between the market price of the bicycle and the price of the goods. More generally, the price of a transformation should be determined by the difference between the valuation of the output goods compared to the valuation of the input goods.

The requirements discussed above, together with the CATS requirements are shown in Table 6.1

For information on the actual implementation of these requirements, the reader is referred to [Vinyals *et al.*, 2007].

1. There is a finite set of market transformations to bid for.
2. Certain transformations are more likely to appear together than others.
3. The number of transformations in a bundle is often related to which transformations appear in the bundle, and, where appropriate, valuations can be configured to be subadditive, additive or superadditive in the number of transformations requested.
4. Every transformation valuation is assessed in terms of the difference between the valuation of its output goods with respect to the valuation of its input goods.
5. sets of XOR'ed bids are constructed in a meaningful way, on a per-bidder basis.
6. Unrequested goods by the auctioneer may be involved in the auction.

Table 6.1: Requirements for the three transformation analysis

6.1.2 Generator 2

The previous generator made a division into different types of transformations, but did not take into consideration the fact that transformations sometimes are images of an underlying structure. Take as an example the construction of a bicycle. A bicycle consists of a pair of wheels, a frame, a handlebar, a chain and so on. A transformation that represents the construction of a bicycle might take as input the pair of wheels, the frame, the handlebar, the chain and outputs a bike. Such a transformation now represents the fact that all the goods in the input are contained in, or needed in the construction of the goods in the output. In other words, there is an underlying graph structure from which such transformations are taken.

It thus seems reasonable to define two types of Input-Output transformations. The first, called *structured transformations*, represent transformations that are taken from some *structured graph*. An edge in a structured graph between a and b represents the fact that a is contained in b. Therefore, a structured graph is not allowed to contain any cycles. More formally, its definition is as follows.

Definition 33 (Structured graph) *Given a set of goods G and a graph $\mathbb{G} = (G, E)$, where $E \subseteq G \times G$. $\mathbb{G}$ is a structured graph if it does not contain any cycles.*

The second type of transformations, called *unstructured transformations*, do not assume any structured relation between the goods in the input and the output. To complete the terminology, an auction containing only structured transformations is called a *structured auction*, an auction containing only unstructured transformations is called an *unstructured auction* and an auction containing both types of transformations is called a *hybrid auction*.

The structured transformations can be used to represent both assembly and dis-assembly transformations. The unstructured transformations can be used to represent trade transformations.

The requirement that certain transformations are more likely to appear together than other transformations is valid for the structured transformations, but not for the unstructured transformations. A bidder that offers a certain type of structured transformation usually is a specialist in this type of transformation. It would be odd for a car factory to offer to produce dinner tables. However, an agent that will want to trade something is usually interested in the valuation and not so much in the specific goods that are traded, although this of course does not need to be the case.

Introducing two types of transformations has as an effect that different types of goods can be discerned. The first two types of goods are the *structured goods* and the *unstructured goods*. Between the structured goods there exists some type of structured relation, whilst the unstructured goods have no specific relation whatsoever. The third type of good is the *tool*. In both assembly and disassembly tools can be used. These tools then occur both in the input and in the output, since they are used in the construction but do not take part in the final product.

The pricing requirement formulated for the previous generator also holds for the division in four different types of transformations. However, it only applies on the structured and unstructured goods. Since the valuation of a transformation should be dependent on the difference between the valuations of the input and the output goods, the valuation of a tool does not have any influence on the valuation of the transformation. However, one should be able to either buy or sell a tool, and therefore a valuation for a tool should be determined.

The requirements discussed above are summarized in Table 6.2

1. There is a finite set of structured transformations to bid for.
2. Certain structured transformations are more likely to appear together than others.
3. The number of structured transformations in a bundle is often related to which structured transformations appear in the bundle.
4. Where appropriate, transformation valuations can be configured to be subadditive, additive or superadditive in the number of transformations requested.
5. Every transformation valuation is assessed in terms of the difference between the valuation of its output goods with respect to the valuation of its input goods.
6. sets of XOR'ed bids are constructed in a meaningful way, on a per-bidder basis.
7. Unrequested goods by the auctioneer may be involved in the auction.

Table 6.2: Requirements for the four transformation analysis

Algorithm 1: MMUCA generator: generator(nAgents, sGoods, usGoods, tools, μ_{bids}, σ_{bids}, μ_{trans}, σ_{trans},μ_{output}, σ_{output},ratioI, ratioO, ratioTrans, ratioTrade, $\mu_{sellRatio}$, $\sigma_{sellRatio}$, maxPrice, maxMulti)

```
1  network ← createNetwork(sGoods, maxMulti);
2  prices ← prices(network, usGoods, maxPrice);
3  for i ← 1 to nAgents do
4      nBids ← ⌈N(μ_bids, σ_bids)⌉;
5      for j← 1 to nBids do
6          nTrans ← ⌈N(μ_trans, σ_trans)⌉;
7          for k← 1 to nTrans do
8              transType ← randomly select a type of transformation
               according to the given ratios;
9              switch transType do
10                 case input
11                     bid ← InputTrans(nGoods, μ_iRatio, σ_iRatio,
                       μ_Output, σ_Output, prices);
12                 end
13                 case output
14                     bid ← OutputTrans(nGoods, μ_sellRatio,
                       σ_sellRatio,μ_Output, σ_Output, prices);
15                 end
16                 case structured
17                     bid ← StructuredTrans(network, nGoods, μ_sellRatio,
                       σ_sellRatio,μ_Output, σ_Output, prices);
18                 end
19                 case unstructured
20                     bid ← UnstructuredTrans(nGoods, μ_sellRatio,
                       σ_sellRatio,μ_Output, σ_Output, prices);
21                 end
22             end
23         end
24     end
25 end
```

The algorithm

The algorithm discussed below is meant to generate an instance of a MMUCA WDP. A generated WDP consists of a set of XOR bids in such a way that it is ready to be fed into the solving methods discussed above. On top of that, the algorithm is designed in such a way that the WDP it produces satisfies the requirements from Table 6.2. The algorithm is implemented in MATLAB, and the code can be obtained via the author.

To this end, first the number of different goods must be determined. That is, the number of structured and unstructured goods and the number of tools must be defined. Second, using the structured goods a structured graph must be generated. After that, the prices of all the goods must be determined. Note that, although the tools are not used, they must be priced due to the fact that one should be able to buy or sell tools. Finally, the bids are generated on a per

Algorithm 2: createNetwork(sGoods, maxMulti)

```
1  nLevels ← rand(nGoods);
2  initialize levels to an array of size nLevels ;
3  for i ← 1 to nLevels-1 do
4      levels(i)← the goods that occur on this level;
5  end
6  level ← 1;
7  for i ← 1 to nGoods minus the number of goods in the highest level do
8      if i is the first good on level level +1 then
9          level ←level + 1
10     end
11     possibleConnections ← the number of goods above i;
12     for j ← 1 to possibleConnections do
13         Randomly determine whether i is connected to the j^th object
           above it
14     end
15 end
```

agent basis and the transformations are generated on a per bid basis.

Generating the structured network

A structured graph is a directed graph that does not contain any cycles. Therefore, moving from one node to another node in the graph can be seen as moving deeper into the graph. Furthermore, due to the lack of cycles in the graph there must be at least one node that has no incoming edges. One can thus say that at least one node lies on level 1, some of the nodes reachable from this first node lie on level 2 and so on. Actually, a node can be reachable from several different "lower" nodes. In this case, the node in question lies on a level that is higher than the "highest" node from which it is reachable. This view leads to the graph generating algorithm displayed in Algorithm 2.

First, the number of levels is determined according to the number of goods that must be contained in the graph. Second, each good is positioned on some level in the network. The last step in generating the graph is to determine the edges, where each node can only be connected to nodes that lie on a higher level.

After the graph has been generated, all that is left is to determine which tools are needed in which transformation.

Generating the $\mathcal{U}_{\text{in}}$ and $\mathcal{U}_{\text{out}}$

The generation procedure for $\mathcal{U}_{\text{in}}$ and $\mathcal{U}_{\text{out}}$ is identical. The parameters **pGoodNeeded** and **pGoodRequested** represent respectively the chance that a good occurs in $\mathcal{U}_{\text{in}}$ and the chance that a good occurs in $\mathcal{U}_{\text{out}}$. In generating the data, it is determined which good occurs in it on a per good basis.

Generating the bids

The generation of bids is done on a per bidder bases. The number of bidders is determined beforehand. Then, for each bidder the number of bids is drawn from a normal distribution with mean μ_{bids} and standard deviation σ_{bids}. After that, the transformations are generated on a per bid basis. Per bid, the number of transformations is drawn from a normal distribution with mean μ_{trans} and standard deviation σ_{trans}.

The first step in generating a transformation is determining what the type of the transformation is. This is done by using an occurrence ratio for each of the different types. The actual type is than drawn from a uniform distribution between 1 and the total number of occurrences (the sum of all occurrence ratios). The next step is influenced by the type of transformation that needs to be generated. In the case of either an Input or an Output transformation, for each good it is determined whether it is participating in the auction using the **iRatio** argument. The **iRatio** is drawn from a normal distribution with median $\frac{1}{numberofgoods}$ and standard deviation 0.1. These parameters are chosen in such a way that the number of goods per transformation does not become too large. In generating the transformation, it is made sure that it contains at least one good to prevent empty transformations.

In the case of a structured transformation, the number of output goods is generated using the **sellRatio**, which is drawn from a normal distribution with medium $\mu_{sellRatio}$ and standard deviation $\sigma_{sellRatio}$. However, only structured goods can be selected. After the output goods have been defined, the number of input goods is determined via the structured graph, together with the number of tools.

In the case of an unstructured transformation both the input and the output goods are generated using **sellRatio**.

After all the transformations belonging to one bid are generated, the price of this bid must be determined. This is done by looking at the difference in value of the input and the output goods per transformation.

6.2 Experimental setup

All the experiments are performed on a Intel Core 2 Duo machine with 2 GB of memory running linux. Two different tests are performed. The first uses a testset generated using the generator described in Section 6.1.1, the second uses a testset generated using the generator described in Section 6.1.2. In both tests, only the results on feasible auctions are taken into considerations. This is because CPLEX can very efficiently detect whether a problem is feasible or not.

6.2.1 Test 1

In [Vinyals *et al.*, 2007] the algorithm presented in Section 4.3.1 has been tested using a dataset generated with generator 1. Due to the fact that the number of constraints increases with the number of transformations, they conjectured that the more transformations appeared in an auction, the harder it would be to solve it.

To test this assumption four different datasets where generated, using four different situations. In the first situation each agent makes only one bid and each bid contains only one transformation. In the second situation each bid contains two transformations while each agent is allowed to submit only one bid. The third situation requires an agent to submit two bids, both containing one transformation. Finally, in the fourth situation each agent submits two bids, both containing two transformations.

For each situation the following types of auctions where generated.

- 30 instances of 40 transformations.
- 30 instances of 80 transformations.
- 30 instances of 120 transformations.
- 30 instances of 160 transformations.
- 30 instances of 200 transformations.

The parameters that where used to generate the auctions are:

- nGoods = 4.
- nMarketIOTransformations = 10.
- maxPrice = 100.
- varianceOfprices = 0.05.
- pGoodRequested = 0.3.
- varAddNewXORClause = 0.
- sigmaTrnasfBid = 0.
- pGoodInOutput = 0.3.
- pGoodInInput = 0.1.
- alpha = 0.1.
- pITransformations = 0.2.
- pOTransformations = 0.2.
- cycles = true.

In order to be able to compare the results in [Vinyals *et al.*, 2007] with the experiments displayed in this thesis, the same tests are performed. To this end the generator defined in [Vinyals *et al.*, 2007] has been used to generate a testset with the parameters described above. All three algorithms described in Chapter 4 are tested by measuring the total cpu-time used to solve each auction. For each situation and each type the average runtime is calculated. In running the experiments, a time limit has been set of 3600 minutes. Finally, of each auction 30 instances are generated.

6.2.2 Test 2

The generator used in the previous test was based on a three way division in types of transformations. The motivation behind the second test is to see if the differentiation in four different types of transformations has an influence on the performance of the algorithms. That is, it is interesting to see if there is a performance difference between auctions that only contain structured transformations, auctions that only contain unstructured transformations and auctions that contain both types of transformations. Input and Output transformations are allowed in all three scenarios.

To that end three different testsets have been generated. Each testset contains the same four situations as described above in terms of numbers of bids and the way the transformations are divided among the agents. However, only auctions with respectively 40, 80 and 120 transformations are generated. The difference between the testsets is in the types of transformations that occur. All the testsets contain input and output transformations. However, Testset 1 contains only structured transformations, Testset 2 only unstructured transformations and Testset 3 a mixture of both. In generating the testsets, 10 goods have been used. Testset 1 was generated using 8 structured goods and 2 tools. Testset 2 was generated using 10 unstructured goods and Testset 3 was generated using 4 structured and 4 unstructured goods and 2 tools. The number 10 has been chosen such that there are enough goods to create a non trivial structured graph while not having too many goods. This last to make the testing still feasible.

In generating all testsets the following parameters have been used, and each type of auction had 5 instances.

- maxMulti = 5
- mOutput = 5
- sOutput = 1
- mSellRatio = 0.6
- sSellRatio = 0.1
- maxPrice = 100
- sBids = 0
- sTrans = 0

Again the measure of performance is the cpu-time, and a time limit has been set to 3600 minutes. Of each auction 10 instances are generated

6.3 Results

The tests have been performed using two different datasets. The the results of Test 1 can be found in Figures 6.1 through 6.3. The results of Test 2 can be found in Figures 6.4 through 6.9. Furthermore, a more detailed graph of Figure 6.7 can be found in 6.10.

The x-axis shows the average cpu-time in minutes needed to obtain a solution to the problem. Do note that the time limit prohibits any average to reach above the 3600 minutes. Therefore, any average that reaches this line only tells us that the program needs 3600 minutes or more to solve the problem. The y-axis shows the number of transformations per auction.

6.4 Discussion

6.4.1 Test 1

The results belonging to the first IP approach do not correspond with the results presented in [Vinyals *et al.*, 2007]. Where in the results presented in this thesis the first IP approach performs worst on situation 2, in [Vinyals *et al.*, 2007] situation 3 appears to be the most complex. The greatest difference with the approach taken by Vinyals *et al.* is that they represent the auctions in OPL Studio 5.0. In feeding these auctions to CPLEX, a translation must be made which might account for the discrepancy.

The first observation that can be made from both Figure 6.1 and Figure 6.3 is that the solving times increases when the number of transformations increases (except for one situation in Figure 6.3). Thus the increase in the number of variables has a clear influence on the complexity of the problem.

The results obtained from the CP approach show that, although the constraint programming approach reduces the amount of variables that are needed, it is inferior to the IP approaches. Using the profiler predicate available in ECL^{i}PS^e, one can see that this is mainly due to the constraints dealing with the variable index. Using such a variable index was the source of reduction in terms of the number of variables needed. Unfortunately, the constraints that are needed to deal with these types of constraints are very expensive. As mentioned in [Brand, 2001], this is due to the nature of the constraint and not so much due to the nature of the implementation.

Using Figure 6.1 and 6.3, the different situations can be ranked with respect to how hard it is for both approaches to solve the auctions. This results in the ranking shown in Table 6.3

rank	first IP approach		second IP approach	
	bids	trans	bids	trans
1	2	2	2	1
2	2	1	1	1
3	1	1	2	2
4	1	2	1	2

Table 6.3: complexity ranking of the different situations based on Test 1

Looking at Table 6.3 more closely, two observations can be made. The first is that it appears to be easier to solve an auction where each agent submits 2 bids than an auction where each agent submits 1 bid. This might have to do with the fact that the number of different allocations decreases when the number of bids per agent increases (keeping the number of transformations constant).

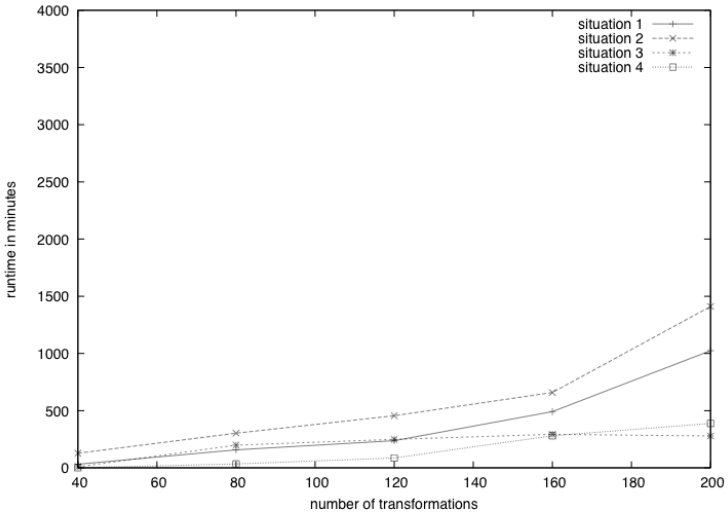


Figure 6.1: The results for the first IP approach

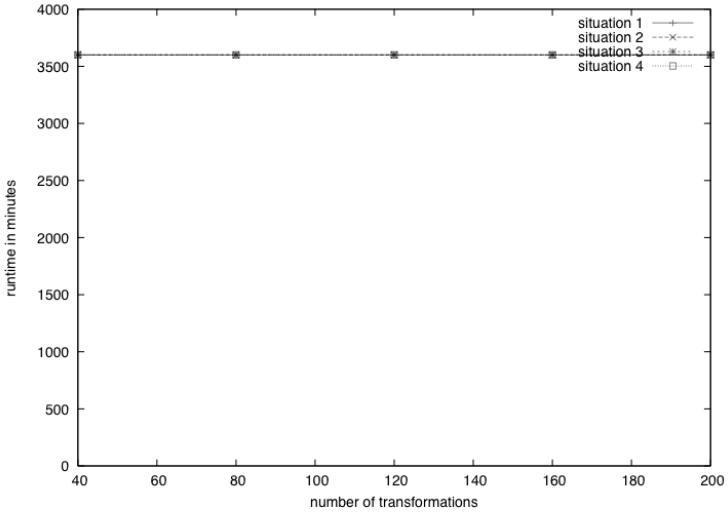


Figure 6.2: The results for the CP approach

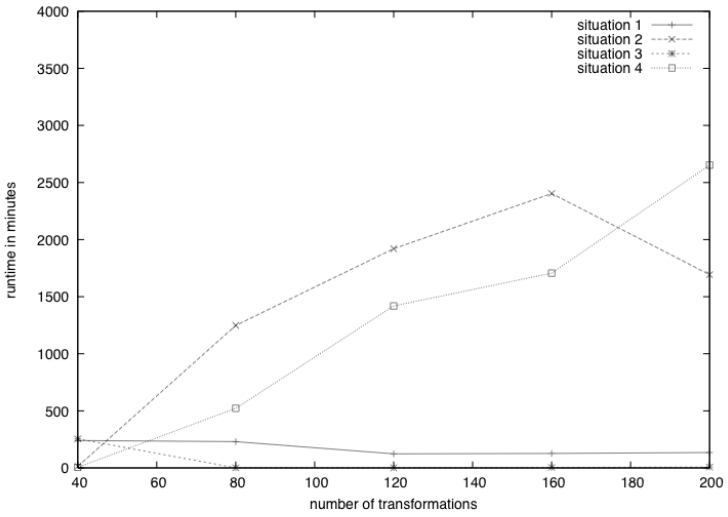


Figure 6.3: The results for the second IP approach

The influence of the number of transformations does not manifest itself in the results obtained from the first IP approach. However, looking at the results from the second IP approach it is clear that auctions containing two transformations are harder than auctions that contain only one transformation. In fact, from Figure 6.3 it can be observed that the running times appear to be almost constant in the case of only one transformation. Remember that the second IP approach is designed in such a way that an auction becomes more easy to solve if it does not contain any cycles. It could thus be the case that auctions with only one transformation per bid are less likely to contain cycles. The experiments performed in Test 2 might strengthen this possibility, since an auction containing only structured transformations does not contain any cycles.

With the previous result in mind, it would be very interesting to see how many cycles an auction actually contains. This could be done by first finding all the cycles in the goods graph using the Floyd-Warshall algorithm. The next step is to identify the transformations that are responsible for the cycles. Comparing the total number of transformations participating in a cycle with the complexity of the auction might show whether the higher the number of transformations, the harder the auction. Future work should be directed into this direction.

In comparing the performance of the first and the second IP approach, the first approach is more consistent. That is, a small increase in complexity means a small increase in running time in the first IP approach. In the second IP approach, the running times fluctuate more. Thus, overall the first IP approach has a better performance on the testset generated by generator 1. The only peculiarity is that for the second situation the average running time of the second IP approach for an auction consisting of 200 nodes drops significantly. It could be possible that this is just a peculiarity of the dataset that has been used. Future work should shed more light on this question.

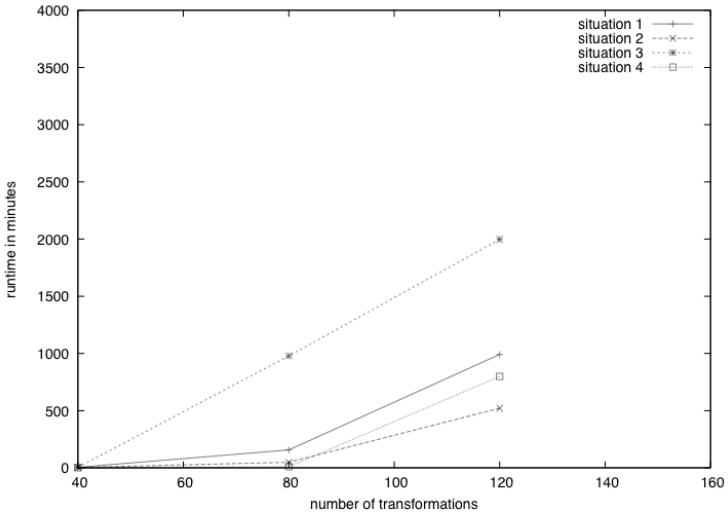


Figure 6.4: First IP approach: Structured transformations

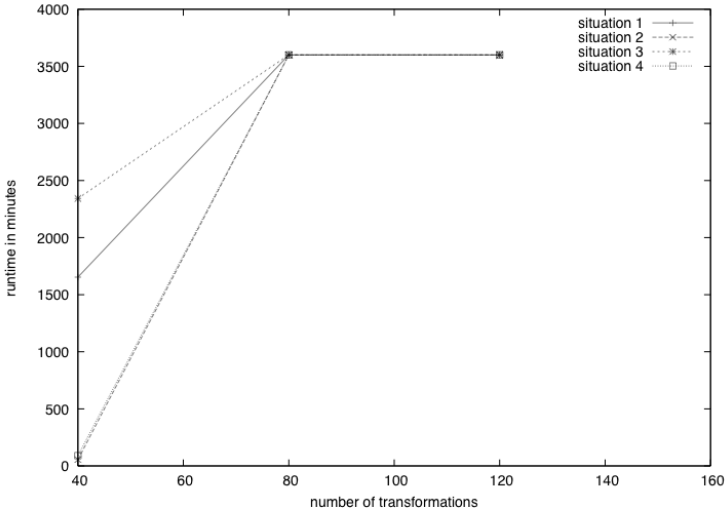


Figure 6.5: First IP approach: Unstructured transformations

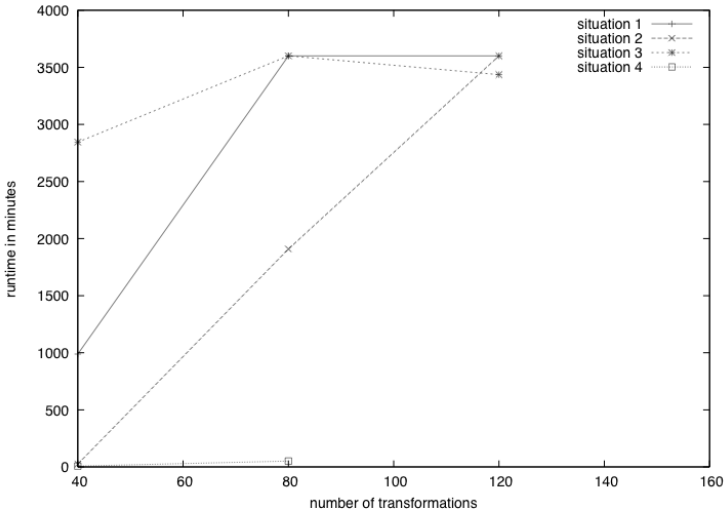


Figure 6.6: First IP approach: Both structured and unstructured transformations

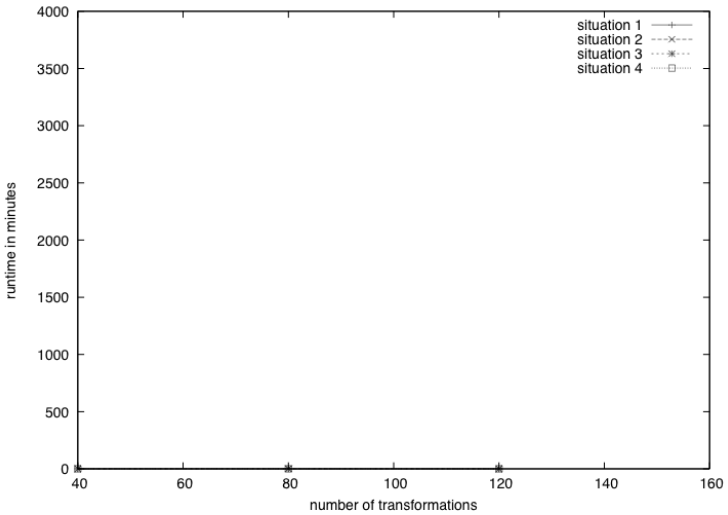


Figure 6.7: Second IP approach: Structured transformations

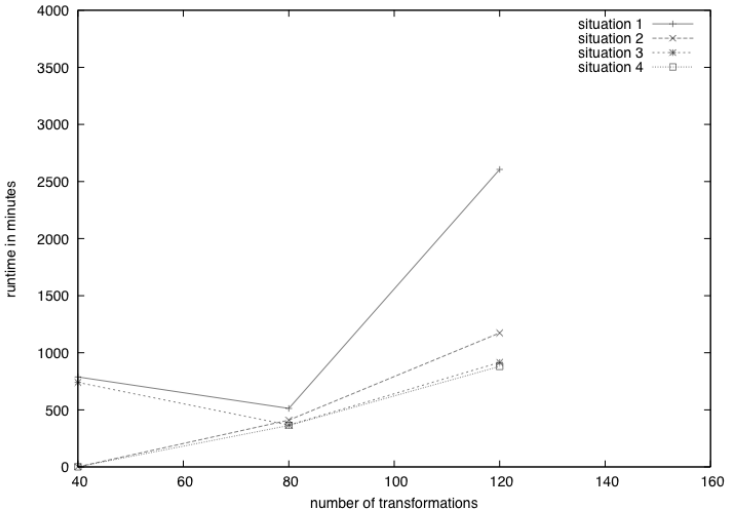


Figure 6.8: Second IP approach: Unstructured transformations

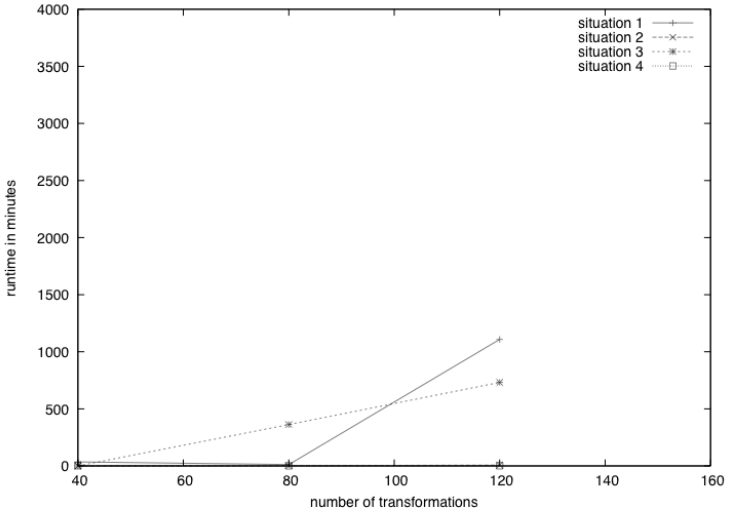


Figure 6.9: Second IP approach: Both structured and unstructured transformations

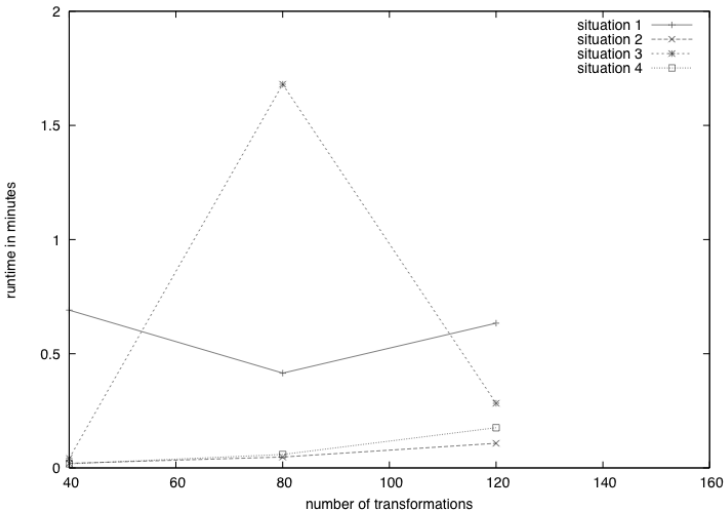


Figure 6.10: Second IP approach: structured transformations

6.4.2 Test 2

Before discussing the results, it must first be stated that the performance of the CP approach was again no match for the IP approaches. Because the results of these tests do not provide any useful information other than that the performance is bad they have been omitted.

The second tests have been performed to see what the influence of the different types of Input-Output transformations are on the running time of the algorithms. Looking at the difference in running time between the auctions with only structured transformations and the auctions containing only unstructured transformations, one can see that the performance of the latter is much better. This is as was expected. That is, it has been shown that for an auction with an acyclic goods graph it will always be possible to find a sequence. If an auction only contains structured transformations, the resulting goods graph will be a subgraph of the structured graph. Since a structured graph is designed not to have any cycles, such auctions should be easier to solve.

Furthermore, one can see that the second IP approach performs much better than the first IP approach on structured auctions. See 6.10 for a more detailed picture of the results of the second IP approach. The fluctuations in Figure 6.10 can be explained from the fact that it is either very fast, or it needs many iterations.

One explanation for better performance of the second IP approach is that it benefits from the lack of cycles. That is, since the second part of the problem is always possible if the goods graph does not contain any cycles, the first problem has to be solved only once. Because the first problem contains fewer variables and constraints than the complete WDP while remaining in the IP formalism, it is easier to solve. This is thus supported by the experimental results.

The performance on auctions that contain both types of Input-Output transformations tends to lie in between. This is the same for both approaches. This

could be an indication that having fewer unstructured transformations is likely to reduce the number of cycles and hence also the complexity of the problem at hand. The fact that the performance of the second IP approach is almost constant also points into this direction.

Interestingly, in comparing the performance of both algorithms, the second IP approach appears to perform much better than the first IP approach in all cases. This indicates that in such situations cycles are less likely to occur. Thus, it seems that in situations where both trade and construction/deconstruction occur the second IP approach would perform best.

	rank	The first IP approach		The second IP approach	
		bids	trans	bids	trans
structured	1	1	2	2	2
	2	2	2	1	2
	3	1	1	2	1
	4	2	1	1	1
unstructured	1	1	2	2	2
	2	2	2	2	1
	3	1	1	1	2
	4	2	1	1	1
hybrid	1	2	2	2	2
	2	1	2	1	2
	3	1	1	2	1
	4	2	1	1	1

Table 6.4: complexity ranking of the different situations based on Test 2

Again, based on the results a complexity ranking can be made for the different situations. This ranking is displayed in Table 6.4. From the ranking it follows that the more transformations in a bid, the easier it is to solve an auction. This could be explained from the fact that increasing the number of transformations per bid makes the first problem, the problem of finding an allocation that satisfies Equation 4.3 easier. That is, in finding such an allocation one is not interested in particular transformations, but in the total number of transformations in a bid. Increasing the number of transformations per bid, while keeping the total number of transformations equal means that the total number of possible allocations decreases. i.e. the search space is reduced.

In comparing the results from Test 2 with the results from Test 1, there is one big discrepancy. In Test 1 more transformations per bid means slower performance, while in Test 2 more transformations per bid means better performance. Since the algorithms used in Test 1 and Test 2 are the same, the cause of this discrepancy must lie in the testset generation. More tests are needed to get a better understanding of this discrepancy.

6.5 Summary

Testing an algorithm is of utmost importance, because it can give an indication of its applicability. Although the problem at hand is NP-complete, testing still

is important since it could be that in real situations a solution still performs well.

To test the three solutions to the WDP for MMUCAs two different tests have been performed. The first with a dataset generated by the generator described in [Vinyals *et al.*, 2007]. Here a distinction is made between Input, Output and Input-Output transformations. The second test is performed with a testset in which two different types of Input-Output transformations are discerned, namely structured and unstructured transformations. The purpose of the second pair of tests has been to see what the influence of these two types of transformations is on the complexity of the WDP.

From the first test, it can be concluded that the CP approach is no match for both IP approaches. In comparing both IP approaches, the first performs more consistently, but on some fronts the second approach is much better. Furthermore, one can notice that increasing the number of bids reduces the complexity of the auctions. This is probably due to the fact that this reduces the number of possible allocations. Second, the second IP approach performs better in the case of one transformation per bid where in the second test the reverse is found. The reason for this probably lies in the dataset generation, and more work must be done to analyze this discrepancy further.

Furthermore, the second set of tests shows that auctions containing only structured transformations are by far the easiest to solve. Auctions containing only unstructured transformations are much harder to solve, while lowering the number of unstructured transformations reduces the complexity. Finally, the second IP approach appears to be performing better than the first IP approach on all three types of auctions.

Chapter 7

Conclusions

The purpose of this thesis is to learn more about the WDP for MMUCAs. More specifically, this thesis looks at the WDP from an experimental viewpoint.

To this end, in Chapter 4 three different approaches to the solving of the WDP are introduced. The first approach that is introduced models the WDP using integer constraints in such a way that an integer programming solver can be used. This is the standard approach taken in combinatorial auctions. The drawback of this approach, however, is that it needs a quadratic number of variables. This is due to the fact that, in integer programming, one cannot use a variable index to an array. Therefore, each possible entry for each point in the sequence needs an own variable.

To remedy this, a constraint programming approach is introduced. This second approach uses fewer variables, but makes use of a more expensive constraint. However, it is hoped that the reduction of variables is such that the solving power of this approach is better than the first.

Finally, a third approach is introduced. This approach is based on the observation that the WDP can be seen as the continued solving of two different problems. First, the problem of finding an allocation of bids such that the combined set of transformations produces at least the number of required goods. Second, given an allocation a sequence must be found. One can show that if the goods graph of an allocation does not contain any cycles, finding a sequence becomes trivial and will always be possible.

To test the different algorithms two different data generators are used. One, due to [Vinyals *et al.*, 2007] is used to compare the algorithms with existing tests. This data generator discerns three different types of transformations, Input, Output and Input-Output transformations. The second data generator is based on the observation that Input-Output transformations can be subdivided in two types of transformations, structured and unstructured transformations.

The first observation that one can make from the test results obtained in Test 1, is that the performance of the constraint programming approach is very bad, compared to the other two approaches. Although the number of variables is reduced, close examination of the program shows that the variable index constraints are very expensive.

Second, one can observe that increasing the number of bids per agent appears

to reduce the complexity of the auction. Increasing the number of transformations per bid does not have any influence on the first IP approach, but it does have an effect on the performance of the second IP approach.

From the results obtained in Test 2, it can be observed that the performance on structured auctions is much better than the performance on unstructured auctions. The cause of this is most likely to be the lack of cycles in the latter type of auction. Furthermore, hybrid auctions appear to be less complex than auctions containing only unstructured transformations. Again this is due to the fact that cycles are less likely to occur in these types of auctions.

The results in this thesis have resulted in several questions that need answering. First, it would be interesting to further investigate the influence of cycles on the complexity of the WDP. It might be that there are other types of cycles or combinations of cycles that still enable one to find a sequence relatively fast. One can think, for example, about what happens when there is no interaction between different cycles. It is likely that this reduces the complexity of the WDP.

Second, it would be interesting to look further at the discrepancies between Test 1 and Test 2. Although some suggestions for explanations have been given, more testing is needed to find the exact cause of the discrepancy.

Finally, the testset generator described in this thesis can still be enhanced. This could be done by improving the generation of transformations by adjusting the probability distributions that are used to decide which goods are used, how many transformations occur in a bid and how many bids each agent makes.

Appendix A

Implementation of the first IC approach

```
:- use_module("../read_input2").
:- lib(eplex).
:- eplex_instance(alg1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% General Methods
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% solve(+Name)
%
% This is the main predicate. It gets as inpute a filename, opens this file,
% reads in the bids and calls the solver
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

solve(Name, UsedTime) :-
readInput1(Name, Input, Output, Price, Start, End, Information, Tm),
        cputime(StartTime),
createConstraintArrays(Information, Tm, Xijkm, Xijk, Xijk2, Xij, Xij2,
        Xm, Sequence),
greaterOrEqualThanZero(Xijkm, Xijk, Xij, Xm),
allIntegers(Xijkm, Xijk, Xij, Xm),
fill(Xijkm, Xijk, Xijk2, Xij, Xij2, Tm, Input),
!,callSolver(Xijkm, Xijk, Xijk2, Xij, Xm, Input, Output, Price, Start,
        End, Sequence),
        cputime(EndTime),
```

```
        UsedTime is EndTime - StartTime.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% callSolver(+Xijkm, +Xijk, +Xij, +Xm, +Input, +Output, +Price, +Start, +End,
%  +Sequence)
%
% This predicate first creates all the constraints. After that it creates the
% formula that needs to be maximized and it calls the solver. In the mean time
% it measures the time that is needed to set op the constraints and solve them.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


callSolver(Xijkm, Xijk, Xijk2, Xij, Xm, Input, Output, Price, Start, End,
           Sequence) :-
setConstraint1(Xijk, Xijk2, Xij),!,
setConstraint2(Xij),!,
setConstraint3(Xijkm, Xijk, Xijk2),!,
setConstraint4(Xijkm, Xm),!,
setConstraint5(Xm),!,
setConstraint6(Input, Output, Start, Xijkm),
setConstraint7(Input, Output, Start, End, Xijkm),
optimizeEx(Xij, Price, Opt),
alg1: eplex_solver_setup(max(Opt), _, [method(primal), timeout(3600),
                                abort_handler(aborted(halted))],
                                0, []),!,
alg1: eplex_solve(Profit),
writeln("solved").

% this predicate catches the situations in which the solver aborts for some
% reason
aborted(halted).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Constraint Predicates
%
% The following predicates set the constraints that are defined in:
% "Bidding Languages and Winner Determination for Mixed Multi-unit
%  Combinatorial Auctions"
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



setConstraint1(Xijk, Xijk2, Xij) :-
(
    foreachelem(El, Xijk2, [I,J,K]),
    param(Xijk, Xij)
```

```
do
            (
El == 1,
alg1: (Xijk[I,J,K] $= Xij[I,J])
    ;
        El == 0
   )
).

setConstraint2(Xij) :-
dim(Xij, [A, Mb]),
(
    count(I, 1, A),
    param(Xij, Mb)
do
            Temp is Xij[I,1..Mb],
    createSum(Temp, Sum),
    constraint2(Sum)
).

% this is done to cope with the situation where the sum consists of a single
% variable

constraint2([Sum]) :-
alg1: (Sum $=< 1).

setConstraint3(Xijkm, Xijk, Xijk2) :-
dim(Xijkm, [_,_,_,M]),
(
    foreachelem(El, Xijk2, [I,J,K]),
    param(Xijkm, Xijk, M)
do
            (
El == 1,
Temp is Xijkm[I,J,K,1..M],
createSum(Temp, Sum),!,
constraint3(Xijk, I,J,K,Sum)
    ;
       El == 0
   )
).

% this is done to cope with the situation where the sum consists of a single
% variable

constraint3(Xijk, I,J,K, [Sum]) :-
alg1: (Xijk[I,J,K] $= Sum).

setConstraint4(Xijkm, Xm) :-
dim(Xijkm, [I,J,K,_]),
```

```
(
    foreachelem(_, Xm, [M]),
    param(Xijkm, Xm, I,J,K)
do
    Temp is Xijkm[1..I, 1..J, 1..K,M],
    flatten(Temp, Temp2),
    createSum(Temp2, Sum),!,
    constraint4(Xm, M, Sum)
).

% this is done to cope with the situation where the sum consists of a single
% variable

constraint4(Xm, M, [Sum]) :-
alg1: (Xm[M] $= Sum).

setConstraint5(Xm) :-
dim(Xm, [M1]),
M is M1-1,
(
    count(I, 1, M),
    param(Xm)
do
            I2 is I+1,
    alg1: (Xm[I2] $=< Xm[I])
    %write([I2]), write(" $=< "), writeln([I])
).

setConstraint6(Input, Output, Start, Xijkm) :-
dim(Xijkm, [I,J,K,M1]),
dim(Input, [I,J,K,G1]),
(
    count(M, 1, M1),
    param(Input, Output, Start, Xijkm, G1)
do
            (
count(G, 1, G1),
param(Input, Output, Start, Xijkm, M)
    do
Mminus is M - 1,
multiset(Start, Input, Output, Xijkm, Mminus, G, Mset),!,
productSum(Xijkm, Input, M, G, Mset)
    )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% productSum(+Xijkm, +Input, +M, +G, +Mset)
%
% The purpose of this predicate is to calculate the sum \sum_{ijk} x^{m}_{ijk}
```

```
% *\In_{ijk}(g)
% for some g and some m. After that it sets the
% inequality Mset >= Sum.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


productSum(Xijkm, Input, M, G, Mset) :-
dim(Xijkm, [Imin,Jmin,Kmin,_]),
Im is Imin + 1,
Jm is Jmin + 1,
Km is Kmin + 1,
(
    fromto([1,0], In, Out, [Im,Sum]),
    param(Xijkm, Input, M, G, Jm, Km)
do
            In = [I, SumI_old],!,
    (
fromto([1,0], In, Out, [Jm, SumIJ]),
param(Xijkm, Input, M, G, I,Km)
    do
        In = [J, SumIJ_old],!,
(
    fromto([1,0], In, Out, [Km, SumIJK]),
    param(Xijkm, Input, M, G, I, J)
do
            In = [K, SumIJK_old],!,
    Temp is Input[I,J,K,G],
    ( Temp == void ->
SumIJK_new = SumIJK_old
    ;
        SumIJK_new = Xijkm[I,J,K,M] * Input[I,J,K,G]
                                    + SumIJK_old
    ),
    Knew is K + 1,
    Out = [Knew, SumIJK_new]
),
SumIJ_new = SumIJK + SumIJ_old,
Jnew is J + 1,
Out = [Jnew, SumIJ_new]
    ),
    SumI_new = SumIJ + SumI_old,
    Inew is I + 1,
    Out = [Inew, SumI_new]
),
alg1: (Mset $>= Sum).

setConstraint7(Input, Output, Start, End, Xijkm) :-
dim(Xijkm, [_, _, _,T]),
(
```

```
    foreachelem(_, End, [G]),
    param(Input, Output, Start, End, Xijkm, T)
do
            multiset(Start, Input, Output, Xijkm, T, G, Mset),!,
    alg1: (Mset $>= End[G])
).


multiset(Start, Input, Output, Xijkm, M, G, Mset) :-
doubleSum(Xijkm,Input, Output, M, G, DS),
Uin is Start[G],
Mset = Uin + DS.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% doubleSum(+Xijkm, +Input, +Output, +Mmin, +G, -Mset)
%
% The purpose of this predicate is to calculate the sum
% \sum_{l=1}^{m}\sum_{ijk} x^{l}_{ijk} * (\Out_{ijk}(g) - \In_{ijk}(g))
% which is saved in Mset
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


doubleSum(Xijkm, Input, Output, Mmin, G, Mset) :-
dim(Xijkm, [Imin, Jmin, Kmin, _]),
M is Mmin + 1,
Im is Imin + 1,
Jm is Jmin + 1,
Km is Kmin + 1,
(
    fromto([1,0], In, Out, [M,Mset]),
    param(Xijkm, Input, Output, G, Im, Jm, Km)
do
            In = [L, Mset_old],!,
    (
fromto([1,0], In, Out, [Im,MsetI]),
param(Xijkm, Input, Output, G, L, Jm, Km)
    do
        In = [I, MsetI_old],!,
(
    fromto([1,0], In, Out, [Jm, MsetIJ]),
    param(Xijkm, Input, Output, L, I, G, Km)
do
            In = [J, MsetIJ_old],!,
    (
fromto([1,0], In, Out, [Km, MsetIJK]),
param(Xijkm, Input, Output, G, L, I, J)
    do
        In = [K, MsetIJK_old],!,
```

```
Temp is Input[I,J,K,G],
( Temp == void ->
    MsetIJK_new = MsetIJK_old
;
    MsetIJK_new = MsetIJK_old + Xijkm[I,J,K,L]
                                        *(Output[I,J,K,G] - Input[I,J,K,G])
),
Knew is K + 1,
Out = [Knew, MsetIJK_new]
    ),
    MsetIJ_new = MsetIJ_old + MsetIJK,
    Jnew is J+1,
    Out = [Jnew, MsetIJ_new]
),
MsetI_new = MsetI_old + MsetIJ,
Inew is I+1,
Out = [Inew, MsetI_new]
    ),
    Mset_new = Mset_old + MsetI,
    Lnew is L+1,
    Out = [Lnew, Mset_new]
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% optimizeEx(+Xij, +Price, -Opt)
%
% This predicate creates the function that needs to be optimized and stores it
% in Opt
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

optimizeEx(Xij, Price, Opt) :-
dim(Xij, [Imax, Jmax]),
Im is Imax + 1,
Jm is Jmax + 1,
(
    fromto([1,0], In, Out, [Im,Opt]),
    param(Xij, Price, Jm)
do
            In = [I, Opt_old],
    (
fromto([1,0], In, Out, [Jm,OptIJ]),
param(Xij, Price, I)
    do
        In = [J, OptIJ_old],
P is Price[I,J],
(
    OptIJ_old == 0,
    (
```

```
P == void,
OptIJ_new = 0
    ;
        P \== void,
OptIJ_new = Xij[I,J] * Price[I,J]
    )
;
    OptIJ_old \== 0,
    (
P == void,
OptIJ_new = OptIJ_old
    ;
        P \== void,
OptIJ_new = Xij[I,J] * Price[I,J] + OptIJ_old
    )
),
        Jnew is J + 1,
Out = [Jnew, OptIJ_new]
    ),
    (
Opt_old == 0,
Opt_new = OptIJ
    ;
        Opt_old \== 0,
Opt_new = OptIJ + Opt_old
    ),
    Inew is I + 1,
    Out = [Inew, Opt_new]
).

greaterOrEqualThanZero(Xijkm, Xijk, Xij, Xm) :-
dim(Xijkm, Dim),
writeln(Dim),
(
    foreachelem(_, Xijkm, [I,J,K,M]),
    param(Xijkm, Xijk, Xij)
do
            alg1: (Xijkm[I,J,K,M] $>= 0),
    alg1: (Xijkm[I,J,K,M] $=< 1),
    (
M == 1,
alg1: (Xijk[I,J,K] $>= 0),
alg1: (Xijk[I,J,K] $=< 1),
        (
    K == 1,
    alg1: (Xij[I,J] $>= 0),
    alg1: (Xij[I,J] $=< 1)
;
    K > 1,
    true
```

```
)
    ;
        M > 1,
        true
    )
),
(
    foreachelem(_, Xm, [M]),
    param(Xm)
do
            alg1: (Xm[M] $>= 0),
    alg1: (Xm[M] $=< 1)
).

allIntegers(Xijkm, Xijk, Xij, Xm) :-
dim(Xijkm, [I,J,K,M]),
A1 is Xijkm[1..I,1..J,1..K,1..M],
A2 is Xijk[1..I,1..J,1..K],
A3 is Xij[1..I,1..J],
A4 is Xm[1..M],
A = [A1,A2,A3,A4],
flatten(A, Af),
%writeln(Af),
alg1: integers(Af).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Helper Predicates
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% fill(+Xijkm, +Xijk, +Xij, +M, +Input)
%
% Not every entry in Xijkm etc. points to a real transformation. This is due
% to the fact that not every bid contains the same amount of transformations
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fill(Xijkm, Xijk, Xij, M, Input) :-
dim(Input, [Im, Jm, Km, _]),
Imax is Im + 1,
Jmax is Jm + 1,
Kmax is Km + 1,
(
    fromto([1], In, Out,[Imax]),
    param(Xijkm, Xijk, Xij, M, Input, Jmax, Kmax, Km)

do
```

```
            In = [I],
            (
fromto([1], In, Out, [Jmax]),
param(Xijkm, Xijk, Xij, M, Input, I, Kmax, Km)
    do
        In = [J],
(
    fromto([1, 0], In, Out, [Kmax, Kcount]),
    param(Xijkm, Xijk, M, Input, I,J)
do
            In = [K, Kcount],
    Temp is Input[I,J,K,1],
    (
Temp == void,
(
    count(L, 1, M),
    param(Xijkm, I, J, K)
do
            subscript(Xijkm, [I,J,K,L], 0)
),
subscript(Xijk, [I,J,K], 0),
Kcount_new is Kcount + 1
    ;
        Temp \== void,
Kcount_new is Kcount
    ),
    Knew is K+1,
    Out = [Knew, Kcount_new]
    ),
    (
Kcount == Km,
subscript(Xij, [I,J], 0)
    ;
        Kcount \== Km,
    ),
    Jnew is J + 1,
    Out = [Jnew]
),
    Inew is I + 1,
    Out = [Inew]
).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% createConstraintArrays(+Information, +Tm, -Xijkm, -Xijk, -Xij, Xm)
%
% +Information    : a list containing [Agents, Maxbids, Maxtrans, Goods].
```

```
% -Xijkm          : a 4D array, with Xijkm[I,J,K,M] == 1 if transformation ijk is
%                   in location m in the sequence and 0 otheriwse.
% -Xijk           : A 3D array, with Xijk[I,J,K] == 1 if the transformation ijk is
%                   used in the sequence.
% -Xij            : A 2D array, with Xij[I,J] == 1 if the bid ij is selected.
% -Xm             : A 1D array, with Xm[M] == 1 if position m in the sequence is
%                   filled.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

createConstraintArrays([A,Mb,Mt,_], Tm, Xijkm, Xijk, Xijk2, Xij, Xij2, Xm, Sequence) :-
dim(Xijkm, [A,Mb,Mt,Tm]),
dim(Xijk, [A,Mb,Mt]),
dim(Xijk2, [A,Mb,Mt]),
dim(Xij, [A,Mb]),
dim(Xij2, [A,Mb]),
dim(Xm, [Tm]),
dim(Sequence, [Tm]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% getSequence(+Xijkm, +Sequence)
%
% This predicate extracts the sequence of transformations that constitutes
% the solution to the WDP.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getSequence(Xijkm, Sequence) :-
(
    foreachelem(El, Xijkm, [I,J,K,M]),
    param(Sequence)
do
            alg1: eplex_var_get(El, typed_solution, El),
    (
El == 1,
subscript(Sequence, [M], [I,J,K])
    ;
        true
    )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% getAcceptedBids(+Xij, -AB)
%
% This predicate extracts the bids that are accepted by the the solution to the
%  WDP.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
getAcceptedBids(Xij, AB) :-
dim(Xij, [I,J]),
Im is I + 1,
Jm is J + 1,
(
    fromto([1,[]], In, Out, [Im, AB]),
    param(Xij, Jm)
do
            In = [I, AB_old],
            (
fromto([1,[]], In, Out, [Jm, ABI]),
param(Xij, I)
    do
        In = [J, ABI_old],
        El is Xij[I,J],
alg1: eplex_var_get(El, typed_solution, El),
Jnew is J + 1,
(
    El == 1,
    Out = [Jnew,[(I,J)|ABI_old]]
;
    Out = [Jnew, ABI_old]
)
    ),
    Inew is I + 1,
    flatten([AB_old, ABI], AB_new),
    Out = [Inew, AB_new]
).


createSum([X1,X2], [X1 + X2]) :-
var(X1),
var(X2).

createSum([X1,X2], [X1+0]) :-
var(X1),
not(var(X2)).

createSum([X1,X2], [X2+0]) :-
not(var(X1)),
var(X2).

createSum([X1,X2], [0]) :-
not(var(X1)),
not(var(X2)).

createSum([X], [X]):-
var(X).
```

```
createSum([X], [0]) :-
not(var(X)).

createSum([X|R], [X+Sum]):-
var(X),
createSum(R, [Sum]).

createSum([X|R], Sum):-
not(var(X)),
createSum(R, Sum).
```

Appendix B

Implementation of the CP approach

```
:- use_module("../read_input2").
:- lib(ic).
:- lib(branch_and_bound).
:- lib(array_constraints_ic).
:- lib(lists).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% solve(+Filename)
%
% This predicate reads in the information that is contained in Filename. It then
% creates the constraints that are needed to solve the problem and uses minimize\2
% to find an optimal solution
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

solve_ic(Filename, UsedTime) :-
readInput2(Filename, Input, Output, Price, Start, End, Information, Nijk
        , T),
cputime(StartTime),
createArrays(Information, T, Xm, Xn, Bijk, Bij, Bm, Sequence),
specifyDomains(T, Xm, Xn, Bijk, Bij, Bm),
createConstraint1and2(Xm, Xn),
createConstraintAllDiff(Xm, Xn),
createConstraint3(T, Bijk, Xn),
createConstraint4(Bij, Bijk, Nijk),
createConstraint5(Bij),
createConstraint6(T, Start, Bijk, Input, Output, Xm),
createConstraint7(T, Start, End, Bijk, Input, Output, Xm),
createConstraint9(T, Bm, Xm),
createConstraint8(T, Bm),
```

```
createConstraint10(Bijk, Bm, Xm),
optimizeExpr(Bij, Price, Opt),
{get_elements(Xm, XmE)},
garbage_collect,
bb_min(labelingTest(XmE, T, Constraints), Opt, bb_options{strategy:restart}),!,
cputime(EndTime),
UsedTime is EndTime - StartTime.

labelingTest(X, T, Constraints) :-
halve(X, Xt, Xr),
myLabeling(Xt, T, [], Constraints),
labelingRest(Xr).

myLabeling([], _, _, _).

myLabeling([X|L], T, Done, _) :-
get_bounds(X, Lo, _),
( Lo > T ->
    !, labelingRest([X|L])
;
   indomain(X),
   myLabeling(L, T, [X|Done], ConList)
       ).

labelingRest([]) :- !.

labelingRest([X|L]) :-
indomain(X), !,
labelingRest(L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                                %
% Create the constraints                                                         %
%                                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

createConstraint1and2(Xm, Xn) :-
(
    foreachelem(_, Xn, [N]),
    param(Xm, Xn)
do
            T1 is Xn[N],
    T2 is Xm[N],
            {
     Xm[T1] = N,
     Xn[T2] = N
    }
).
```

```
createConstraint3(T, Bijk, Xn) :-
(
    foreachelem(_, Bijk, [N]),
    param(Bijk, Xn, T)
do
            ( N =< T ->
T1 is Xn[N],
T2 is Bijk[N],
#=<(T1,T,T2)
    ;
       {Bijk[N] = 0}
   )
).

createConstraint4(Bij, Bijk, Nijk) :-
(
    foreachelem([I,J,_], Nijk, [N]),
    param(Bij,Bijk,Bijk)
do
            T1 is Bij[I,J],
    T2 is Bijk[N],
    #=(T1, T2)
).

createConstraint5(Bij) :-
dim(Bij, [I,J]),
Im is I,
Jm is J + 1,
(
    count(I, 1, Im),
    param(Bij, Jm)
do
            (
fromto([1,0], In, Out, [Jm, Sum]),
param(I, Bij)
    do
        In = [J, Sum_old],
(
    Sum_old == 0,
    Sum_new = Bij[I,J]
;
    Sum_old \== 0,
    Sum_new = Sum_old + Bij[I,J]
),
Jnew is J + 1,
Out = [Jnew, Sum_new]
    ),
    (eval(Sum) #=< 1)
).
```

```
createConstraint6(T, Start, Bijk, Input, Output, Xm) :-
dim(Input, [_,G1]),
(
    count(M, 1, T),
    param(Input, Output, Start, Bijk, Xm, G1)
do
            (
count(G, 1, G1),
param(Input, Output, Start, Bijk, Xm, M)
    do
        Mminus is M - 1,
multiset(Start, Bijk, Input, Output, Xm, Mminus, G, Mset),
T2 is Xm[M],
{T1 = Input[T2,G]},
List is Xm[1..Mminus],
        %suspend(eval(Mset) #>= T1, 0, List->inst)
eval(Mset) #>= T1
    )
).

createConstraint6(T, Start, Bijk, Input, Output, Xm, ConstraintsR) :-
T2 is T + 1,
dim(Input, [_,G1]),
G2 is G1 + 1,
(
    fromto([1, []], In, Out, [T2, Constraints]),
    param(Input, Output, Start, Bijk, Xm, G2)
do
            In = [M, ConstraintsOld],
            (
fromto([1, []], In, Out, [G2, ConstraintsTemp]),
param(Input, Output, Start, Bijk, Xm, M)
    do
        In = [G, ConstraintsOldTemp],
Mminus is M - 1,
multiset(Start, Bijk, Input, Output, Xm, Mminus, G, Mset),
{T1 = Input[Xm[M],G]},
ConstraintsNewTemp = [eval(Mset) #>= T1|ConstraintsOldTemp],
Gnew is G + 1,
Out = [Gnew, ConstraintsNewTemp]
    ),
    reverse(ConstraintsTemp, ConstraintsTempR),
    ConstraintsNew = [ConstraintsTempR|ConstraintsOld],
    Mnew is M + 1,
    Out = [Mnew, ConstraintsNew]
),
reverse(Constraints, ConstraintsR).
```

```
createConstraint7(T, Start, End, Bijk, Input, Output, Xm) :-
dim(Start, [Gmax]),
(
    count(G, 1, Gmax),
    param(T, Start, End, Bijk, Input, Output, Xm)
do
            multiset(Start, Bijk, Input, Output, Xm, T, G, Mset),
    eval(Mset) #>= eval(End[G])
).

createConstraint8(T, Bm) :-
Tm is T - 1,
(
    count(M, 1, Tm),
    param(Bm)
do
            M2 is M + 1,
    eval(Bm[M]) #=< eval(Bm[M2])
).

createConstraint9(T, Bm, Xm) :-
(
    foreachelem(_, Xm, [N]),
    param(Xm, Bm, T)
do
          ( N =< T ->
       T1 is Xm[N],
       T2 is Bm[N],
       #>(T1, T, T2)
   ;
       true
   )
).

createConstraint10(Bijk, Bm, Xm) :-
(
    foreachelem(_, Xm, [N]),
    param(Xm, Bijk, Bm)
do
            T3 is Xm[N],
            {T1 = Bijk[T3],
    T2 = Bm[N]},
    #\=(T1, 1, T2)
).

createConstraintAllDiff(Xm, Xn) :-
{
          get_elements(Xm, XmE),
  get_elements(Xn, XnE)
       },
```

```
alldifferent(XmE),
alldifferent(XnE).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                                %
% Helping Predicates                                                             %
%                                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% multiset(+Start, +Bijk, +Input, +Output, +Xm, +M, +G, -Mset)
%
% This predicate creates, for a good G, the sum that calculates the amount of
% goods that is available at position M in the sequence. The array_constraints
% library is used to define constraints on nested arrays
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

multiset(Start, Bijk, Input, Output, Xm, M, G, Mset) :-
M2 is M + 1,
(
    fromto([1,Start[G]], In, Out, [M2, Mset]),
    param(Input, Output, Bijk, Xm, G)
do
            In = [L, Sum_old],
    {Bijk[Xm[L]] = T1},
    {Output[Xm[L], G] = T2},
    {Input[Xm[L],G] = T3},
    (
Sum_old == 0,
Sum_new = T1*(T2 - T3)
    ;
        Sum_old \== 0,
Sum_new = Sum_old + T1*(T2 - T3)
    ),
    Lnew is L + 1,
    Out = [Lnew, Sum_new]
),
!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% createArrays(+Information, +T, -Xm, -Xn, -Bijk, -Bij, -Sequence)
%
% This predicate creates the arrays that are needed to define the constraints.
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

createArrays([A,Mb,_,_], T, Xm, Xn, Bijk, Bij, Bm, Sequence) :-
T2 is T*2,
dim(Xm, [T2]),
dim(Bm, [T2]),
dim(Xn, [T2]),
dim(Bijk, [T2]),
dim(Bij, [A,Mb]),
        dim(Sequence, [T]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% specifyDomains(+T, +Xm, +Xn, +Bijk, +Bij)
%
% This predicate specifies the domains of the problem variables
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

specifyDomains(T, Xm, Xn, Bijk, Bij, Bm) :-
T2 is T*2,
{Xm :: 1..T2,
Xn :: 1..T2,
Bijk :: 0..1,
Bij :: 0..1,
Bm :: 0..1}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% optimzeExpr(+Bij, +Price, -Opt)
%
% This predicate defines the variable that is used by minimize\2 to find the
% optimal solution. In this case it is the some of the prices of the selected bids
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

optimizeExpr(Bij, Price, Opt) :-
dim(Bij, [Imax, Jmax]),
Im is Imax + 1,
Jm is Jmax + 1,
(
    fromto([1,0], In, Out, [Im,OptExpr]),
    param(Bij, Price, Jm)
do
            In = [I, Opt_old],
    (
fromto([1,0], In, Out, [Jm,OptIJ]),
param(Bij, Price, I)
```

```
    do
        In = [J, OptIJ_old],
P is Price[I,J],
(
    OptIJ_old == 0,
    (
P == void,
OptIJ_new = 0
    ;
        P \== void,
OptIJ_new = Bij[I,J] * Price[I,J]
    )
;
    OptIJ_old \== 0,
    (
P == void,
OptIJ_new = OptIJ_old
    ;
        P \== void,
OptIJ_new = Bij[I,J] * Price[I,J] + OptIJ_old
    )
),
        Jnew is J + 1,
Out = [Jnew, OptIJ_new]
    ),
    (
Opt_old == 0,
Opt_new = OptIJ
    ;
        Opt_old \== 0,
Opt_new = OptIJ + Opt_old
    ),
    Inew is I + 1,
    Out = [Inew, Opt_new]
),
Opt #= -eval(OptExpr).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% getSequence(+Xm, +Nijk, +Sequence)
%
% This predicate finds the sequence of bids that constitutes the solution
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getSequence(Xm, Nijk, Sequence) :-
dim(Sequence, [T]),
T2 is T + 1,
(
    fromto(1, M, Out, T2),
```

```
    param(Xm, Nijk, Sequence, T, T2)
do
            P is Xm[M],
    (
P =< T,
Out is M +1,
El is Nijk[P],
subscript(Sequence, [M], El)
    ;
        P > T,
Out is T2
    )
).
```

Appendix C

Implementation of the second IP approach

```
:- use_module("../read_input2").
:- lib(eplex).
:- lib(lists).
:- lib(hash).
:- eplex_instance(alg5).
:- eplex_instance(alg5sequence).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% General Methods
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% solve(+Name, -UsedTime)
%
% This is the main predicate. It gets as inpute a filename, opens this file,
% reads in
% the bids and calls the solver. In the end, it outputs the CPU time that was
% needed
% to solve the auction.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

solve(Filename, UsedTime) :-
readInput1(Filename, Input, Output, Price, Start, End, Information
                    , _Tm),
cputime(StartTime),
createArrays(Information, Bij, InputB, OutputB),
```

```
mergeInput(Input, Output, InputB, OutputB, Information),!,
callSolver(Information, Bij, InputB, OutputB, Input, Output, Start
                    , End, Price),
cputime(EndTime),
UsedTime is EndTime - StartTime.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% callSolver(+Information, +Bij, +InutB, +OutputB, +Input, +Output, +Start,
%            +End, +Price)
%
% This predicate is responsable for creating the constraints that solve the
% first part of the problem and to call upon the predicate that solves the
% second part of the problem
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


callSolver(Information, Bij, InputB, OutputB, Input, Output, Start, End
           , Price) :-
optimizeExpr(Bij, Price, OptExpr),
createEplexConstraints(Bij, Start, End, InputB, OutputB),!,
alg5:eplex_solver_setup(max(OptExpr), _, [method(primal)
                                , timeout(3600), abort_handler(aborted(boe))]
                                , 0, []),
findOptimalSequence(Information, Input, Output, Start, Bij, 0).

aborted(boe) :- writeln("execution was aborted").

findOptimalSequence(_, _, _, _, _, PassedTime) :-
PassedTime > 3600.

findOptimalSequence(_Gm, Input, Output, Start, Bij, PassedTime) :-
AvailableTime is 3600 - PassedTime,
cputime(CurrentTime),
alg5:eplex_set(timeout, AvailableTime),
alg5:eplex_solve(Cost),!,
dim(Bij, I),
dim(BijCopy, I),
getBij(BijCopy, Bij),!,
(
    % find out whether the allocation is proper.
    cputime(CurrentTime2),
    PassedTime2 is PassedTime + (CurrentTime2 - CurrentTime),
    findSequence(Input, Output, Start, BijCopy, _Sequence
                         , PassedTime2)
;
    % The allocation is not proper, find a new allocation
    setExclusion(BijCopy, Bij),
```

```
    alg5sequence:eplex_cleanup,
    cputime(CurrentTime2),
    PassedTime2 is PassedTime + (CurrentTime2 - CurrentTime),
    findOptimalSequence(_Gm, Input, Output, Start, Bij, PassedTime2)
),!.

getBij(BijCopy, Bij) :-
(
    foreachelem(El, Bij, [I,J]),
    param(BijCopy)
do
            T is BijCopy[I,J],
            alg5: eplex_var_get(El, typed_solution, T)
), !.

findSequence(_, _, _, _, _, PassedTime) :-
PassedTime > 3600.

findSequence(Input, Output, Start, BijCopy, _Sequence, PassedTime) :-
cputime(CurrentTime),
getTransformations(BijCopy, Input, Output, AllInput, AllOutput
                         , Trans),
( Trans = [] ->
    true
;
    createSequenceArrays(AllInput, Pos),
    setSequenceConstraints(AllInput, AllOutput, Start, Pos),!,
    flatten_array(Pos, PosE),
    createSum2(PosE, Sum),
    dim(AllInput, [I,_]),
    alg5sequence: (Sum $>= I),
    cputime(CurrentTime2),
    AvailableTime is 3600 - PassedTime - (CurrentTime2 - CurrentTime),
    ( AvailableTime > 0 ->
alg5sequence:eplex_solver_setup(max(Sum), _
                              , [timeout(AvailableTime)
                              , abort_handler(aborted(boe))], []),
alg5sequence:eplex_solve(C),!
    ;
        true
    )
).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
% Create the constraints                                                       %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
createEplexConstraints(Bij, Start, End, InputB, OutputB) :-
allIntegers(Bij),
greaterThenOrEqualToZero(Bij),
createEplexConstraint1(Bij),
createEplexConstraint2(Bij, Start, End, InputB, OutputB).


allIntegers(Bij) :-
flatten_array(Bij, BijE),
alg5: integers(BijE).

greaterThenOrEqualToZero(Bij) :-
(
    foreachelem(_, Bij, [I,J]),
    param(Bij)
do
            alg5: (Bij[I,J] $>= 0),
    alg5: (Bij[I,J] $=< 1)
).


createEplexConstraint1(Bij) :-
dim(Bij, [A, Mb]),
(
    count(I, 1, A),
    param(Bij, Mb)
do
            Temp is Bij[I,1..Mb],
    createSum(Temp, Sum),
    constraint3(Sum)
).

% this is done to cope with the situation where the sum consists of a
% single variable
constraint3([Sum]) :-
alg5: (Sum $=< 1).

createEplexConstraint2(Bij, Start, End, InputB, OutputB) :-
dim(Bij, [I,J]),
Im is I + 1,
Jm is J + 1,
(
    foreachelem(_, Start, [G]),
    param(Bij, Start, End, InputB, OutputB, Im, Jm)
do
            (
fromto([1,Start[G]], In, Out, [Im, Mset]),
param(Bij, InputB, OutputB, Jm, G)
    do
```

```
        In = [I, MsetOld],
        (
    fromto([1, 0], In, Out, [Jm, MsetI]),
    param(Bij, InputB, OutputB, I, G)
do
            In = [J, MsetIOld],
    Temp is (OutputB[I,J,G] - InputB[I,J,G]),
    MsetINew = MsetIOld + Bij[I,J] * Temp,
    Jnew is J + 1,
    Out = [Jnew, MsetINew]
),
Inew is I + 1,
MsetNew = MsetOld + MsetI,
Out = [Inew, MsetNew]
    ),
    alg5: (Mset $>= End[G])
), !.


createExcludeConstraint(BijCopyE, BijE) :-
createSum(BijCopyE, S),
(
    fromto([BijE, 0], In, Out, [[], Sum])
do
            In = [[El|BijE], SumOld],
    (El == 1 ->
SumNew = SumOld + El
    ;
        SumNew = SumOld
    ),
    Out = [BijE, SumNew]
),
alg5: (Sum $=< S).

setSequenceConstraints(AllInput, AllOutput, Start, Pos) :-
flatten_array(Pos, PosE),
alg5sequence:integers(PosE),
setConstraintBin(Pos),
setConstraintAll(Pos),
setConstraintSequence(AllInput, AllOutput, Start, Pos).

setConstraintBin(Pos) :-
(
    foreachelem(_, Pos, [I,M]),
    param(Pos)
do
           alg5sequence: (Pos[I,M] $=< 1),
   alg5sequence: (Pos[I,M] $>= 0)
       ).
```

```
setConstraintAll(Pos) :-
dim(Pos, [T,_]),
(
    count(I, 1, T),
    param(Pos, T)
do
    List is Pos[I,1..T],
    List2 is Pos[1..T,I],
    createSum2(List, Sum),
    createSum2(List2, Sum2),
    alg5sequence: (Sum $=< 1),
    alg5sequence: (Sum2 $=< 1)
).

setConstraintSequence(AllInput, AllOutput, Start, Pos) :-
dim(Start, [Gm]),
dim(AllInput, [T,_]),
        (
    count(G, 1, Gm),
    param(AllInput, AllOutput, Start, Pos, T)
do
    (
count(M, 1, T),
param(AllInput, AllOutput, Start, Pos, G, T)
    do
        Mmin is M - 1,
        multiset(Start, Pos, AllInput, AllOutput, Mmin, G, T, Mset),
createSumInput(Pos, AllInput, M, G, Sum),
alg5sequence: (Sum $=< Mset)
    )
),!.

createSumInput(Pos, AllInput, M, G, Sum) :-
dim(Pos, [Im,_]),
(
    count(I, 1, Im),
    fromto(0, OldS, NewS, Sum),
    param(Pos, AllInput, M, G)
do
            NewS = OldS + Pos[I,M]*AllInput[I, G]
).


setExclusion(BijCopy, Bij) :-
% writeln(BijCopy),
(
    foreachelem(El, BijCopy, [I,J]),
    fromto(0, OldSum, NewSum, Sum),
    param(Bij)
do
```

```
            ( El = 0 ->
NewSum = OldSum + Bij[I,J]
    ;
        NewSum = OldSum + (1-Bij[I,J])
    )
),
alg5: (Sum $>= 1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                          %
% Helping Predicates                                                       %
%                                                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% multiset(+Start, +Pos, +Input, +Output, +M, +G, -Mset)
%
% This predicate creates, for a good G, the sum that calculates the amount
% of goods that is available
% at position M in the sequence.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

multiset(Start, Pos, Input, Output, M, G, T, Mset) :-
M2 is M + 1,
T2 is T + 1,
(
    fromto([1,Start[G]], In, Out, [M2, Mset]),
    param(Input, Output, Pos, G, T2)
do
            In = [L, Mset_old],
    (
fromto([1,0], In, Out, [T2, MsetT]),
param(Input, Output, Pos, L, G)
    do
        In = [L2, MsetT_old],
MsetT_new = MsetT_old + Pos[L2, L]*(Output[L2, G]
                                                    - Input[L2, G]),
L2new is L2 + 1,
Out = [L2new, MsetT_new]
    ),
    Mset_new = Mset_old + MsetT,
    Lnew is L + 1,
    Out = [Lnew, Mset_new]
),!.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% createArrays(+Information, +T, -Xm, -Xn, -Bijk, -Bij, -Sequence)
%
% This predicate creates the arrays that are needed to define the constraints.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

createArrays([A,Mb,_,G], Bij, InputB, OutputB) :-
dim(Bij, [A,Mb]),
        dim(InputB, [A, Mb, G]),
dim(OutputB, [A, Mb, G]).


createSequenceArrays(AllInput, Pos) :-
dim(AllInput, [I, _]),
dim(Pos, [I,I]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% specifyDomains(+T, +Xm, +Xn, +Bijk, +Bij)
%
% This predicate specifies the domains of the problem variables
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

specifyDomains(T, Xm, Xn, Bijk, Bij, Bm) :-
T2 is T*2,
{Xm :: 1..T2,
Xn :: 1..T2,
Bijk :: 0..1,
Bij :: 0..1,
Bm :: 0..1}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% optimzeExpr(+Bij, +Price, -Opt)
%
% This predicate defines the variable that is used by minimize\2 to find
% the optimal solution. In this case it is the some of the prices of the
% selected bids
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

optimizeExpr(Bij, Price, OptExpr) :-
dim(Bij, [Imax, Jmax]),
Im is Imax + 1,
Jm is Jmax + 1,
(
    fromto([1,0], In, Out, [Im,OptExpr]),
```

```
    param(Bij, Price, Jm)
do
            In = [I, Opt_old],
    (
fromto([1,0], In, Out, [Jm,OptIJ]),
param(Bij, Price, I)
    do
        In = [J, OptIJ_old],
P is Price[I,J],
(
    OptIJ_old == 0,
    (
P == void,
OptIJ_new = 0
    ;
        P \== void,
OptIJ_new = Bij[I,J] * Price[I,J]
    )
;
    OptIJ_old \== 0,
    (
P == void,
OptIJ_new = OptIJ_old
    ;
        P \== void,
OptIJ_new = Bij[I,J] * Price[I,J] + OptIJ_old
    )
),
        Jnew is J + 1,
Out = [Jnew, OptIJ_new]
    ),
    (
Opt_old == 0,
Opt_new = OptIJ
    ;
        Opt_old \== 0,
Opt_new = OptIJ + Opt_old
    ),
    Inew is I + 1,
    Out = [Inew, Opt_new]
).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% getTransformations(+BijCopy, +Input, +Output, +AllInput, +AllOutput, -Trans)
%
% The purpose of this predicate is to collect all the transformations that
% belong to the selected bids. These transformations are stored in Trans
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

getTransformations(BijCopy, Input, Output, AllInput, AllOutput, Trans) :-
dim(Input, [_, _, Km, Gm]),
Kmax is Km + 1,
Gmax is Gm + 1,
(
    foreachelem(El, BijCopy, [I,J]),
    fromto(0, Counter, NewCounter, S),
    fromto([], OldL, NewL, List),
    fromto([], OldL2, NewL2, List2),
    param(Input, Output, Kmax, Gmax)
do
            (El = 1 ->
(
    fromto([1, [], []], In, Out, [Kmax, ListK, ListK2]),
    fromto(Counter, OldCounterK, NewCounterK, NewCounter),
    param(Input, Output, I, J, Gmax)
do
            In = [K, OldLK, OldLK2],
    ( subscript(Input, [I,J,K,1], void) ->
NewLK = OldLK,
NewLK2 = OldLK2,
NewCounterK = OldCounterK
    ;
        NewCounterK is OldCounterK + 1,
                (
    fromto([1,[]], In, Out, [Gmax, ListGK]),
    param(Input, Output, I, J, K)
do
            In = [G, OldListGK],
    T1 is Input[I,J,K,G],
            T2 is Output[I,J,K,G],
    NewListGK = [T1-T2|OldListGK],
    NewG is G + 1,
    Out = [NewG, NewListGK]
),
NewLK = [ListGK | OldLK],
NewLK2 = [(I-J-K)|OldLK2]
    ),
    NewK is K + 1,
    Out = [NewK, NewLK, NewLK2]
),
NewL = [ListK|OldL],
NewL2 = [ListK2|OldL2]
    ;
        NewL = OldL,
NewL2 = OldL2,
        NewCounter is Counter
```

```
    )
),
( S = 0 ->
    dim(Trans, [0])
;
    flatten(List, ListE),
    flatten(List2, List2E),
    dim(AllInput, [S, Gm]),
    dim(AllOutput, [S, Gm]),
    dim(Trans, [S]),
    (
foreach(El, List2E),
count(K, 1, M),
param(Trans)
    do
                subscript(Trans, [K], El)
    ),
    (
fromto([1,Gm,ListE], In, Out, [_,_,[]]),
param(AllInput, AllOutput, Gm)
    do
                In = [I, G, [Input-Output|Rest]],
subscript(AllInput, [I,G], Input),
subscript(AllOutput, [I,G], Output),
( G = 1 ->
    NewG is Gm,
    NewI is I + 1
;
            NewG is G - 1,
    NewI is I
),
Out = [NewI, NewG, Rest]
    )
).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% mergeInput(Input, Output, InputB, OutputB)
%
% This predicate takes as input Input and Output, and merge all the
% transformations belonging to one bid, storing them in InputB and OutputB
%
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


mergeInput(Input, Output, InputB, OutputB,  [_, _, _, Gm]) :-
dim(Input, [Im, Jm, Km, Gm]),
Kmax is Km + 1,
```

```
(
    count(G, 1, Gm),
    param(Input, Output, InputB, OutputB, Im, Jm, Kmax)
do
            (
count(I, 1,Im),
param(Input, Output, InputB, OutputB, Jm, Kmax, G)
    do
        (
    count(J, 1, Jm),
    param(Input, Output, InputB, OutputB, I, Kmax, G)
do
            (
fromto([1,0,0], In, Out, [Kmax, IB, OB]),
param(Input, Output, I, J, G, Kmax)
    do
        In = [K, IB, OB],
T1 is Input[I,J,K,G],
( T1 = void ->
    IBNew is IB,
    OBNew is OB,
    Knew is Kmax
;
    IBNew is IB + Input[I,J,K,G],
    OBNew is OB + Output[I,J,K,G],
    Knew is K + 1
),
Out = [Knew, IBNew, OBNew]
    ),
    subscript(InputB, [I, J, G], IB),
    subscript(OutputB, [I, J, G], OB)
)
    )
),
fill(InputB, 0),
fill(OutputB, 0),
!.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% fill(+Array, +Value)
%
% This predicatae takes as input an Array and some value, and it fills
% all the elements in the array that have not been instantiated yet with
% the value Value
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fill(Array, Value) :-
```

```
(
    foreachelem(El, Array),
    param(Value)
do
            ( var(El) ->
El = Value
    ;
       true
    )
), !.

createSum([X1,X2], [X1 + X2]) :-
var(X1),
var(X2).

createSum([X1,X2], [X1+0]) :-
var(X1),
not(var(X2)).

createSum([X1,X2], [X2+0]) :-
not(var(X1)),
var(X2).

createSum([X1,X2], [0]) :-
not(var(X1)),
not(var(X2)).

createSum([X], [X]):-
var(X).

createSum([X], [0]) :-
not(var(X)).

createSum([X|R], [X+Sum]):-
var(X),
createSum(R, [Sum]).

createSum([X|R], Sum):-
not(var(X)),
createSum(R, Sum).

createSum2([], 0).

createSum2([X|R], Sum) :-
createSum2(R, Sum1),
Sum = Sum1 + X.
```

Bibliography

[Apt and Wallace, 2006] K. Apt and M. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, 2006.

[Apt, 2003] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[Brand, 2001] S. Brand. Constraint Propagation in Presence of Arrays. Series: Computer Science, 16, 2001.

[Bratko, 2001] I. Bratko. *Prolog Programming for Artificial Intelligence*. Pearson Education, 2001.

[Cerquides *et al.*, 2007] J. Cerquides, U. Endriss, A. Giovannucci, and J. A. Rodríguez Aguilar. Bidding languages and winner determination for mixed multi-unit combinatorial auctions. In M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007)*, pages 1221–1226, January 2007.

[Clarke, 1971] E. H. Clarke. Multipart pricing of public goods. *Public Choice*, 11(1), September 1971.

[Cramton *et al.*, 2006] P. Cramton, Y. Shoham, and R. Steinberg. *Combinatorial Auctions*. MIT Press, 2006.

[Dantzig, 1963] G.B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[Eiselt *et al.*, 2000] H.A. Eiselt, C. Sandblom, and K. Spielberg. *integer programming and network models*. Springer, 2000.

[Fourier, 1827] J.B.J. Fourier. Engl. transl. (partially) in: D.A. kohler, translation of a report by Fourier on his work on linear inequalities. (reported in:) Analyse des traveaux de 'Academie Royale de Sciences pendant l'annee 1824, 1827.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. Freeman, cop., New York, 1979.

[Green and Laffont, 1979] J. Green and J. Laffont. *Incentives in Public Decision Making*. North Holland, 1979.

[Groves, 1973] T. Groves. Incentives in teams. *Econometrica*, 41(4):617–31, July 1973.

[Holmstrom, 1979] B. Holmstrom. Grove schemes on restricted domains. *Econometrica*, 47:1137–1144, 1979.

[Klemperer, 2004] P. Klemperer. *Auction: Theory and Practice*. Princeton University Press, 2004.

[Leyton-Brown *et al.*, 2000] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. ACM Press, 2000.

[Nisan, 2000] N. Nisan. Bidding and allocation in combinatorial auctions. In *ACM Conference on Electronic Commerce*, pages 1–12, 2000.

[Papadimitriou, 1981] C. H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, 1981.

[Rothkopf *et al.*, 1998] M. H. Rothkopf, A. Pekec, and R. M. Harstad. Computationally manageable combinational auctions. *Manage. Sci.*, 44(8):1131–1147, 1998.

[Sandholm *et al.*, 2002] T. W. Sandholm, S. Suri, A. Gilpin, and D. Levine. Winner determination in combinatorial auction generalizations. In *Proc. AAMAS-2002*. ACM Press, 2002.

[Schrijver, 1986] A. Schrijver. *Theory of linear and integer programming*. J. Wiley, 1986.

[Uckelman and Endriss, 2007] J. Uckelman and U. Endriss. A constraint programming formulation of the WDP for mixed mulit-unit combinatorial auctions. Private correspondence, 2007.

[Vickrey, 1961] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, Mar 1961.

[Vinyals *et al.*, 2007] M. Vinyals, A. Giovannucci, J. Cerquides, P. Messeguer, and J. A. Rodríguez-Aguilar. Towards a realistic bid generator for mixed multi-unit combinatorial auctions. In *14th RCRA workshop: Experimental evaluation of algorithms for solving problems with combinatorial explosion*, 2007.