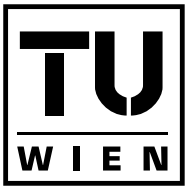
. . . . . .

Univ.Prof. Dr. Georg Gottlob



TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTERARBEIT

Hypertree Decompositions for Optimal Winner Determination in Combinatorial Auctions

ausgeführt am

Institut für Informationssysteme
Abteilung für Datenbanken und Artificial Intelligence
der Technischen Universität Wien

unter der Anleitung von

O.Univ.Prof. Dr.techn. Georg Gottlob

Privatdoz. Dr.techn. Nysret Musliu

Projektass. Dipl.-Ing. Werner Schafhauser

durch

Ekaterina Lebedeva

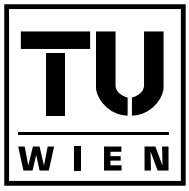
Wien, 10. März 2008

. . . . . .

Ekaterina Lebedeva

. . . . . .

Univ.Prof. Dr. Georg Gottlob



TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MASTER THESIS

Hypertree Decompositions for Optimal Winner Determination in Combinatorial Auctions

carried out at the

Institute of Information Systems
Database and Artificial Intelligence Group
of the Vienna University of Technology

under the instruction of

O.Univ.Prof. Dr.techn. Georg Gottlob

Privatdoz. Dr.techn. Nysret Musliu

Projektass. Dipl.-Ing. Werner Schafhauser

by

Ekaterina Lebedeva

Vienna, March 10, 2008

. . . . . .

Ekaterina Lebedeva

Contents

List of Figures

Abstract

A combinatorial auction is an auction in which various items are sold and a bid can be placed for more than one item at once. The winner determination problem for a combinatorial auction is the task of determining a set of mutually disjoint bids that brings the maximal revenue from the auction. This problem is known to be NP-complete.

To cope with the NP-completeness of the winner determination problem there were attempts to identify the most general classes of combinatorial auction instances on which the problem is feasible in polynomial time. One of these attempts resulted in a polynomial time algorithm, called $\mathsf{ComputeSetPacking}_k$, that utilizes the notion of hypertree decomposition. The algorithm is applied on combinatorial auction instances with associated dual hypergraphs having the hypertree width bounded by some natural number.

In this thesis we describe the essential concepts involved in solving the winner determination problem by means of the $\mathsf{ComputeSetPacking}_k$ algorithm. We implemented the algorithm, and the description of our implementation is given in the thesis. The experimental evaluation of our implementation showed how the essential parameters of the combinatorial auctions (distributions used for generating problem instances, numbers of items and bids) and of the decomposition method (the width and the size of the constructed hypertree) influence the performance of the algorithm. The results and the analysis of our experimental tests, as well as a comparison of $\mathsf{ComputeSetPacking}_k$ with the other approaches for different distributions with respect to their hardness to be solved, are also presented in the thesis.

Dedication

I dedicate my work to my grandparents and to my parents, whose constant love and support have always been the main source of my strength and inspiration.

Acknowledgements

This thesis would not take place without the suggestion of an interesting and challenging topic given by Prof. Georg Gottlob, whom I would like to acknowledge for supervising my work and for providing the guidelines for my research in Combinatorial Auctions. I am very grateful to Prof. Nysret Musliu for advising my work, supporting and encouraging me, as well as for revising this thesis. I would also like to thank a Ph.D. student of Prof. Musliu, Werner Schafhauser, for his assistance in the moments of my theoretical and practical doubts, for interesting discussions and for very useful suggestions.

Moreover, I would like to acknowledge Stepan Mikhaylov for his significant help in solving programming issues.

The two years of my M.Sc. studies were funded by the Erasmus Mundus Scholarship, for which I am grateful to the European Commission. I gratefully acknowledge the coordinators of the European Master in Computational Logic program, Prof. Enrico Franconi at Free University of Bozen-Bolzano and Prof. Alexander Leitsch at Vienna University of Technology, for their essential help with organizational difficulties.

My special thank to Bruno Woltzenlogel Paleo for the philosophical ideas in the moments of uncertainty, for the moral support in the moments of anxiety, for the inspiration in the moments of fatigue and for making happy every moment that we spent together.

Finally, I am very much thankful to all the friends with whom I have had wonderful, enjoyable and unforgettable time in Italy and in Austria.

Chapter 1

Introduction

In combinatorial auctions bidders can place their bids not only on single items, but also on sets of items. This models the case when a bidder's value for a set of items is different than the sum of the values of the individual items.

The concept of combinatorial auctions has acquired a particular practical importance since it may be applicable in many real-world domains. Combinatorial auction mechanisms were introduced as early as 1976 by Charles L. Jackson in his Ph.D. thesis "Technology for Spectrum Markets" for radio spectrum rights [16]. Since then the interest in combinatorial auctions has been growing as the rise in computing power made the implementation of combinatorial auctions more attractive [5]. Recently combinatorial auctions have been employed in a variety of industries. Some of the examples in which combinatorial auctions have been used along with other mechanisms are: truckload transportation, airport arrival and departure slots, bus routes, allocating radio spectrum for wireless communication services and electronic trading systems [4, 5].

The study of combinatorial auctions lies in the intersection of economics, operations research and computer science [4]. In this thesis we consider the combinatorial auctions discipline from the computer science perspective.

In a combinatorial auction an auctioneer receives price offers for various bundles of items. In terms of combinatorial auctions these offers are called bids. The auctioneer is interested in maximizing his revenue from the auction when determining which bids he accepts. Bids accepted by the auctioneer are called winning bids, and the winner determination must be such that each item may appear in at most one winning bid.

This problem, called "winner determination problem", is known to be NP-complete [23, 24], meaning that a polynomial-time algorithm that solves the problem is unlikely to exist unless P=NP. Furthermore, the problem is not approximable in polynomial time unless NP=ZPP [24]. Hence, there have been much research efforts addressing the problem, which may be clas-

sified as follows [3, 18]:

1. Designing approximate algorithms that are provably fast, but on some problem instances may give results far from the optimal solution [10].

2. Imposing restrictions on the combinatorial auction instance (e.g., restricting bid prices or the bundles on which bids can be placed).

3. Designing decomposition methods for the problem, and tree search algorithms that provably find an optimal solution [10, 24, 27].

The last approach is associated with the desire to identify the most general classes of instances on which the winner determination problem is feasible in polynomial time. In [3] it is proposed to use the notion of item graphs, coming from graph theory, to isolate the tractable class of combinatorial auctions. The winner determination problem is known to be feasible in polynomial time on instances having associated item graphs with bounded treewidth [3]. However, the problem of determining whether an item graph with tree width equal to 3 exists was proven to be intractable [11].

Therefore, a different kind of structural requirement to single out tractable classes of combinatorial auctions was investigated. More precisely, the notion of hypertree decomposition was applied. The winner determination problem is solvable in polynomial time on combinatorial auction instances which associated dual hypergraphs have bounded hypertree width [11]. Moreover, the class of tractable instances identified by means of hypertree decomposition approach properly contains the class of instances having an item item graph with bounded tree width [11].

Furthermore, a polynomial time algorithm, called $\mathsf{ComputeSetPacking_k}$, that utilizes the notion of hypertree decomposition was proposed in [11]. Let $\mathcal{C}(\widetilde{\mathcal{H}}, k)$ denote the class of all the hypergraphs whose dual hypergraphs have hypertree width bounded by a fixed k. The winner determination problem can be solved in polynomial time on the class $\mathcal{C}(\widetilde{\mathcal{H}}, k)$ by means of the algorithm $\mathsf{ComputeSetPacking_k}$ [11].

In this thesis we present the main notions of combinatorial auctions theory, giving an overview of previous research targeted on solving the winner determination problem for combinatorial auctions. Moreover, we describe the essential concepts involved in solving the winner determination problem by means of $\mathsf{ComputeSetPacking_k}$ algorithm. We implemented and experimentally evaluated $\mathsf{ComputeSetPacking_k}$. The experiments showed how the essential parameters of the combinatorial auctions (distributions used for generating problem instances, numbers of items and bids) and of the decomposition method (the width and size of the constructed hypertree) influence the performance of the algorithm. The results and analysis of our experimental tests, as well as a comparison of $\mathsf{ComputeSetPacking_k}$ with the

other approaches for different distributions with respect to their hardness to be solved, are also presented within the thesis.

This work is organized according to the following chapters:

- Chapter 2, *Basic Concepts*, presents basic formal notations and definitions, as well as concepts and well-known results of theories that are used within the context of this thesis.
- Chapter 3, *Solving the Winner Determination Problem for Combinatorial Auctions*, talks about the complexity of the problem and presents some approaches to solve it. One of them is the algorithm $\mathsf{ComputeSetPacking_k}$ [11], which was implemented in this thesis.
- Chapter 4, $\mathsf{ComputeSetPacking_k}$ *Implementation*, includes the description of algorithms used by us to construct the required input for $\mathsf{ComputeSetPacking_k}$ and shows how the essential steps of $\mathsf{ComputeSetPacking_k}$ are carried out.
- Chapter 5, *Experimental Evaluation*, firstly describes existing bid generation techniques, either parameterized by number of bids and number of items [5, 10, 24] or a suite for distributions based on real-world situations [20]. Then the chapter presents our experimental results, when $\mathsf{ComputeSetPacking_k}$ was executed on benchmark instances generated with some of these distributions.

Chapter 2

Basic Concepts

In this chapter we present basic formal notations and definitions, as well as concepts of theories that are used within the context of this thesis. First, we formally describe what a combinatorial auction is and present the main challenge in combinatorial auctions theory - finding efficient solutions for the combinatorial auction winner determination problem. The latter can be viewed as a constraint satisfaction problem, which is also defined in the chapter. Moreover, we present basics of graph and hypergraph theories, as well as the framework of decomposition methods which are involved in the approach used by us to cope with the combinatorial auction winner determination problem.

2.1 Combinatorial Auctions (CA)

A combinatorial auction is an auction in which various items are sold and bidders are allowed to bid for any combination of items at once. This is essential when a bidder considers the utility of a set of items to be different from the simple sum of the utilities of the items considered separately. An auctioneer is interested in finding the most profitable outcome for a combinatorial auction. However, this task is known to be NP-complete [23, 24].

In the following subsections we present formal definitions of combinatorial auctions and their winner determination problem. The definitions there are based on the definitions given in [11].

2.1.1 Basic Definitions of Combinatorial Auctions

Definition 2.1.1. (Combinatorial Auction, CA) A *combinatorial auction* is a pair $\langle \mathcal{I}, \mathcal{B} \rangle$, where $\mathcal{I} = \{I_1, \ldots, I_m\}$ is a set of all items in the auction, $\mathcal{B} = \{B_1, \ldots, B_n\}$ is the set of all bids in the auction. Each bid B_i is a tuple $\langle S_i, p_i \rangle$, where S_i is a set of items from $\mathcal{I}$ and $p_i \geq 0$ is a real number which stands for the price offered by a buyer for S_i.

Notation 2.1.1. $S : \mathcal{B} \rightarrow 2^{\mathcal{I}}$, is a function s.t. $S(B_i) = S(\langle S_i, p_i \rangle) = S_i$

Notation 2.1.2. $p : \mathcal{B} \rightarrow \mathbb{R}^+$, is a function s.t. $p(B_i) = p(\langle S_i, p_i \rangle) = p_i$

Definition 2.1.2. (Outcome for CA) An *outcome* for a combinatorial auction $\langle \mathcal{I}, \mathcal{B} \rangle$ is a set $\mathcal{U} \subseteq \mathcal{B}$ in which all bids are pairwise disjoint sets, i.e. if ($B_i \in \mathcal{U}$ and $B_j \in \mathcal{U}$) then $S(B_i) \cap S(B_j) = \emptyset$ for each B_i, B_j s.t. $B_i \neq B_j$.

Note that there may be many outcomes for a combinatorial auction.

Notation 2.1.3. We denote the set of all outcomes for a combinatorial auction as $\mathbf{U}$.

Definition 2.1.3. (Revenue for CA) Let $\mathcal{A}$, s.t. $\mathcal{A} = \langle \mathcal{I}, \mathcal{B} \rangle$, be a combinatorial auction. The *revenue* $Rev(\mathcal{U})$ of an outcome $\mathcal{U}$ of $\mathcal{A}$ is the real number calculated as follows:

$$Rev(\mathcal{U}) = \sum_{B_i \in \mathcal{U}} p(B_i)$$

2.1.2 Winner Determination Problem (WDP) for Combinatorial Auctions

In a combinatorial auction a seller is accepting different bids and his objective is to determine a set of mutually disjoint bids (an outcome) that will bring him the maximum revenue over all possible outcomes. In this respect, the seller is interested in finding the solution for the so-called winner determination problem.

Definition 2.1.4. (Winner Determination Problem, WDP) Let $\mathcal{A}$, s.t. $\mathcal{A} = \langle \mathcal{I}, \mathcal{B} \rangle$, be a combinatorial auction. The *winner determination problem* of $\mathcal{A}$ is the problem of finding a set $\mathcal{W}$ such that:

1. $\mathcal{W} \subseteq \mathcal{B}$
2. $\mathcal{W} \in \mathbf{U}$, where $\mathbf{U}$ is the set of outcomes for $\mathcal{A}$
3. $Rev(\mathcal{W}) = \max\limits_{\mathcal{U}_i \in \mathbf{U}} Rev(\mathcal{U}_i)$

This problem was proven to be intractable, more precisely NP-complete [23]. There has been much interest in developing approximate algorithms for combinatorial auction winner determination [10]. However, the problem is not approximable in polynomial time (unless NP=ZPP) [24]. In Chapter 3 we discuss this question in more details.

2.2 Graphs

Graphs are mathematical structures that are helpful for modelling pairwise relations between objects from a certain collection. In this section we present some concepts from graph theory that are usefull within the context of this thesis.

2.2.1 Basic Definitions of Graphs

Definition 2.2.1. (Graph) An *undirected graph* G is a pair $\langle V, E\rangle$, where V is a finite set of vertices, and E is a set of unordered pairs of vertices (edges): $E = \{\{v_1, v_2\} \mid v_1, v_2 \in V\}$.

If the graph is undirected, the adjacency relation defined by the arcs is symmetric.

Definition 2.2.2. (Path) Let G, s.t. $G = \langle V, E\rangle$, be an undirected graph. A sequence $\langle\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \ldots, \{v_{k-1}, v_k\}\rangle$, where $\{v_i, v_{i+1}\} \in E$ for all $i \in [1, k-1]$, is a *path* between $v_1, v_k \in V$.

Definition 2.2.3. (Connected Graph) G is *connected* iff for any $v_i, v_j \in V$ there exists a path between v_i and v_j.

Definition 2.2.4. (Acyclic Graph) G is *acyclic* iff there is no path in G that starts and ends at the same vertex $v \in V$.

Definition 2.2.5. (Tree) A *tree* is a connected, undirected, acyclic graph.

2.2.2 Tree Decompositions of Graphs

The concept of tree decompositions was introduced by Robertson and Seymour in their work on graph minors [22].

Definition 2.2.6. (Tree Decomposition of a Graph) Let G, s.t. $G = \langle V, E\rangle$, be a graph. A *tree decomposition* of G is a tuple $\langle T, \chi\rangle$, where T is a tree, s.t. $T = \langle N, E\rangle$, and χ is a function, s.t. $\chi : N \rightarrow 2^V$, which satisfies all the following conditions:

1. $\bigcup_{t \in N} \chi(t) = V$
2. for all $\{v, w\} \in E$ there exists a $t \in N$ that $v \in \chi(t)$ and $w \in \chi(t)$
3. for all $i, j, t \in N$ if t is on the path from i to j in T, then $\chi(i) \cap \chi(j) \subseteq \chi(t)$

The first condition ensures that each vertex of G occurs in the χ-set of at least one node of T. The second condition says that each edge of G is contained within the χ-set of some node of T. The last condition ensures that for each vertex v of the graph the set of vertices $\{t \in N \mid v \in \chi(t)\}$ forms a connected subtree of T.

Definition 2.2.7. (Width of a Tree Decomposition of a Graph) The *width of a tree decomposition* $\langle \chi, T\rangle$, where $T = \langle N, E\rangle$, is $\max_{t \in N} \mid \chi(t) \mid -1$.

Definition 2.2.8. (Tree Width of a Graph) The *tree-width* of a graph G ($tw(G)$) is the minimum width over all tree decompositions of G.

The notion of treewidth can be used as a characterization of graph acyclicity. In particular, a graph is acyclic if and only if its treewidth is equal to one [22].

Definition 2.2.9. (Structured Item Graph) A graph with the tree-width bounded by some natural number k ($tw(G) \leq k$) is called *structured item graph.*

2.3 Hypergraphs

A hypergraph is a generalization of a graph, where an edge not only may be two-element subset of the set of vertices, but also can consist of any subset of the set of vertices. This notion is used for the structural description of a large number of important problems, such as constraint satisfaction problems or conjunctive queries. The notion of hypergraphs is also suitable for the representation of combinatorial auctions, since a combinatorial auction winner determination problem can be viewed as a constraint satisfaction problem.

Moreover, the representation of a combinatorial auction as a hypergraph is useful to see the analogy between the winner determination problem for combinatorial auctions and the maximum weighted set packing problem (as described in Subsection 2.3.3). To make this analogy more intuitive, we sometimes refer to the vertices of hypergraphs as I_i, standing for "items", and to the hyperedges as B_i, standing for "bids".

Most of the definitions in this section are based on the definitions given in [11, 12, 13].

2.3.1 Basic Definitions of Hypergraphs

Definition 2.3.1. (Hypergraph) A *hypergraph* $\mathcal{H}$ is a pair $\langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V} = \{v_1 \dots v_n\}$ is a nonempty set of vertices, and $\mathcal{E} = \{e_1 \dots e_m\}$ is a set of subsets of $\mathcal{V}$, i.e. $\mathcal{E} \subseteq 2^{\mathcal{V}} \setminus \emptyset$. Each element of $\mathcal{E}$ is called the *hyperedge* of $\mathcal{H}$.

Note, that any graph may be defined as a hypergraph, in which every hyperedge has two elements.

Definition 2.3.2. (Dual Hypergraph) Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, be a hypergraph. The *dual hypergraph* $\widetilde{\mathcal{H}}$ of $\mathcal{H}$ is a hypergraph $\langle \widetilde{\mathcal{V}}, \widetilde{\mathcal{E}} \rangle$, such that

1. there is a bijective mapping $\eta : \mathcal{E} \to \widetilde{\mathcal{V}}$
2. there is a bijective mapping $\mu : \mathcal{V} \to \widetilde{\mathcal{E}}$
3. for all $v \in \mathcal{V}$, $\mu(v) = \{\eta(e) | v \in e, e \in \mathcal{E}\}$

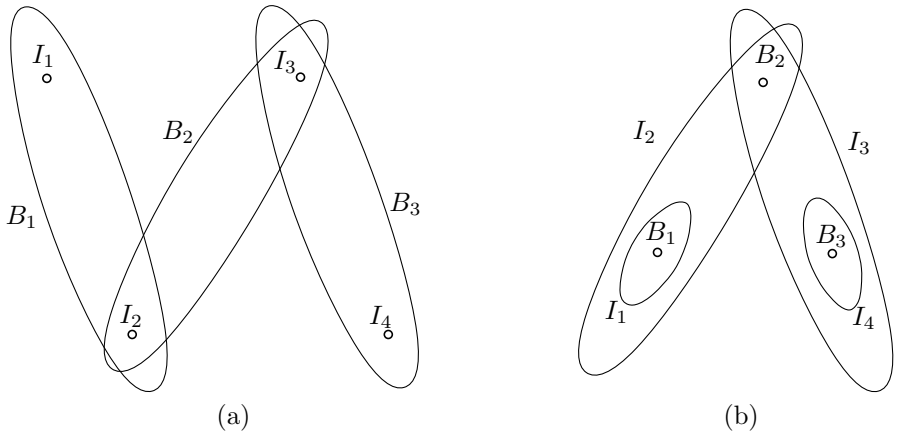


Figure 2.1: Example of a hypergraph and its dual hypergraph: (a) Hypergraph $\mathcal{H}$; (b) Dual hypergraph $\widetilde{\mathcal{H}}$ for $\mathcal{H}$

Intuitively, in the construction of the dual hypergraph, we may think that items of the original hypergraph become bids and bids of the original hypergraph become items. To uphold this intuition we assume throughout the thesis that μ and η are name-preserving functions, i.e. the vertices of the dual hypergraph have the same name as the corresponding edges of the original hypergraph, and the edges of the dual hypergraph have the same name as the corresponding vertices of the original hypergraph. However, the reader should keep in mind that, according to Definition 2.3.2, the vertices and hyperedges of the dual hypergraph are images of, respectively, the hyperedges and vertices of the original hypergraph.

Example 2.1. *An example of a hypergraph $\mathcal{H}$, such that $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V} = \{I_1, I_2, I_3, I_4\}$ and $\mathcal{E} = \{B_1, B_2, B_3\}$, is presented in Figure 2.1(a). The dual hypergraph $\widetilde{\mathcal{H}}$ for $\mathcal{H}$, such that $\widetilde{\mathcal{H}} = \langle \widetilde{\mathcal{V}}, \widetilde{\mathcal{E}} \rangle$, is given in Figure 2.1(b). Note that $\widetilde{\mathcal{V}} = \{B_1, B_2, B_3\}$ and $\widetilde{\mathcal{E}} = \{I_1, I_2, I_3, I_4\}$.*

To a hypergraph may be associated a primal graph, which is a graph having the same set of vertices and containing an edge between any pair of vertices that appear in the same hyperedge of the hypergraph.

Definition 2.3.3. (Primal Graph) Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, be a hypergraph. The *primal graph* $\bar{G}_{\mathcal{H}}$ for $\mathcal{H}$ is the graph $\langle V, E \rangle$, where

1. $V = \mathcal{V}$
2. $E = \{\{v, w\} \mid v \neq w \text{ and } \exists e \in \mathcal{E} \text{ s.t. } v, w \in e\}$

Note that two different hypergraphs may have the same primal graph as it happens for the two graphs in Figure 2.2.

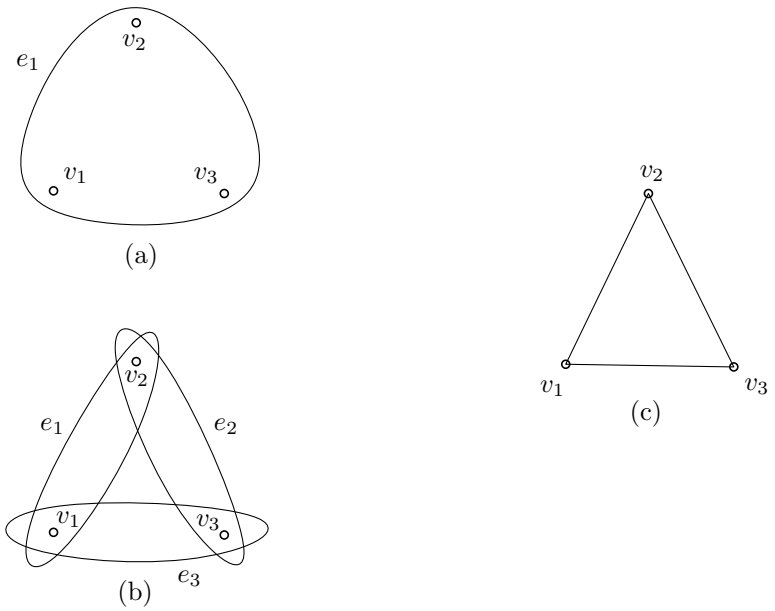


Figure 2.2: Example of a primal graph: (a) Hypergraph $\mathcal{H}_{(a)}$; (b) Hypergraph $\mathcal{H}_{(b)}$; (c) Primal graph for $\mathcal{H}_{(a)}$ and for $\mathcal{H}_{(b)}$

Definition 2.3.4. (Connected Hypergraph) A hypergraph is *connected* if its primal graph consists of a single connected component.

Definition 2.3.5. (Reduced Hypergraph) A hypergraph $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, is *reduced* if there is no $e_1, e_2 \in \mathcal{E}$, s.t. $e_1 \subseteq e_2$.

Definition 2.3.6. (Item Graph) Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, be a hypergraph. An *item graph* $\dot{G}_{\mathcal{H}}$ for $\mathcal{H}$ is the graph $\langle V, E \rangle$, where

1. $V = \mathcal{V}$
2. E are s.t. for any $e \in \mathcal{E}$ the vertices occurring in e induce a connected subgraph of $\dot{G}_{\mathcal{H}}$

Therefore, the primal graph $\bar{G}_{\mathcal{H}}$ for a given hypergraph $\mathcal{H}$ is a particular type of item graph for $\mathcal{H}$. Moreover, any item graph for $\mathcal{H}$ may be constructed from its primal graph by eventual deleting some edges preserving the connectedness condition. Note that there may be many item graphs associated to a given hypergraph. This is demonstrated in Figure 2.3.

Example 2.2. *Consider the hypergraph $\mathcal{H}$ from Figure 2.3(a). Its primal graph is shown in Figure 2.3(b). To obtain an item graph for $\mathcal{H}$, it is possible to remove either the edge $\{v_1, v_3\}$ from the primal graph, resulting in the item graph shown in Figure 2.3(c), or the edge $\{v_2, v_3\}$, resulting in the item graph shown in Figure 2.3(d).*

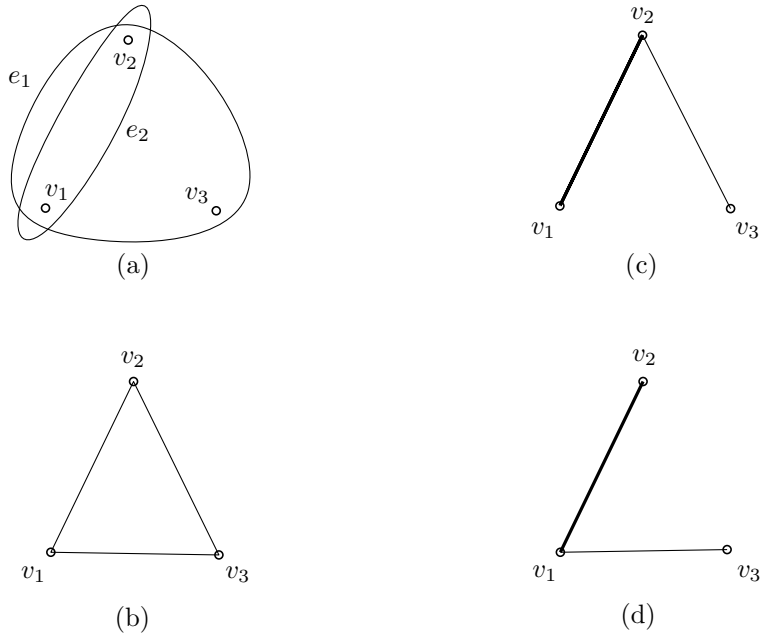


Figure 2.3: Example of item graphs: (a) Hypergraph $\mathcal{H}$; (b) Primal graph for $\mathcal{H}$; (c,d) Two item graphs for $\mathcal{H}$

Definition 2.3.7. (Join Graph) Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, be a hypergraph. A *join graph* $\acute{G}_{\mathcal{H}}$ for $\mathcal{H}$ is an item graph $\langle V, E \rangle$ of the dual hypergraph of $\mathcal{H}$.

Definition 2.3.8. (Join Tree) Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, be a hypergraph. A *join tree* for $\mathcal{H}$ is a join graph for $\mathcal{H}$ that is a tree.

It is important to note, that the notion of join trees may be used to determine if a hypergraph is acyclic: a hypergraph is acyclic iff it has a join tree [2, 13].

Example 2.3. *Recall the hypergraph from Figure 2.2(b). It is equivalent to its dual hypergraph, and it is not acyclic since it does not have a join tree. The hypergraph from Figure 2.1(a) has only one join tree. Note that a hypergraph can have more than one join tree, if its dual hypergraph has more than one item graphs - see Figure 2.3.*

2.3.2 Hypertree Decompositions (HTD) of Hypergraphs

Definition 2.3.9. (Hypertree for a Hypergraph) Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, be a hypergraph. A *hypertree for a hypergraph* $\mathcal{H}$ is a triple $\langle T, \chi, \lambda \rangle$, where $T = \langle N, E \rangle$ is a rooted tree and χ and λ are labelling functions which associate two sets $\chi(t) \subseteq \mathcal{V}$ and $\lambda(t) \subseteq \mathcal{E}$ to each node $t \in N$, i.e.:

$$\chi : N \to 2^{\mathcal{V}}$$
$$\lambda : N \to 2^{\mathcal{E}}$$

Notation 2.3.1. For every node $t \in N$ we denote the rooted subtree of T with root t as T_t.

Definition 2.3.10. (Hypertree Decomposition, HTD) Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, be a hypergraph. A hypertree decomposition of $\mathcal{H}$ is a hypertree $\mathcal{T}$, s.t. $\mathcal{T} = \langle T, \chi, \lambda \rangle$ and $T = \langle N, E \rangle$, for $\mathcal{H}$ which satisfies all the following conditions:

1. for each hyperedge $e \in \mathcal{E}$ there exists $t \in N$ such that $e \subseteq \chi(t)$ (t *covers* e)
2. for each vertex $v \in \mathcal{V}$, the set $\{t \in N \mid v \in \chi(t)\}$ induces a (connected) subtree of T
3. for each $t \in N$, $\chi(t) \subseteq (\bigcup_{e \in \lambda(t)} e)$
4. for each $t \in N$, $(\bigcup_{e \in \lambda(t)} e) \cap (\bigcup_{t_j \in T_t} \chi(t_j)) \subseteq \chi(t)$

The first condition says that each hyperedge of $\mathcal{H}$ is covered by (the χ-set of) at least one node of the hypertree decomposition $\langle T, \chi, \lambda \rangle$. The second condition requires that in every node t of T every vertex from $\chi(t)$ is covered by at least one hyperedge from $\lambda(t)$. The third condition imposes the connectedness condition one the hypertree T with respect to vertices from the χ-sets. The last condition requires that for any subtree T_t of T a vertex appearing in the χ-set of any node of T_t must also appear in $\chi(t)$, unless it does not appear in any edge from $\lambda(t)$.

Note that, if we consider only the first two conditions in the definition of hypertree decompositions above, we will have the definition of *tree decomposition of a hypergraph* (analogous to the one for graphs). If we omit the fourth condition from the definition we will get the definition of *generalized hypertree decomposition.* In the following example we illustrate a generalized hypertree decomposition for the hypergraph from Figure 2.1(b).

Example 2.4. *Consider the hypergraph in Figure 2.1(b) and its decomposition in Figure 2.4. In the example we refer to the root of the hypertree as t and to its only child as n. There is a vertex B_1, such that*

- $B_1 \in I_2$ *and* $I_2 \in \lambda(t)$
- $B_1 \in \chi(n)$
- $B_1 \notin \chi(t)$

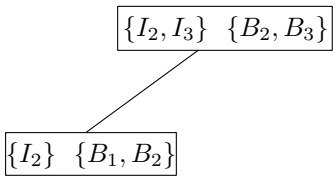


Figure 2.4: Example of a generalized hypertree decomposition for the hypergraph $\widetilde{\mathcal{H}}$ from Figure 2.1(b)

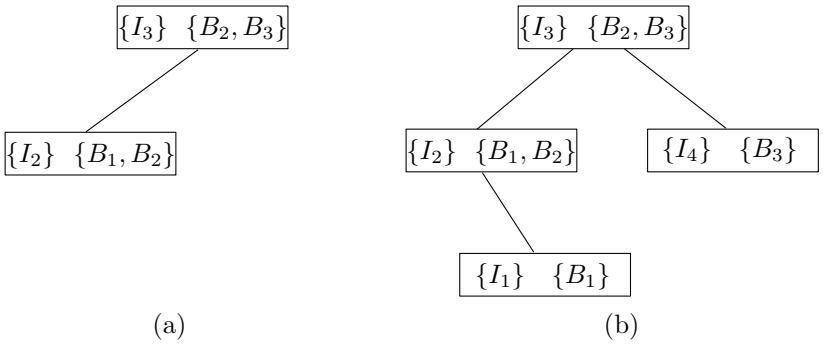


Figure 2.5: Example of a hypertree decomposition and a complete hypertree decomposition for the hypergraph $\widetilde{\mathcal{H}}$ from Figure 2.1(b): (a) HTD; (b) Complete HTD

Therefore, while the first three conditions in the definition of hypertree decomposition are satisfied, the fourth condition is violated. Hence, Figure 2.4 shows a generalized hypertree decomposition for the hypergraph. Note that, if we remove the hyperedge I_2 from the λ-set of the root t, we will get the hypertree decomposition shown in Figure 2.5(a).

Definition 2.3.11. (Strongly Covered Hyperedge in HTD) Given a hypergraph $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, and its hypertree decomposition $\mathcal{T}$, s.t. $\mathcal{T} = \langle T, \chi, \lambda \rangle$ and $T = \langle N, E \rangle$. A hyperedge $e \in \mathcal{E}$ is *strongly covered* in $\mathcal{T}$ if there exists $t \in N$ such that $e \subseteq \chi(t)$ and $e \in \lambda(t)$ (t strongly covers e).

Definition 2.3.12. (Complete HTD) A hypertree decomposition $\mathcal{T}$ of hypergraph $\mathcal{H}$ is a *complete decomposition* of $\mathcal{H}$ if every edge of $\mathcal{H}$ is strongly covered in $\mathcal{T}$.

Example 2.5. *An example of a complete hypertree decomposition for a hypergraph from Figure 2.1(b) is reported in Figure 2.5(b).*

Definition 2.3.13. (Width of HTD) The *width of a hypertree decomposition* $\langle T, \chi, \lambda \rangle$ of $\mathcal{H}$ is $\max_{t \in N} | \lambda(t) |$.

Definition 2.3.14. (Hypertree Width) The hypertree width $hw(\mathcal{H})$ of $\mathcal{H}$ is the minimum width over all its hypertree decompositions.

Note that acyclic hypergraphs are exactly hypergraphs that have hypertree width equal to one.

Example 2.6. *Figure 2.1 shows acyclic hypergraphs only.*

Now we would like to draw attention to a feature of hypertree decompositions that plays an important role in finding a decomposition of minimal width for a hypergraph $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$: as soon as a hyperedge $e \in \mathcal{E}$ has been covered by the χ-set of some node t of the hypertree decomposition, any subset of variables of e may be used for decomposing the remaining cycles of the hypergraph. The intuition is that the use of a subset of vertices in e preserves the connectedness condition while minimizing the size of the λ-sets for each node of the hypertree.

2.3.3 Maximum-Weighted Set Packing Problem

Definition 2.3.15. (Weighted Hyperedge) Let $e \in \mathcal{E}$ be a hyperedge of a hypergraph $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$. A *weighted hyperedge* $\widehat{e}$ for e is a tuple $\langle e, w \rangle$, where $w \in \mathbb{R}^+$ stands for the weight of e.

Notation 2.3.2. $e : \widehat{\mathcal{E}} \to 2^{\mathcal{V}}$, is a function s.t. $e(\widehat{e}_i) = e(\langle e_i, w_i \rangle) = e_i$

Notation 2.3.3. $w : \widehat{\mathcal{E}} \to \mathbb{R}^+$, is a function s.t. $w(\widehat{e}_i) = w(\langle e_i, w_i \rangle) = w_i$

Notation 2.3.4. We denote a set of weighted hyperedges as $\widehat{\mathcal{E}}$, i.e. $\widehat{\mathcal{E}} = \{\widehat{e}_1, \ldots, \widehat{e}_n\}$, where $\widehat{e}_i$ are weighted hyperedges.

Definition 2.3.16. (Weighted Hypergraph) A *weighted hypergraph* $\widehat{\mathcal{H}}$ is a pair $\langle \mathcal{V}, \widehat{\mathcal{E}} \rangle$, where $\mathcal{V} = \{v_1 \ldots v_n\}$ is a nonempty set of vertices, and $\widehat{\mathcal{E}} = \{\widehat{e}_1 \ldots \widehat{e}_m\}$ is a set of tuples, where for all $i \in \{1, 2, \ldots, m\}$ $\widehat{e}_i = \langle e_i, w_i \rangle$, $e_i \in 2^{\mathcal{V}} \setminus \emptyset$ and $w_i \in \mathbb{R}^+$ is the weight of e_i.

Definition 2.3.17. (Packing) Let $\widehat{\mathcal{H}}$, s.t. $\widehat{\mathcal{H}} = \langle \mathcal{V}, \widehat{\mathcal{E}} \rangle$, be a (weighted) hypergraph. A *packing* $\mathcal{P}$ for $\widehat{\mathcal{H}}$ is a set $\mathcal{P} \subseteq \widehat{\mathcal{E}}$ in which all sets of items are pairwise disjoint, i.e. for each pair $P_i, P_j \in \mathcal{P}$ s.t. $P_i \neq P_j$, it holds that $e(P_i) \cap e(P_j) = \emptyset$

Notation 2.3.5. We denote the set of all packings for a hypergraph as $\mathbf{P}$.

Definition 2.3.18. (Conforming Packings) Let $\mathcal{P}^n$ and $\mathcal{P}^c$ be two packings for $\mathcal{H}^n$ and $\mathcal{H}^c$ respectively, where $\mathcal{H}^n = \langle \mathcal{V}^n, \mathcal{E}^n \rangle$ and $\mathcal{H}^c = \langle \mathcal{V}^c, \mathcal{E}^c \rangle$ are two (weighted) hypergraphs. $\mathcal{P}^n$ *conforms with* $\mathcal{P}^c$, denoted $\mathcal{P}^n \looparrowright \mathcal{P}^c$, if

1. for each $P \in \mathcal{P}^c \cap \mathcal{E}^n$, $\quad P \in \mathcal{P}^n$
2. for each $P \in \mathcal{E}^c \setminus \mathcal{P}^c$, $\quad P \notin \mathcal{P}^n$

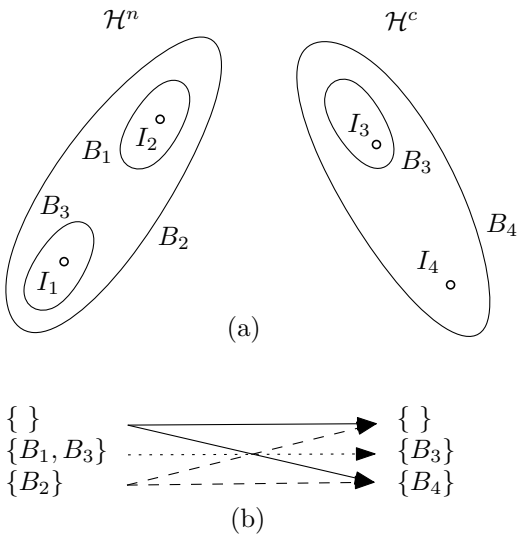


Figure 2.6: Example of conforming and non-conforming packings: (a) Hypergraphs $\mathcal{H}^n$ and $\mathcal{H}^c$; (b) Packings for $\mathcal{H}^n$ and for $\mathcal{H}^c$

Example 2.7. *Consider the two hypergraphs $\mathcal{H}^n$ and $\mathcal{H}^c$ shown in Figure 2.6(a). Some possible packings for $\mathcal{H}^n$ and for $\mathcal{H}^c$ respectively are shown in 2.6(b). An arrow coming from a packing $\mathcal{P}^n$ to a packing $\mathcal{P}^c$ shows that $\mathcal{P}^n \looparrowright \mathcal{P}^c$. Different styles are used to distinguish arrows coming from different packings.*

Definition 2.3.19. (Weight of Packing) Let $\widehat{\mathcal{H}}$ s.t. $\widehat{\mathcal{H}} = \langle \mathcal{V}, \widehat{\mathcal{E}} \rangle$, be a weighted hypergraph. The *weight of packing* $\mathcal{P}$ for $\widehat{\mathcal{H}}$ is the rational number $w(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} w(P_i)$.

The maximum weighted set packing problem for a hypergraph $\widehat{\mathcal{H}}$ is the problem of finding a packing for $\widehat{\mathcal{H}}$ that has the maximum weight over all possible packings for $\widehat{\mathcal{H}}$. Formally:

Definition 2.3.20. (Maximum-Weighted Set Packing Problem) Let $\widehat{\mathcal{H}}$, s.t. $\widehat{\mathcal{H}} = \langle \mathcal{V}, \widehat{\mathcal{E}} \rangle$, be a weighted hypergraph. The maximum-weighted set packing problem for $\widehat{\mathcal{H}}$ is the problem of finding $\mathcal{M} \subseteq \widehat{\mathcal{E}}$ such that:

1. $\mathcal{M} \in \mathbf{P}$, where $\mathbf{P}$ is the set of all packings for $\widehat{\mathcal{H}}$
2. $w(\mathcal{M}) = \max_{\mathcal{P}_i \in \mathbf{P}} w(\mathcal{P}_i)$

To see that the maximum-weighted set packing problem for a hypergraph $\widehat{\mathcal{H}}$, s.t. $\widehat{\mathcal{H}} = \langle \mathcal{V}, \widehat{\mathcal{E}} \rangle$, is just another formulation for the winner determination problem for a combinatorial auction $\mathcal{A}$, s.t. $\mathcal{A} = \langle \mathcal{I}, \mathcal{B} \rangle$, take

- $\mathcal{V} = \mathcal{I}$
- $\widehat{\mathcal{E}} = \mathcal{B}$

In this settings, the set of the solutions for the maximum weighted set packing problem for $\widehat{\mathcal{H}}$ with weighting function w coincides with the set of the solutions for the winner determination problem on $\mathcal{A}$.

2.4 Constraint Satisfaction Problems (CSP)

Many well-known problems in Computer Science and Mathematics, such as graph-colorability, eight queens puzzle, the Sudoku solving problem, the boolean satisfiability problem, can be formulated as constraint satisfaction problems. The winner determination problem for combinatorial auctions can also be formulated as a constraint satisfaction problem. Informally, constraint satisfaction is the process of finding a solution to a problem which has a set of constraints on the values and combinations of variables.

Many of the formal definitions and discussions in this subsection are based on the definitions given in [7, 13].

Definition 2.4.1. (Constraint) A *constraint* C is a pair $\langle S, R\rangle$, that consists of a relation R defined on a finite sequence of variables S (S is called *the constraint scope*).

Intuitively, a constraint is a rule that says which of the variable assignments are legal.

Definition 2.4.2. (Constraint Network) A *constraint network* $\mathcal{N}$ is a triple $\langle Var, \mathcal{D}, \mathcal{C}\rangle$, where

1. $Var = \{v_1, \ldots, v_n\}$ is a finite set of *variables*
2. $\mathcal{D} = \{D_{v_1}, \ldots, D_{v_n}\}$ is a set of *domains*, such that each variable $v_i \in Var$ has a nonempty domain D_{v_i} of its possible values, such that $D_{v_i} \in \mathcal{D}$
3. $\mathcal{C} = \{C_1, \ldots, C_t\}$ is a set of constraints. Each constraint C_i is defined on a subset of variables S_i, where $S_i \subseteq Var$

Note that in the constraint $\langle S_i, R_i\rangle$, where $S_i = \langle v_1, \ldots, v_r\rangle$, the relation R_i is a subset of the Cartesian product $D_{v_1} \times \cdots \times D_{v_r}$.

Definition 2.4.3. (Variable Instantiation) *Variable instantiation* is a mapping

$$\nu : Var' \to \bigcup_{v \in Var'} D_v$$

s.t. $Var' \subseteq Var$ and D_v is the domain of v for each $v \in Var'$.

In other words ν is a partial function from Var to the corresponding domains of the variables in Var. A variable v is *instantiated* by a mapping ν if v belongs to Var'.

Example 2.8. *Consider, for example, a set of variables* $Var = \{v_1, v_2, v_3\}$, *with their corresponding domains* $D_{v_1} = \{0, 2\}$, $D_{v_2} = \{1, 2\}$, $D_{v_3} = \{1, 2, 4\}$. *Then some of the possible instantiations of variables from* Var *are* $\nu(v_1) = 0$, $\nu(v_2) = 2$, $\nu(v_3) = 4$.

We say that a set V of variables is instantiated if each variable from V is instantiated.

Definition 2.4.4. (Satisfying a Constraint) An instantiation ν of a set of variables $Var = \{v_1, \ldots, v_k\}$ *satisfies a constraint* $\langle S, R\rangle$ iff

1. $v \in Var$ for each variable $v \in S$
2. there exist a tuple $t \in R$, s.t. for every variable $v \in S \quad t(v) = \nu(v)$

Example 2.9. *Recall Example 2.8. Consider a relation* $R = \{\langle 0, 1\rangle, \langle 0, 2\rangle\}$ *over variables* $\langle v_1, v_2\rangle$. ν *satisfies* R, *because* $\{v_1, v_2\} \in Var$ *and there is a tuple* $t = \langle 0, 2\rangle$ *in* R, *s.t.* $t(v_1) = \nu(v_1) = 0$ *and* $t(v_2) = \nu(v_2) = 2$.

An instantiation that does not violate any constraint is called *consistent* or legal. A *complete* instantiation is one in which every variable from the set Var is mentioned. A solution is a complete and consistent instantiation of the variables.

Definition 2.4.5. (Solution to a Constraint Network). Let $\mathcal{N}$ be a constraint network, such that $\mathcal{N} = \langle Var, \mathcal{D}, \mathcal{C}\rangle$. A *solution* to $\mathcal{N}$ is an instantiation ν of all variables from Var that satisfies all the constraints from $\mathcal{C}$.

Often the concepts of "constraint network" and "constraint satisfaction problem" are used interchangeably. In fact, however, solving a constraint satisfaction problem is a task over the constraint network $\mathcal{N}$, such as determining whether a solution to $\mathcal{N}$ exists and, if yes, finding this solution.

In general, constraint satisfiability is known to be NP-hard [7]. One of the approaches to identify tractable classes of constraint satisfaction problems is to consider structural properties of the constraint scopes. In the general case these properties can be formalized as graph-theoretic properties of the constraint hypergraph [13].

The constraint hypergraph of a constraint network is the hypergraph such that its vertices are the variables of the network and its hyperedges are the sets of those variables which appear together in a constraint scope:

Definition 2.4.6. (Constraint Hypergraph) Let $\mathcal{N}$, s.t. $\mathcal{N} = \langle Var, \mathcal{D}, \mathcal{C}\rangle$, be a constraint network. The *constraint hypergraph* of $\mathcal{N}$ is a hypergraph $\langle \mathcal{V}, \mathcal{E}\rangle$ such that:

1. $\mathcal{V} = Var$
2. $\mathcal{E} = \{S \mid C = \langle S, R\rangle, C \in \mathcal{C}\}$

A constraint network is acyclic if its constraint hypergraph is acyclic. Acyclic constraint networks are polynomially solvable [6].

2.5 Decomposition Methods for Constraint Satisfaction Problems

As already mentioned in Section 2.4, constraint satisfaction problems which associated constraint hypergraphs are acyclic can be solved efficiently. In this section we first present a general framework of methods for decomposing a cyclic constraint satisfaction problems into acyclic ones. Then we discuss the idea of CSP-solving algorithms based on hypertree decompositions, since the hypertree decomposition method presents a very promising approach for identifying and solving tractable classes of constraint satisfaction problems.

2.5.1 General Framework of Decomposition Methods for Constraint Satisfaction Problems

It is well known that *acyclic* constraint satisfaction problems are polynomially solvable [6]. Many constraint satisfaction problems happen to be *nearly acyclic*. Among these are problems that are not acyclic, but can be transformed to equivalent acyclic problems by simple operations, and problems with corresponding hypergraphs containing either few or small cycles [13]. Hence, there has been much research in artificial intelligence focused on developing techniques for decomposing (nearly) cyclic constraint satisfaction problems into acyclic ones, thus identifying tractable classes of constraint satisfaction problems.

The general formal framework of decomposition methods was introduced in [13].

Let $\mathcal{H}$, s.t. $\mathcal{H} = \langle \mathcal{V}, \mathcal{E}\rangle$, be a constraint hypergraph. Assume w.l.o.g. that every vertex from $\mathcal{V}$ appears in at least one hyperedge from $\mathcal{E}$. Assume also w.l.o.g. that $\mathcal{H}$ is connected and reduced. A *decomposition method* D associates to $\mathcal{H}$ a parameter D-width($\mathcal{H}$).

Definition 2.5.1. (k-tractable Class of a Decomposition Method) Let D be a decomposition method for a constraint satisfaction problem. For any natural k, the *k-tractable class* $\mathsf{C}(\mathsf{D}, k)$ of D is defined as $\mathsf{C}(\mathsf{D}, k) = \{\mathcal{H} \mid \mathsf{D}\text{-width} \leq k\}$.

Therefore, for every decomposition method, constraint satisfaction problems can be classified according to the following infinite hierarchy of tractable

classes:

$$\mathsf{C}(\mathsf{D},1) \subset \mathsf{C}(\mathsf{D},2) \subset \cdots \subset \mathsf{C}(\mathsf{D},k) \subset \ldots$$

such that for each (cyclic) constraint satisfaction problem $\mathcal{K}$ belonging to class $\mathsf{C}(\mathsf{D},k)$ there exists a decomposition of width $\leq k$, i.e. $\mathcal{K}$ can be transformed in polynomial time (depending on k) into an equivalent acyclic constraint satisfaction problem $\mathcal{K}'$ [13, 15].

The solution to $\mathcal{K}'$ (and therefore $\mathcal{K}$) may be found efficiently by processing the respective join-tree in a bottom-up fashion and performing semi-joins of parent relations with corresponding children relations while ascending. In other words, starting from the leaves, the constraint size associated to each node t of the T may be reduced by filtering the tuples that do not agree on the common attributes with the constraint of some of the child node of t. In this way, at every hypertree decomposition node, we solve a partial problem with respect to $\mathcal{K}'$. If, at any node of T after performing a semi-join, we get an empty relation, then the given CSP instance has no solution. If, at the end, the constraint relation in the root is not empty, then a solution exists, and we may "collect" it traversing the tree in a top-down fashion. This approach was originally introduced in [29].

It is desirable that a decomposition method D satisfies the following properties [12]:

1. It should be as general as possible, i.e. for all k the classes $\mathsf{C}(\mathsf{D},\mathrm{k})$ of D should be as large as possible.

2. It should be polynomially computable, i.e. given a hypergraph $\mathcal{H}$, it should be possible, for a fixed constant k, to decide the existence of a decomposition of width k of $\mathcal{H}$ and compute one (if any exist) in polynomial time.

3. Given a problem, the result of its hypergraph decomposition of bounded width should lead to the polynomial solution of the problem.

There are many decompositions satisfying the last two properties. However, as shown in [13], the method of hypertree decompositions is the most general method satisfying all three properties known so far. The idea of solving constraint satisfaction problems with generalized hypertree decompositions is presented in [12].

2.5.2 Hypertree Decomposition Method for Constraint Satisfaction Problems

Let $\mathcal{H}$ be a hypergraph representation of a constraint satisfaction problem $\mathcal{K}$. Let T, where $\mathcal{T} = \langle T, \chi, \lambda \rangle$ and $T = \langle N, E \rangle$, be a complete hypertree decomposition of $\mathcal{H}$ of width k. To obtain a join-tree of an acyclic hypergraph $\mathcal{H}'$, we define, for each node $t \in N$, a new constraint with the scope

$\chi(t)$, which associated constraint relation is the projection on $\chi(t)$ of the join of the relations in $\lambda(t)$. It takes $O(m^{|\lambda(t)|-1}\log m)$ time to build a fresh constraint for each $t \in N$, where m is the size of the largest relation to be joined. This $\mathcal{H}'$ is a hypergraph corresponding to a new CSP instance $\mathcal{K}'$, which solution is equivalent to the solution of the original $\mathcal{K}$. Then $\mathcal{K}'$ may be solved in $O(n'm^{w-1}\log m)$ time, where w is the decomposition width and n' is the number of nodes in $\mathcal{T}$.

It is important to note that a hypertree with the smallest width for a given constraint satisfaction problem $\mathcal{K}$ gives the best way of putting together constraints of $\mathcal{K}$ in order to obtain an acyclic equivalent instance $\mathcal{K}'$ to be solved efficiently [12].

In fact, the hypertree width is known as a tractability measure for some NP-hard constraint satisfaction problems [14]. Intuitively, the smaller the hypertree width is, the faster the corresponding problem can be solved.

Chapter 3

Solving the Winner Determination Problem for Combinatorial Auctions

The winner determination problem for combinatorial auctions is known to be NP-complete. In this chapter we talk about existing approaches for identifying tractable classes of combinatorial auctions. Additionally, we present existing optimal algorithms for solving the problem [10, 11, 24, 27]. One of these algorithms, $\mathsf{ComputeSetPacking_k}$, is the algorithm implemented and experimentally tested within this thesis, and details about its implementation are left to Chapter 4.

3.1 Complexity of the Winner Determination Problem for Combinatorial Auctions

It is NP-complete to determine the winners in general combinatorial auctions [23]. Numerous attempts to cope with this computational complexity can be found in the literature. Among them, three main directions of developing algorithms for solving the winner determination problem can be singled out [3, 18]:

1. *Designing approximation algorithms* [10]. Unfortunately, there is no polynomial-time algorithm that can give a sufficient approximation [23, 24].

2. *Designing optimal polynomial time algorithms for restricted classes of combinatorial auctions* [23, 26]. The drawback of this approach is that the bidders cannot fully express their preferences.

3. *Designing optimal search algorithms that are often fast, but require exponential time in the worst case (unless P=NP)* [10, 11, 24, 27]. These

algorithms are mainly based on tree search algorithms and decomposition techniques.

In this thesis we focus on the last direction among the directions mentioned above.

3.2 Tractable Classes of Combinatorial Auctions

In this section we give an overview of two approaches for the decomposition of hypergraphs - structured item graphs and hypertree decompositions - aimed to identify tractable classes of combinatorial auctions. Any combinatorial auction instance can be evaluated by both of these notions, since for every hypergraph it is possible to construct its item graph and its hypertree decomposition. However, the hypertree decomposition technique is known to be strictly more general than the technique of item graphs with bounded tree width [11].

The tree-width of an item-graph of the hypergraph corresponding to a combinatorial auction is proposed in [3] as the main complexity parameter for solving the winner determination problem. However, with regard to this notion, two computational problems arise:

- Solving the winner determination problem for a combinatorial auction when an *item graph is given.*
- *Constructing* an item graph with the smallest possible width.

A winner determination problem is polynomially solvable if the item-graph for a combinatorial auction has the tree-width equal to 1 [26]. Moreover, according to [3], a winner determination problem can be solved in polynomial time if an item graph with bounded tree width is given. The second result is practically useful when a structured item graph of small width either is given or can be efficiently determined. Therefore, the question of determining whether a structured item graph of a certain tree-width exists and can be computed in polynomial time is of a particular interest, and has been extensively researched. The difficulty is that there may be exponentially many item graphs for a hypergraph corresponding to a given combinatorial auction.

An algorithm for constructing an item tree with tree-width equal to 1 (if it exists) in polynomial time was presented in [3]; yet the question whether an item graph with small tree-width may be constructed was left open. Another important result from [3] says that constructing an item graph with minimum number of edges is NP-complete.

Polynomial time algorithms for finding the structural item graph with the minimum tree width were shown for the cases when the item graph to be constructed is a line [17], a cycle [9] or a tree [3], yet the crucial

open problem was whether it is tractable to check if there exists an item graph with bounded treewidth for a given combinatorial auction, and how it can be efficiently constructed if it exists. Later it was proved in [11] that determining whether structured item graphs with a given tree-width exist is computationally intractable for tree-width greater or equal to 3.

However, the problems of deciding if a hypertree decomposition of bounded width exists, and computing one are solvable in polynomial time [14]. Hence, hypertree decompositions are considered as a promising concept for solving the combinatorial auction winner determination problem.

According to [11], the winner determination problem is tractable on the class of the instances with corresponding dual hypergraphs having hypertree width bounded by a fixed natural number. Note that the key parameter in this tractability result is the hypertree width of the dual hypergraph $\widetilde{\mathcal{H}}$ for the auction hypergraph $\mathcal{H}$.

Furthermore, the hypertree decomposition technique for solving the winner determination problem proposed in [11] is strictly more general than the technique of structural item graphs, in the sense that strictly larger classes of instances are tractable by the hypertree decomposition approach than by the item graphs with bounded treewidth approach. Therefore, the problem of determining whether a structured item graph with bounded tree-width exists is not due to its generality, but rather because of some specificity in its definition [11].

3.3 Exact Algorithms

There has been much effort in developing algorithms for solving the combinatorial auction winner determination problem. Due to the computational complexity of the problem, the main challenge is to develop algorithms that provably find an optimal solution. This section focuses on today's foremost optimal algorithms for solving the problem [10, 11, 24, 27].

3.3.1 Combinatorial Auction Structured Search (CASS)

Combinatorial Auction Structured Search is a branch-and-bound algorithm presented in [10, 19] that exploits contextual information of an auction. This is achieved by dividing bids into groups, called "bins", in the following way: For every item in $\mathcal{I}$ a bin is created. Then, given an ordering of all items in the auction, CASS distributes bids among bins, such that a bid is put into the bin corresponding to its lowest-order item. Hence, all bids are mutually exclusive in a bin. This division lets the algorithm not only to exclude from the consideration conflicting bids from the same bin, but also to skip entire bins (due to the ordering).

The algorithm performs a depth-first search using a heuristic h for backtracking. It remembers the allocation $\mathcal{L}^{best}$ (a set of compatible bids) with

the highest revenue $Rev(\mathcal{L}^{best})$ found so far. During the search, it finds at every other allocation $\mathcal{L}$ an upper bound $h(\mathcal{L})$ on the weight-revenue that can be obtained with the remained items from the auction. It backtracks when $Rev(\mathcal{L}) + h(\mathcal{L}) \leq Rev(\mathcal{L}^{best})$. Moreover, if an outcome $\mathcal{U}$ is reached CASS remembers it if $Rev(\mathcal{U}) > Rev(\mathcal{L}^{best})$ and backtracks.

Moreover, the algorithm uses a particular caching technique to keep the knowledge from earlier searches to shrink in some cases a value of the upper bound function h.

A more detailed description of the algorithm, its caching scheme, as well as techniques for the construction of the upper-bound and some other heuristics applied in CASS can be found in [10, 19]. Additionally, the C++ source code of the algorithm is publicly available.

3.3.2 Combinatorial Auction Branch On Bids (CABOB)

Combinatorial Auction Branch On Bids [27] is another special-purpose winner determination algorithm. It is a descendant of the Branch On Bids (BOB) algorithm of the same authors [26]. CABOB is a depth-first branch-and-bound tree search algorithm that branches on bids. During the search, the algorithm maintains a graph structure, called "bid graph", which incorporates information about conflicting bids: the vertices of the graph correspond to bids that do not include any already allocated item (i.e. still "available" bids); two vertices of the graph are connected by an edge if the corresponding bids compete for an item. While searching, the algorithm uses upper and lower bounds on the revenue that the non-allocated items can contribute.

Moreover, CABOB partitions bids into individual connected components, such that every item appears only in the bids of at most one of the components; then it uses upper and lower bounding for further pruning across these components.

A more detailed explanation of CABOB, as well as other techniques applied in it, including preprocessing algorithms, description of upper-, lower-bound and bid ordering heuristics can be found in [25, 27]. Unfortunately, CABOB's executables and source code are not publicly available.

3.3.3 CPLEX

CPLEX is a general-purpose mixed integer programming package [1]. Since the winner determination problem for combinatorial auctions can be formulated as a (mixed) integer program, CPLEX is able to obtain optimal solutions for combinatorial auctions. According to [1], "much more general combinatorial auctions than the ones treated so far can be expressed as mixed integer programs, and [...] they can be successfully managed by

[1] see www.cplex.com

standard operations research algorithms and commercially available software".

3.3.4 ComputeSetPacking$_k$

ComputeSetPacking$_k$ is a polynomial-time algorithm proposed in [11] for solving the maximum weight set packing problem (hence, the winner determination problem for combinatorial auctions) on the class of those instances for which the corresponding dual hypergraphs have bounded hypertree width.

A pseudo-code of ComputeSetPacking$_k$ is presented in Algorithms 3.3.1, 3.3.2, 3.3.3 and 3.3.4, and it is given using notations that are typical for the maximum-weighted set packing problem. However, a reader should bear in mind that the maximum set packing problem can be viewed as a constraint satisfaction problem, therefore the algorithm follows the scheme described in Section 2.5.

Subsequently we give a more detailed explanation of the essential parts of the algorithm.

The algorithm receives two arguments in input:

- a weighted hypergraph $\mathcal{H}$
- a complete k-width hypertree decomposition $\mathcal{T}$ of the dual hypergraph $\widetilde{\mathcal{H}}$ of $\mathcal{H}$

Every node n of the hypertree decomposition $\mathcal{T}$ represents a subproblem of the initial problem. To see this, consider a hypergraph $\mathcal{H}^n = \langle \mathcal{V}^n, \mathcal{E}^n \rangle$, where $\mathcal{V}^n = \lambda(n)$ and $\mathcal{E}^n = \chi(n)$. For each such hypergraph $\mathcal{H}^n$ the algorithm constructs a set $\mathbf{P}^n$ of all its possible packings (i.e. partial packings for the initial problem). This set represents a constraint defined in the node n with the scope $\chi(n)$ and tuples of the constraint relation corresponding to every $\mathcal{P} \in \mathbf{P}^n$. The size of $\mathcal{H}^n$ is bounded by $(|\mathcal{E}|+1)^k$, which is imposed by the nature of the original problem: each vertex may be referred to a single hyperedge or left uncovered.

For each packing, its weight is computed. Then the algorithm traverses the hypertree $\mathcal{T}$ twice. The first traversal is called the Bottom-Up phase, and the second traversal is called the Top-Down phase.

In the Bottom-Up phase the algorithm processes nodes of $\mathcal{T}$ from the leaves to the root, and for every node n of $\mathcal{T}$ (except leaves) the algorithm removes from $\mathbf{P}^n$ those packings that do not conform with any packing of any of the child-nodes of n. Afterwards, for each of the remaining packings $\mathcal{P}$ in $\mathbf{P}^n$, the algorithm finds, in the packings of every child node c of n, a packing $best[\mathcal{P}, c]$ conforming with $\mathcal{P}$ and such that $best[\mathcal{P}, c]$ has maximal weight. Furthermore, the weight of $\mathcal{P}$ is updated, in a way that the weight-revenue "brought" by the conforming packing with the maximal

weight among packings of each of the child-nodes is added to the former weight of $\mathcal{P}$.

After the Bottom-Up phase is complete, in the root r of the tree there will be a set of packings $\mathbf{P}^r$, such that every $\mathcal{P}^r \in \mathbf{P}^r$ has a weight $w_{\mathcal{P}^r}$ associated to it. This $w_{\mathcal{P}^r}$ is the maximal weight, which is possible to get using the hyperedges occurring in the considered packing $\mathcal{P}^r$. Moreover, each $\mathcal{P}^r \in \mathbf{P}^r$ has references to the best (in terms of weight-revenue) conforming packing in every child-node of r; and each of those packings has references to the packings that are best conforming with it in all the corresponding child-nodes of the tree, and so on down to the leaves of the tree. It is only left to choose from the root-node r of the tree a packing that has the maximal weight among all other packings $\mathbf{P}^r$, and to traverse the tree in Top-Down fashion, from r down to the leaves, accumulating the best partial packings found.

Algorithm 3.3.1: COMPUTESETPACKING$_{\text{K}}(\mathcal{H}, \mathcal{T}(\widetilde{\mathcal{H}}))$

comment: $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$

comment: $\mathcal{T}(\widetilde{\mathcal{H}}) = \langle T, \lambda, \chi \rangle$

comment: $T = \langle N, E \rangle$

main
for each $n \in N$
do $\begin{cases} \mathcal{H}^n \leftarrow \text{weighted hypergraph s.t. } \mathcal{H}^n = \langle \mathcal{V}^n, \mathcal{E}^n \rangle, \\ \quad \text{where } \mathcal{V}^n \subseteq \mathcal{V} \text{ , } \mathcal{V}^n = \lambda(n) \text{ , } \mathcal{E}^n \subseteq \mathcal{E} \text{ and } \mathcal{E}^n = \chi(n) \\ \mathbf{P}^n \leftarrow \text{set of all packings for } \mathcal{H}^n \\ \textbf{for each } \mathcal{P} \in \mathbf{P}^n \\ \quad \textbf{do } w_{\mathcal{P}} \leftarrow \text{WEIGHT}(\mathcal{P}) \end{cases}$
BOTTOMUP()
let r be the root of T
$\mathcal{P}^r_{best} \leftarrow arg\ max_{\{\mathcal{P} \in \mathbf{P}^r\}} w_{\mathcal{P}}$
$res \leftarrow \mathcal{P}^r_{best}$
TOPDOWN$(r, \mathcal{P}^r_{best})$
return (res)

Algorithm 3.3.2: COMPUTESETPACKING$_{\text{K}}(\mathcal{H}, \mathcal{T}(\widetilde{\mathcal{H}}))$

comment: $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$

comment: $\mathcal{T}(\widetilde{\mathcal{H}}) = \langle T, \lambda, \chi \rangle$

comment: $T = \langle N, E \rangle$

procedure WEIGHT($\mathcal{P}$)
$w \leftarrow \sum_{\mathcal{P}_i \in \mathcal{P}} w(\mathcal{P}_i)$
return (w)

procedure TAKERAWPARENTNODE()
$nodes \leftarrow \{n \mid n \in T,\ n \notin Done$ and $\forall c$ s.t. $\{n, c\} \in E,\ c \in Done\}$
return $(node \in nodes)$

Algorithm 3.3.3: COMPUTESETPACKING$_{\text{K}}(\mathcal{H}, \mathcal{T}(\widetilde{\mathcal{H}}))$

comment: $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$

comment: $\mathcal{T}(\widetilde{\mathcal{H}}) = \langle T, \lambda, \chi \rangle$

comment: $T = \langle N, E \rangle$

procedure BOTTOMUP()
$Done \leftarrow$ the set of all leaves of T
$n \leftarrow$ TAKERAWPARENTNODE()
repeat
for each $c \in \{child \mid child \in N$ and $\{n, child\} \in E\}$
do $\mathbf{P}^n \leftarrow \mathbf{P}^n \setminus \{\mathcal{P} \mid \mathcal{P} \in \mathbf{P}^n$ and $\neg\exists \mathcal{Q} \in \mathbf{P}^c$ s.t. $\mathcal{P} \not\mapsto \mathcal{Q}\}$
for each $\mathcal{P} \in \mathbf{P}^n$
do $\begin{cases} \textbf{for each } c \in \{child \mid child \in N \text{ and } \{n, child\} \in E\} \\ \quad \textbf{do} \begin{cases} best[\mathcal{P}, c] \leftarrow arg\ max_{\{\mathcal{Q} \mid \mathcal{Q} \in \mathbf{P}^c \text{ and } \mathcal{P} \not\mapsto \mathcal{Q}\}}(w_{\mathcal{Q}}) \\ w_{\mathcal{P}} \leftarrow w_{\mathcal{P}} + \text{WEIGHT}(best[\mathcal{P}, c]) - \\ \text{WEIGHT}(best[\mathcal{P}, c] \cap \mathcal{P}) \end{cases} \end{cases}$
$Done \leftarrow Done \cup n$
$n \leftarrow$ TAKERAWPARENTNODE()
until $n = \emptyset$

Algorithm 3.3.4: $\textsc{ComputeSetPacking}_{\textsc{k}}(\mathcal{H}, \mathcal{T}(\widetilde{\mathcal{H}}))$

comment: $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$

comment: $\mathcal{T}(\widetilde{\mathcal{H}}) = \langle T, \lambda, \chi \rangle$

comment: $T = \langle N, E \rangle$

procedure $\textsc{TopDown}(n \in N, \mathcal{P} \in \mathbf{P}^n)$

for each $c \in \{child \mid child \in N \text{ and } \{n, child\} \in E\}$

do $\begin{cases} res \leftarrow res \cup best[\mathcal{P}, c] \\ \textsc{TopDown}(c, best[\mathcal{P}, c]) \end{cases}$

In [11] it is shown that, given a weighted hypergraph $\mathcal{H}$ and a k-width hypertree decomposition $\mathcal{T}$ of the dual hypergraph $\widetilde{\mathcal{H}}$ of $\mathcal{H}$, such that $T = \langle N, E \rangle$, $\mathsf{ComputeSetPacking_k}$ correctly outputs a solution for the corresponding winner determination problem in time $O(|\mathcal{T}| \times (|\mathcal{E}| + 1)^{2k})$.

Moreover, since deciding whether a k-width hypertree decomposition exists and, if yes, computing it, are polynomially solvable problems, the described decomposition method defines a k-tractable class for the maximum weighted set packing problem [11].

Chapter 4

ComputeSetPacking$_k$ Implementation

This chapter is dedicated to the description of our implementation of ComputeSetPacking$_k$. We start the description of the algorithms that construct a specific decomposition, namely a complete hypertree decomposition of the dual hypergraph for the problem. Then we show how essential steps of ComputeSetPacking$_k$, such as formulation of partial problems in every node of the hypertree decomposition and propagation of the solutions of these partial problems along the hypertree, are carried out. For convenience, we describe these steps using a database notation, considering the winner determination problem as a constraint satisfaction problem.

Throughout the chapter we will illustrate these and other ideas on the following example of a combinatorial auction.

Example 4.1. *Consider a combinatorial auction* $\langle \mathcal{I}, \mathcal{B} \rangle$*, where*

$$\begin{aligned}
&\mathcal{I} = \{I_1, I_2, I_3, I_4\},\\
&\mathcal{B} = \{B_1, B_2, B_3, B_4, B_5\},\\
&B_1 = \langle \{I_1, I_2\}, 1 \rangle,\\
&B_2 = \langle \{I_2, I_3\}, 2 \rangle,\\
&B_3 = \langle \{I_1, I_3\}, 2 \rangle,\\
&B_4 = \langle \{I_2, I_4\}, 1 \rangle,\\
&B_5 = \langle \{I_4\}, 2 \rangle.
\end{aligned}$$

The hypergraph representation of the combinatorial auction is shown in Figure 4.1. Note that the hypergraph is cyclic.

4.1 General Architecture

In this section we give an informal overview of the processing steps for a given combinatorial auction in order to solve the corresponding winner

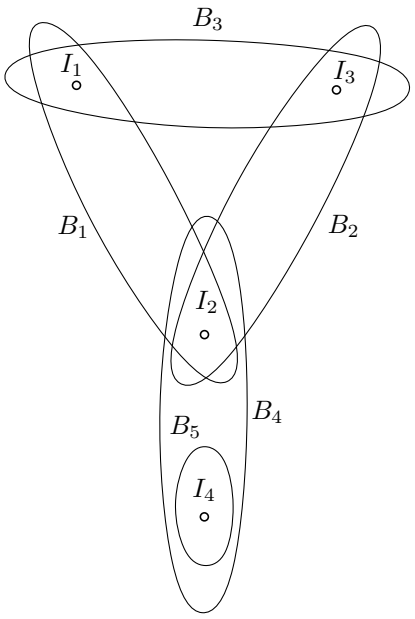


Figure 4.1: Hypergraph for the combinatorial action from Example 4.1

determination problem. Figure 4.2 illustrates the sequence of structural modifications of a weighted hypergraph corresponding to the combinatorial auction from Example 4.1.

Given a weighted hypergraph $\mathcal{H}$ representing a combinatorial auction. To decompose the underlying constraint satisfaction problem the following steps are carried out:

- Construction of the dual hypergraph $\widetilde{\mathcal{H}}$ for $\mathcal{H}$.
- Construction of the hypertree decomposition $\mathcal{T}$ of the dual hypergraph $\widetilde{\mathcal{H}}$.
- Completion of $\mathcal{T}(\widetilde{\mathcal{H}})$.
- Construction of the constraint network $\mathcal{N}$ based on $\widetilde{\mathcal{H}}$ and formulation the corresponding acyclic constraint satisfaction problem.

Note that the dual hypergraph $\widetilde{\mathcal{H}}$ (Figure 4.3) represents a new formulation for the original constraint satisfaction problem: constraints in $\widetilde{\mathcal{H}}$ say that we can choose only one vertex per hyperedge.

After these transformation steps, the weighted hypergraph $\mathcal{H}$, the constraint network $\mathcal{N}$ and the complete $\mathcal{T}(\widetilde{\mathcal{H}})$ will be given as input for ComputeSetPacking$_k$.

In the following sections we describe these steps in more details.

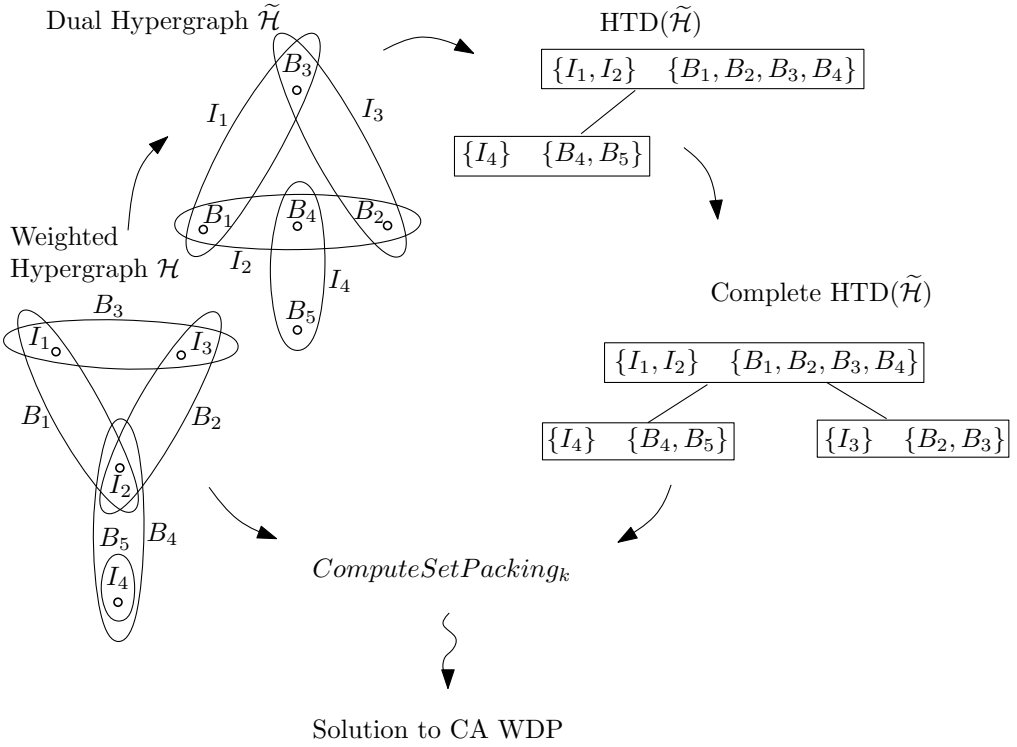


Figure 4.2: Example of a Processing Flow

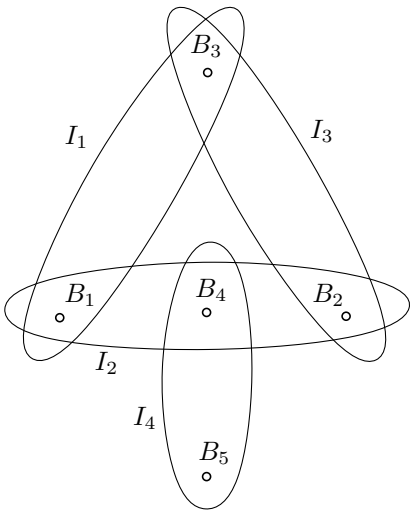


Figure 4.3: Dual hypergraph for the hypergraph from Figure 4.1

4.2 Constructing a Complete Hypertree Decomposition

A complete hypertree decomposition of the hypergraph associated to a constraint network is required as one of the input arguments for the algorithm ComputeSetPacking$_k$. As explained in Section 2.5, this decomposition represents an acyclic constraint network equivalent to the initial one. In the next subsections we show procedures that we use for constructing such a decomposition.

4.2.1 Variable Orderings

Variable ordering is a useful concept for managing many problems in graph theory. Some of these problems are to decide the consistency of a constraint network and decompose a cyclic constraint network into an equivalent acyclic network. One of the algorithms for coping with these tasks is the Bucket Elimination Adaptive Consistency algorithm [7]. This algorithm is presented in more details in the following section. Here we describe two heuristic methods for computing variable orderings. Both methods were used in our implementation.

Min-Induced-Width and Max-Cardinality methods compute the variable orderings for a graph [28]. However, we can also use these methods to get a variable ordering for a hypergraph. For that we just use its primal graph as an input for the algorithms [8].

Min-Induced-Width orders the variables of the graph from last to first in the following way: It first selects a vertex which is connected with the least number of vertices in the graph, and puts this variable in the last position in the ordering. Then the algorithm creates edges connecting to each other the neighbouring vertices of the selected vertex. Afterwards, it eliminates the selected vertex and all its adjacent edges from the original graph. The selection of next variables continues recursively with the remaining subgraph [7].

Max-Cardinality orders the vertices of a graph from first to last according to the following procedure: It firstly picks up a random vertex from the graph. After this it selects, one after another, those vertices that are connected to a maximal number of already ordered vertices (breaking ties randomly) [7].

4.2.2 Bucket Elimination for Hypertree Decompositions

The Bucket Elimination Adaptive Consistency method is a method used in Constraint Processing [7]. Given a variable ordering d, the method firstly associates a bucket to each variable of a problem. Then, it puts each constraint C into the bucket associated to the latest variable (according to d)

in the scope of C. Hence, each constraint of the problem is placed in some bucket, and constraints that have the same latest variable (according to d) in their scopes are placed in the same bucket. Then, by processing in a direction that is reversed to the direction of the ordering d, the method solves subproblems represented by every bucket, recording the result as a new constraint and putting it in the bucket of the latest variable (according to d) of its scope.

The Bucket Elimination Adaptive Consistency method can be extended to create a tree decomposition $\langle T, \chi \rangle$, where $T = \langle N, E \rangle$, for a hypergraph representing a constraint satisfaction problem [8, 21]. For every bucket, a node $t \in N$ is created. Every time a solution of the subproblem represented by a bucket is put into another bucket, the method connects by an edge two nodes from N that are associated to those two buckets, and adds this edge to E. At the end, for each $t \in N$, the set $\chi(t)$ is composed of exactly those vertices that appear in the corresponding bucket. A pseudo-code of this method is presented in Algorithm 4.2.1 [1].

Algorithm 4.2.1: BUCKETELIMINATION($\mathcal{H}, d$)

comment: $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$

comment: $d = \langle v_1, \ldots, v_n \rangle$ is an ordering of vertices in $\mathcal{V}$

main
$E \leftarrow \{\}$
for $i \leftarrow 1$ **to** n
do $\begin{cases} Bucket_i \leftarrow \{\} \\ t_i \leftarrow \text{ new node in } N \end{cases}$
for $i \leftarrow n$ **to** 1
do $\begin{cases} Bucket_i \leftarrow Bucket_i \cup e, \\ \quad \text{s.t. } v_i \in e,\ e \in \mathcal{E} \text{ and } e \text{ is not considered yet} \end{cases}$
for $i \leftarrow n$ **to** 1
do $\begin{cases} A \leftarrow Bucket_i \setminus \{v_i\} \\ Bucket_j \leftarrow Bucket_j \cup A, \\ \quad \text{s.t. } v_j \text{ is the latest vertex in } A \text{ according } d \\ E \leftarrow E \cup < t_i, t_j > \end{cases}$
for $i \leftarrow 1$ **to** n
do $\chi(t_i) \leftarrow Bucket_i$
return $(< \langle N, E \rangle, \chi >)$

This method will produce a tree decomposition of smallest width, when an optimal variable ordering is provided [7].

[1] The implementation of the Bucket Elimination algorithm for constructing hypertree decompositions can be found at www.dbai.tuwien.ac.at/proj/hypertree/index.html

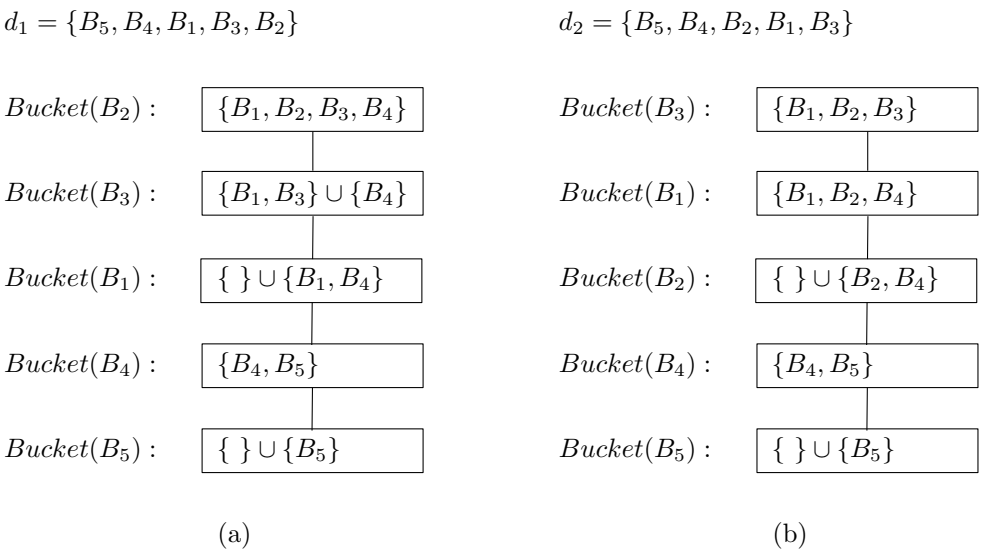


Figure 4.4: Execution of Bucket Elimination for two orderings on the hypergraph from Figure 4.3

Example 4.2. *Consider the hypergraph depicted in Figure 4.3. Figure 4.4 shows a schematic execution of Bucket Elimination method for two variable orderings:* $d_1 = \{B_5, B_4, B_1, B_3, B_2\}$ *and* $d_2 = \{B_5, B_4, B_2, B_1, B_3\}$. *The initial partitioning of variables into buckets is shown on the left side of* $\cup$*-symbol, while variables added to buckets are on the right side of* $\cup$*-symbol.*

A generalized hypertree decomposition can be obtained from a tree decomposition $\langle T, \chi \rangle$, where $T = \langle N, E \rangle$, by associating to each node $t \in N$ an additional set $\lambda(t) \subseteq \mathcal{E}$, such that the following condition is satisfied:

$$\text{for each } t \in N,\ \chi(t) \subseteq \Big(\bigcup_{e \in \lambda(t)} e \Big)$$

Intuitively, for each node t we need to "cover" each variable found in $\chi(t)$ by at least one edge presented in $\lambda(t)$. To solve this covering problem the greedy set covering heuristic can be applied [21]. For every node t this heuristic creates the set $\lambda(t)$ by iteratively choosing hyperedges covering most of the "uncovered" vertices from the $\chi(t)$-set.

Example 4.3. *Continuing our example, a generalized hypertree decomposition built based on the tree decomposition from Figure 4.4(a) (after some simplifications) is presented in Figure 4.5.*

4.2.3 Completing the Hypertree Decomposition

In order to represent the winner determination problem for a combinatorial auction as a constraint satisfaction problem we need to consider all hyper-

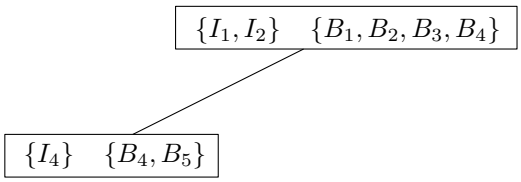


Figure 4.5: Generalized hypertree decomposition of the hypergraph from Figure 4.3

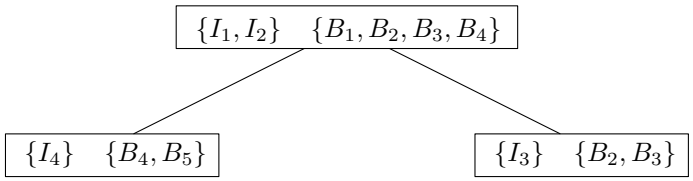


Figure 4.6: Complete hypertree decomposition of the hypergraph from Figure 4.3

edges of the hypergraph representation of the underlying problem. This is required for formulating the constraints imposed by the intersection of those hyperedges. However, as was discussed in Subsection 2.3.2, it is not required that all hyperedges of the hypergraph are strongly covered in a generalized hypertree decomposition. Hence, we need to complete this decomposition with respect to those hyperedges that have not been strongly covered. Given a hypergraph and its generalized hypertree decomposition, Algorithm 4.2.2 constructs a complete hypertree decomposition having the same width as the given generalized hypertree [13].

Intuitively, for every hyperedge $e \in \mathcal{E}$ of a hypergraph $\langle \mathcal{V}, \mathcal{E} \rangle$, that does not appear in the λ-set of some node of T of the generalized hypertree decomposition $\mathcal{T}$, s.t. $\mathcal{T} = \langle T, \chi, \lambda \rangle$ and $T = \langle N, E \rangle$, the algorithm does the following:

1. Finds a node $t \in N$, s.t. $e \subseteq \chi(t)$ (t exists in N by the first condition in Definition 2.3.10 of hypertree decomposition)

2. Creates a new node n, with $\chi(n) = e$ and $\lambda(n) = \{e\}$, and adds this node n to the N as a child of the node t

Example 4.4. *A complete hypertree decomposition for the generalized hypertree decomposition from Figure 4.5 and hypergraph from Figure 4.3 is shown in Figure 4.6.*

Algorithm 4.2.2: CompleteHTD($\mathcal{H}, \mathcal{T}$)

comment: $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$

comment: $\mathcal{T} = \langle T, \chi, \lambda \rangle$

comment: $T = \langle N, E \rangle$

main
for each $e \in \mathcal{E}$
do { **if** Missing($e, \mathcal{T}$) **then** $\mathcal{T} \leftarrow$ AddNode($e, \mathcal{T}$)
return ($\mathcal{T}$)

procedure Missing($e \in \mathcal{E}, \mathcal{T}$)
for each $n \in N$
do { **if** $e \in \lambda(n)$ **then return** (**false**)
return (**true**)

procedure AddNode($e \in \mathcal{E}, \mathcal{T}$)
$n \leftarrow$ new leaf s.t. $n \in N$ and n is a child of t,
where $t \in N$ and $e \subseteq \chi(t)$
$\chi(n) \leftarrow e$
$\lambda(n) \leftarrow \{e\}$
return ($\mathcal{T}$)

4.3 Formulating and Solving a Constraint Satisfaction Problem

Given a combinatorial auction instance, we first formulate a constraint network based on it. Then we use hypertree decomposition method to make this constraint network acyclic, as described in Subsection 2.5.2. Finally, we solve the corresponding acyclic constraint satisfaction problem following ideas from [29] which are briefly described in Subsection 2.5.1. Subsequently we illustrate how these steps are involved in ComputeSetPacking$_k$.

4.3.1 Constructing an Initial Constraint Network

Given a combinatorial auction and its hypergraph $\mathcal{H}$ representation, we formulate a constraint network (constraint satisfaction problem) for solving the corresponding winner determination problem. We first reformulate the

problem by constructing the dual hypergraph $\widetilde{\mathcal{H}}$ for the original hypergraph $\mathcal{H}$.

With each hyperedge of the dual hypergraph $\widetilde{\mathcal{H}}$ we associate a single constraint as it is shown in Algorithm 4.3.1. Hence, the number of constraints in the formulated constraint satisfaction problem is equal to the number of hyperedges in $\widetilde{\mathcal{H}}$.

Algorithm 4.3.1: CONSTRUCTCONSTRAINTNETWORK($\widetilde{\mathcal{H}}$)

comment: $\widetilde{\mathcal{H}} = \langle \widetilde{\mathcal{V}}, \widetilde{\mathcal{E}} \rangle$

main
$\mathcal{N} \leftarrow$ constraint network
for each $\widetilde{e} \in \widetilde{\mathcal{E}}$
do
$S \leftarrow \langle \widetilde{v} \mid \widetilde{v} \in \widetilde{e} \rangle$
$n \leftarrow |S|$
$R[S] \leftarrow \{t_j \mid t_j \in \{0,1\}^n$ and $\forall i \in \{1,n\}\ \ t(i) = 1$ iff $i = j\}$
$R[S] \leftarrow R[S] \cup \{0\}^n$
$C^{\widetilde{e}} \leftarrow$ constraint s.t. $C^{\widetilde{e}} = \langle S, R \rangle$
$\mathcal{N} \leftarrow \mathcal{N} \cup C^{\widetilde{e}}$
return $(\mathcal{N})$

In database terms, a constraint for a hyperedge $\widetilde{e}$ of a hypergraph $\widetilde{\mathcal{H}}$, $\widetilde{\mathcal{H}} = \langle \widetilde{\mathcal{V}}, \widetilde{\mathcal{E}} \rangle$, is a relation instance, with the relation name $\widetilde{e}$ and a set of attributes, such that there is an attribute corresponding to every vertex $\widetilde{v} \in \widetilde{e}$. Tuples are formulated based on constraints imposed by the combinatorial auction: each tuple of the relation instance represents one of the legal combinations of bids for the considered item of the combinatorial auction, respecting the condition that an item may appear at most in one bid at a time, i.e. in each hyperedge in the dual hypergraph we may choose at most one vertex.

Indeed, every tuple of a constraint defined as above corresponds to a partial packing for the original hypergraph $\mathcal{H}$, $\mathcal{H} = \langle \mathcal{V}, \mathcal{E} \rangle$, i.e. a packing for a hypergraph $\mathcal{H}'$, s.t. $\mathcal{H}' = \langle \mathcal{V}', \mathcal{E}' \rangle$, $\mathcal{V}' \subseteq \mathcal{V}$, $\mathcal{E}' \subseteq \mathcal{E}$, where in the set $\mathcal{E}'$ there are exactly the hyperedges from the scope of the constraint.

Example 4.5. *Constraints formulated for the combinatorial auction from Example 4.1 are presented on the Figure 4.7. Consider, for instance, the constraint I_2. The set of partial packings that this constraint represents is $\{\{\}, \{B_1\}, \{B_2\}, \{B_3\}\}$. The second tuple of the constraint I_2 corresponds to the partial packing $\{B_1\}$.*

I_1	B_1	B_3
	0	0
	1	0
	0	1

I_2	B_1	B_2	B_4
	0	0	0
	1	0	0
	0	1	0
	0	0	1

I_3	B_2	B_3
	0	0
	1	0
	0	1

I_4	B_4	B_5
	0	0
	1	0
	0	1

Figure 4.7: Constraints for the combinatorial auction from Example 4.1

4.3.2 Constructing an Acyclic Constraint Network

Given a cyclic constraint network and its complete hypertree decomposition $\mathcal{T}$ with bounded width, we can formulate an equivalent acyclic constraint network and efficiently solve the corresponding constraint satisfaction problem. To do so, we need to associate a constraint to each node n of the hypertree decomposition. If there is only one element in the set $\lambda(n)$, we just take the corresponding constraint from the initial constraint network. However, if there is more than one element in $\lambda(n)$, we need to make joins among the corresponding constraints, project the result over the variables in $\chi(n)$, and associate the obtained constraint with the node n. Moreover, we need to associate a weight for every tuple of the obtained constraint computed as a sum of weights of those hyperedges that appear in the corresponding packing.

Example 4.6. *Recall the combinatorial auction from Example 4.1 and the complete hypertree decomposition of the dual hypergraph for it shown in Figure 4.6. We need to make a join between constraints I_1 and I_2, project the result of the join over the variables B_1, B_3, B_2, B_4, and place the resulting constraint in the the root. Additionally, we need to compute the weight for every tuple of the constraint as a sum of weights of those hyperedges that appear in the corresponding partial packing. The constraint obtained as a result of the join and weights computed for every tuple of this constraint are shown in Figure 4.8. Consider, for instance, the sixth tuple $t_6 = \langle 0, 1, 0, 1 \rangle$. Then $w(t_6) = w(B_3) + w(B_4) = 2 + 1 = 3$. Thus, the acyclic constraint network for the problem from Example 4.1 consists from the constraints I_3, I_4 and $\pi_{B_1,B_3,B_2,B_4}(I_1 \bowtie I_2)$.*

$\pi_{B_1,B_3,B_2,B_4}(I_1 \bowtie I_2)$	B_1	B_3	B_2	B_4	w
	0	0	0	0	0
	0	0	1	0	2
	0	0	0	1	1
	0	1	0	0	2
	0	1	1	0	4
	0	1	0	1	3
	1	0	0	0	1

Figure 4.8: Constraint in the root node of the hypertree from Figure 4.6 after performing join

4.3.3 Filtering Non-Conforming Tuples

After the acyclic constraint network is formulated, we can solve the corresponding constraint satisfaction problem by considering every node n of the hypertree decomposition and filtering those tuples in the constraint associated to n that do not agree on common attributes with any of the tuples of some child-node of n. We do this in the bottom-up fashion, starting from the leaves of the hypertree, by computing semi-joins between the corresponding relations. Moreover, while performing these semi-joins we are looking for a conforming tuple that is the best in terms of weight-revenue for every tuple in the constraint relation of the parent-node.

In the next subsubsections we explain these procedures in more details. In the examples we use the following notations: "C_{root}", "C_{left}", "C_{right}" stand for the constraints associated to the root-node, the left-child node and the right-child node respectively; "w" is the weight of a tuple in the corresponding constraint, while "w_l" and "w_r" are updated weight-values in the root-node after semi-joins with the left-child node and with the right-child node respectively.

Performing Semi-Joins

In order to accomplish semi-joins more efficiently, instead of looking through all tuples quadratically, we firstly sort relations of both constraints participating in the semi-join over their common attributes. Then, we traverse the sorted relations starting from their first tuples until we reach their last tuples removing from the relation of the parent constraint tuples that do not have a tuple in the relation of the child constraint identical on the common attributes. Moreover, sorting allows a faster search of the "best" conforming tuples already while performing semi-joins.

Example 4.7. *Continuing Example 4.1, recall the complete hypertree de-*

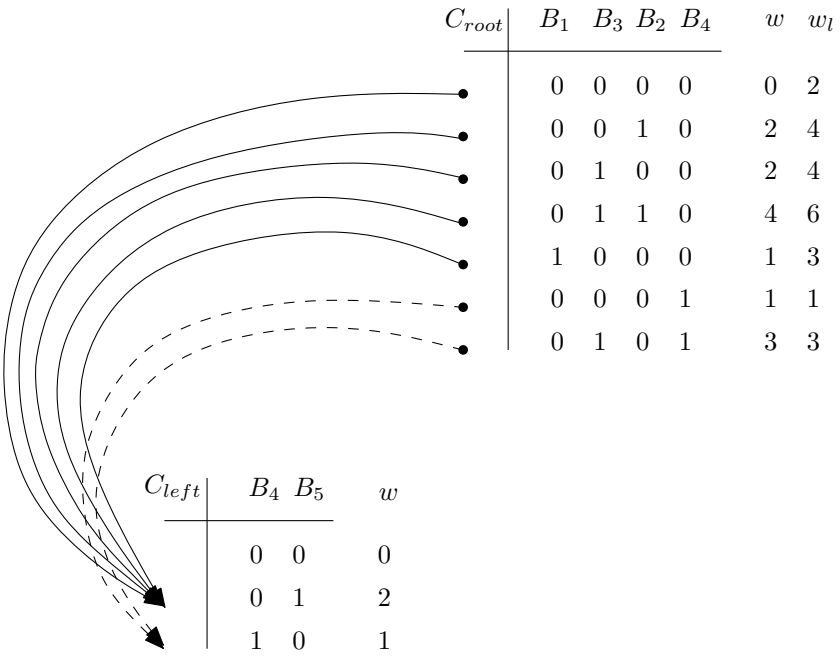


Figure 4.9: Semi-join of the relation in the root with the relation in the left child for the hypertree in Figure 4.6

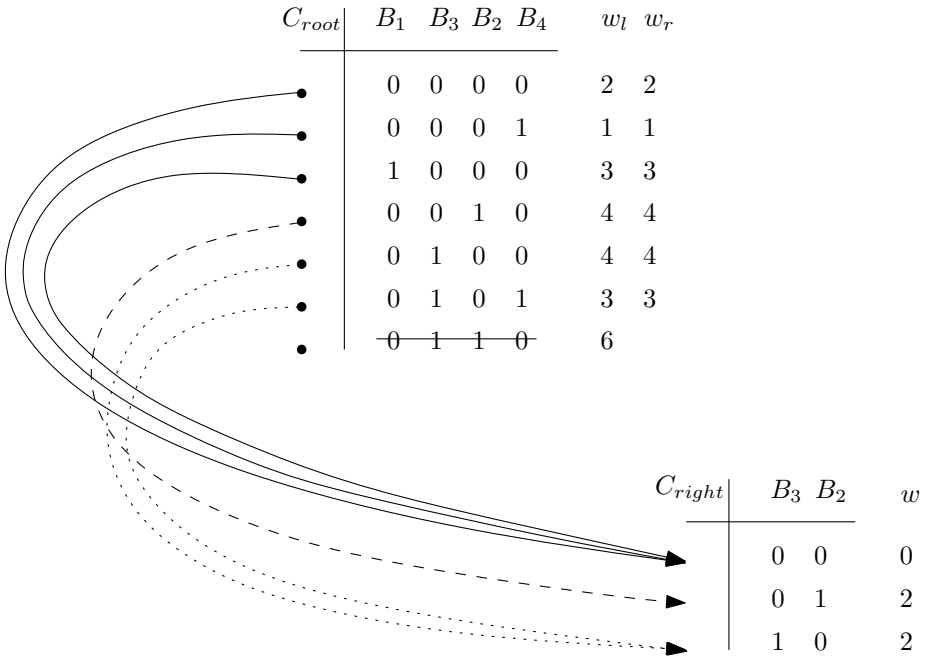


Figure 4.10: Semi-join of the relation in the root with the relation in the left child for the hypertree in Figure 4.6

composition for it from Figure 4.6. As we see, the hypertree decomposition consists of three nodes, one of which is the root, and the other two are child-nodes of the root: the left child and the right child. Hence, we need to make two consequent semi-joins of the constraint associated to the root node with the constraints associated to child-nodes of the hypertree. The semi-join with the left-child is performed over the attribute B_4, and the semi-join with the right-child is performed over the attributes B_3 and B_2. These semi-joins are shown on Figure 4.9 and on Figure 4.10 respectively. After every semi-join the constraint in the root-node is updated with the result.

Note that after performing the semi-join with the constraint relation in the right-child (Figure 4.10), one of the tuples was eliminated from the relation in the root-node. The reason is that there is no tuple in C_{root} that would agree with the removed tuple on the common attributes (i.e. on B_3 and B_2).

Determining Best Partial Solutions

While performing semi-joins, for every tuple t_p in the parent-node relation we look for "best" tuples that equal to t_p on their common attributes in the relations of every child-node. Those tuples are the ones that, conforming with the partial packing represented by t_p, bring the higher profit in terms of weight.

Example 4.8. *Consider once again Figures 4.9 and 4.10. In these figures an arrow coming from a tuple t_p of the parent-node and pointing to a tuple t_c of the child-node means that t_p and t_c agree on the common attributes, and the tuple t_c is the tuple from the constraint of the considered child node that has the highest weight among all other tuples that agree with t_p.*

Note that in Figure 4.9 the weights of the first five tuples in the constraint of the parent node were updated. It is explained by the fact, that for each tuple t_p among the first five ones the "best" tuple t_c of the left-child relation was the tuple, corresponding to a partial packing of the underlying hypergraph, where the hyperedge B_5 appears. Hence, considering that B_5 is not a common attribute for the semi-joined constraints, the tuple t_c will "bring" the weight-revenue equal to the weight of the hyperedge B_5. Take, for instance, the forth tuple $t_4 = \langle 0, 1, 1, 0\rangle$ of the parent-constraint. Then $w_l(t_4) = w(t_4) + w(t_c) - w(t_4 \cap t_c) = 4 + 2 - 0 = 6$.

For the last two tuples in the parent-constraint, the best tuples found in the left-child constraint will not "bring" any weight-revenue. Hence, weights of those tuples were not updated. Consider, for instance, the last tuple $t_7 = \langle 0, 1, 0, 1\rangle$. Then $w_l(t_7) = w(t_7) + w(t_c) - w(t_7 \cap t_c) = 3 + 1 - 1 = 3$.

Accumulating the Answer

If the resulting constraint in the root-node is not empty after the last semi-join in the root-node is performed, then the solution to the winner determination problem exists. To get the solution, we first choose a tuple having the maximal weight in the relation of the root-node and include the partial packing that this tuple represents into solution. Then we traverse the hypertree down to the leaves, accumulating the "best" packings conforming to the packings that are already included to solution.

Example 4.9. *Finishing Example 4.1, the constraint associated to the root node of the complete hypertree decomposition (Figure 4.6), after the bottom-up phase was finished, is shown in Figure 4.10. There are two tuples* $t_4 = \langle 0, 0, 1, 0 \rangle$ *and* $t_5 = \langle 0, 1, 0, 0 \rangle$ *having the maximal weight* $w_r(t_4) = w_r(t_5) = 4$ *among the other tuples in the relation. This is the highest possible revenue from the combinatorial auction. To find the allocation of non-intersecting hyperedges that lead to this revenue, we firstly choose any of* t_4 *or* t_5*, and then consider in Top-Down fashion the other tuples that are being referenced by the choosen one. For instance, consider* t_4 *that represents the partial packing* $\{B_2\}$*. We first add* $\{B_2\}$ *to the solution:* $res = \{B_2\}$*. Then we consider the "best" tuples for* t_4 *in the left- and right- children nodes of the root. The "best" tuple from the left-child node (Figure 4.9) represents the partial packing* $\{B_5\}$*, which we add to the solution* $res = res \cup \{B_5\} = \{B_2\} \cup \{B_5\} = \{B_2, B_5\}$*. Since the "best" tuple from the right-child node (Figure 4.10) represents the empty partial packing* $\{\ \}$*, a solution to the problem is* $res = \{B_2, B_5\}$*.*

In a similar way, if we choose the tuple t_5 *in the root node of the hypertree as a best partial packing, the solution for the problem will be* $\{B_3, B_5\}$*.*

Chapter 5

Experimental Evaluation

This chapter describes experimental work investigating the solution for the winner determination problem for combinatorial auctions. It firstly introduces various techniques to generate test data based on statistical methods, then it gives an overview of distributions that were actually implemented by other researches on combinatorial auctions. Moreover, it describes a test suite for combinatorial auction algorithms that models realistic data. Finally, it presents experimental results obtained with the execution of the $\mathsf{ComputeSetPacking}_k$ algorithm on instances generated with these distributions, as well as the analysis of the results.

5.1 Experimental Setup

There may be different approaches to conduct experimental work on combinatorial auctions. One of these approaches is to use *human subjects*. Such tests may be useful in understanding the nature of human behavior under different auctions mechanisms. However, because of many reasons mentioned in [20], they are not suitable for evaluating the computational characteristics of those mechanisms. Another way to experiment on combinatorial auctions is to examine *particular problems*, such as coordination of railroad tracks or airport time slot allocation, to which combinatorial auctions are appear to suite well. The advantage of this approach is that it provides specific descriptions of problem domains to which combinatorial auctions can be applied. However, the approach does not provide a method to generate test data and it does not give the means to evaluate how the problem's difficulty depends on the number of items and bids [20].

To overcome the flaws of the approaches mentioned above, a number of studies proposed several bid generation techniques, parameterized by number of bids and items [5, 10, 24]. Besides, a suite for distributions based on real-world situations (Combinatorial Auctions Test Suite) was presented in [20]. In the next subsections we describe such parameterized distributions

in more details.

5.1.1 Artificial Distributions: General Characterization

Much of the research on the algorithms for combinatorial auctions was conducted in the absence of test suites. In order to test proposed algorithms, various distributions for generating test data were developed. As noticed in [20], each of these distributions can be seen as the answer to three questions: how many items to request in a bundle, which items to request, and what is the price offered for a bundle. In the following tables we summarize, following [20], some possible techniques that address these questions. These tables can be used as a framework to classify distributions.

Number of Items in a Bundle:

Uniform	Uniformly distributed on $[1, num\_items]$
Normal	Normally distributed with $\mu = \mu\_items$ and $\sigma = \sigma\_items$
Constant	Fixed at $constant\_items$
Decay	Starting with 1, repeatedly increment the size of the bundle until $rand(0,1)$ exceeds α, where $rand(0,1)$ is a real number drawn uniformly from $[0,1]$.
Binomial	Request n items with probability $p^n(1-p)^{num\_items-n}\binom{num\_items}{n}$
Exponential	Request n items with probability $Cexp^{-n\ q}$

Which Items to Request in a Bundle:

Random	Draw n random items from the set of all items, without replacement

Price Offer for a Bundle:

Fixed Random	Uniform on $[low\_fixed, hi\_fixed]$
Linear Random	Uniform on $[low\_linearly \cdot n, hi\_lineary \cdot n]$
Normal	Draw from a normal distribution with $\eta = \eta\_price$ and $\sigma = \sigma\_price$

5.1.2 Artificial Distributions: Published Work

With the classification framework presented in Subsection 5.1.1, we proceed here to study and classify distributions that were actually implemented by

other researches on combinatorial auctions. Our study is summarized in Table 5.1.

Each distribution was given a name by their proponents depending either on the technique employed to generate the number of items in a bid or on the technique for bid's price generation. In the last column in the table we show the names [1] of the corresponding distributions given by Leyton-Brown et al. in [20]. Note that we omit from the table the information about which items to request in a bundle, since all of the presented below distributions use random technique for answering this question.

5.1.3 Combinatorial Auctions Test Suite (CATS)

While the bid-generation techniques discussed above may be sufficient for evaluating or comparing algorithms, the suits they generate do not represent the scenarios happening in the real-world auctions. Therefore, it is hard to predict to which real-world problems each algorithm may be applied.

A universal test suite for combinatorial auction algorithms is introduced in [20]. Combinatorial Auctions Test Suite (CATS) is "a suite of distribution families for generating realistic, economically motivated combinatorial bids in five broad real-world domains." For most of these distributions the following procedure for generating bids are used:

1. Construction of a graph representing the adjacency relationships between items
2. Derivation of complementary properties between items and substitutability properties of bids (using the graph)

All five CATS distributions may generate slightly more bids then they were asked. For the detailed overview of the distributions see [20]. Below we describe them briefly.

Path in Space This class of problems can be described as the problem of purchasing a connection between two points. Although this distribution can be applied to model many of the real-life domains, the authors use the railway domain as an intuitive example. In the graph corresponding to the problem, nodes represent locations and edges represent connections between locations. Therefore, the items of the auctions are edges of the graph, and bids are the sets of the edges that make a path between two nodes.

The graph is generated randomly with various parameters that can be adjusted. The technique generates non-planar graphs. Bids are

[1] CATS' `L3` distribution corresponds to Sandholm's [24, 27] uniform distribution with the $constant\_items = 3$.

Table 5.1: Classification of Artificial Distributions

author	# of items	price	CATS
Random			
Sandholm [24, 27]	uniform	fixed random with $low\_fixed = 0$, $hi\_fixed = 1$	L1
Andersson et al. [1]	uniform	fixed random with $low\_fixed = 1$, $hi\_fixed = 1000$	L1a
Weighted Random			
Sandholm [24, 27]	uniform	linear random with $low\_linearly = 0$, $hi\_linearly = 1$	L2
Andersson et al. [1]	uniform	linear random with $low\_linearly = 500$, $hi\_linearly = 1500$	L2a
Uniform			
Sandholm [24, 27]	constant	fixed random with $low\_fixed = 0$, $hi\_fixed = 1$	L3
Decay			
Sandholm [24, 27]	decay with $\alpha = 0.55$	linear random with $low\_linearly = 0$, $hi\_linearly = 1$	L4
Andersson et al. [1]	decay with $\alpha = 0.55$	linear random with $low\_linearly = 1$, $hi\_linearly = 1000$	L4a
Binomial			
Fujishima et al. [10]	binomial with $p = 0.2$	linear random with $low\_linearly = 0.5$, $high\_linearly = 1.5$	L7
Andersson et al. [1]	binomial with $p = 0.2$	linear random with $low\_linearly = 0.5$, $high\_linearly = 1.5$	L7a
Exponential			
Fujishima et al. [10]	exponential with $q = 5$	linear random with $low\_linearly = 0.5$, $high\_linearly = 1.5$	L6
Andersson et al. [1]	exponential with $q = 5$	linear random with $low\_linearly = 500$, $high\_linearly = 1500$	L6a

generated in the following way: randomly choose two nodes, start with the value for a path between those nodes equal to their Euclidean distance, make XOR bids on all alternative but more profitable paths between the nodes. The paths' values are random in parameterized proportion to the Euclidean distance between the nodes.

Proximity in Space The most intuitive examples of the real-world problem for this distribution is the sale of adjacent pieces of real estate. Nodes of the graph for this problem represent the items and edges represent the adjacency relationship, in such a way that there may be a variable number of neighbours per node.

The size of each bundle is determined by the decay distribution. A first item is added to the bid randomly. The other items for the bid are added as follows. For every new item to be added with uniformly random distribution, there is a small probability that it will not be adjacent with the existing items in the bundle. Otherwise, an item from the set of nodes bordering a node of the bundle is added. The price offered for the bundle depends on the sum of common and private valuations for the items in the bundle, and also takes into consideration a superadditive function on the number of items.

Arbitrary Relationships This technique generates problems in which there are arbitrary complementarity relationships between items. The graph corresponding to this problem is fully-connected. Each edge from n_1 to n_2 of the graph is labelled with the value $d(n_1, n_2) = rand(0, 1)$.

The technique for modelling bids is the generalization of the technique for Proximity in Space distribution: For each bid B, select the first item randomly. Proceed to add items, each with the probability proportional to $\sum_{n_2 \in B} d(n_1, n_2) \cdot p_i(n_1)$, where $p_i(n_1)$ represents bidder i's private valuation of the item (n_1).

Temporal Matching This approach models problems with real-world domains in which complementarity arises from a temporal relationship between items. Perhaps the most striking example of a problem of this king is the airport take-off and landing problem that is used in [20] for the description of the technique. In this example items are the privileges to use the runway at a particular airport at a particular time. Substitutable bids are different departure/arrival units. The graph used in the example is the real map of the four busiest US airports. The bidding mechanism presumes that airlines have a certain tolerance (expressed with statistical parameters) for the departure and arrival time of a plane, considering its most preferred departure and arrival times. The value of a bundle is derived from a particular agent's utility function.

Temporal Scheduling Temporal scheduling is a CA formulation for the distributed job-scheduling problem: a factory conducts an auction for time-slices on some resource; each bidder has a job requiring some amount of machine time, and one or more deadlines by which the job must be accomplished.

In the CATS formulation of the problem a specific time-slice is represented by an item. A set of substitutable bids satisfying the deadline constraints is generated. Note that two bids are substitutable, if they represent different possible schedules for the same job. The number of deadlines for each job is determined by the decay distribution. If $d_1 < \cdots < d_n$, where d_i is some deadline, and the value of a job accomplished by d_1 is v_1 (which is superadditive in the job length), then the value of a job accomplished by d_i is $v_i = v_1 \cdot \frac{d_1}{d_i}$. Hence, the later a deadline is, the less the bidder is willing to pay for it.

5.2 Experiments

To examine the performance of the $\mathsf{ComputeSetPacking_k}$ algorithm in practice, we measured its running time and the width of the hypertree decompositions varying the number of goods and bids on the input data generated with the CATS software. Our empirical results are based on computational experiments on instances involving up to 1500 bids and 1500 items.

For our experiments we considered L2, L3, L4, L6 and L7 artificial distributions, as well as matching, regions and scheduling distributions from the CATS suite [20].

All experiments were executed on twelve computers with Intel Celeron D 3.06 GHz and 1.24 GB of RAM running Windows XP. For every experiment, the algorithm was running until either an exact solution was found or a memory overflow took place.

5.2.1 Goals

The major goal of our experiments was to estimate and compare the hardness of different combinatorial auction benchmark distributions with respect to solving the corresponding combinatorial auctions with the $\mathsf{ComputeSetPacking_k}$ algorithm.

As already discussed in Subsection 2.5.2 and in Section 3.2, hypertree width is a significant tractability parameter for the winner determination problem in combinatorial auctions. Since we use a heuristic method (Bucket Elimination) to build hypertrees for combinatorial auctions, the widths of our hypertree decompositions (Definition 2.3.13) are only an estimation of the hypertree width (Definition 2.3.14) of the corresponding problems. Throughout the experiments when we talk about the width of the hypertree

decomposition we refer to a complete hypertree for the dual hypergraph built by the Bucket Elimination method for the underlying problem. For each experimental case we built two hypertree decompositions by the Bucket Elimination method - one with Min-Induced-Width ordering heuristics and the other with Max-Cardinality ordering heuristics. Then we considered the smaller width of the resulting hypertree decompositions.

With our experiments we observed how the width of hypertree decompositions changes while varying the number of bids and the number of items for different distributions.

Another goal of the experiments was to estimate a threshold of the width of the hypertree decomposition, such that a problem having a hypertree decomposition with the width lower than the threshold is easy enough to be solvable by the algorithm with the computational resources that we used for our experimental work. Additionally, for every distribution we experimentally determined ranges of values for the numbers of items and bids for which the corresponding problems have hypertree decompositions with width below the threshold.

To achieve our experimental goals we divided our experiments in two parts. In the first part we estimated the dependency of the running time of the algorithm on the width of the hypertree decompositions for different distributions. In the second part of our experiments we evaluated the dependency of the width of hypertree decompositions on the numbers of items and bids for different distributions. Next two subsections present these experiments.

5.2.2 Running Time Dependency on the Width of Hypertree Decompositions

In these experiments we estimated how running time depends on the width of hypertree decompositions for different distributions, and we found the range of values for the width of hypertree decompositions for which the program's execution finishes without producing memory overflow in our computers.

Experimental Procedure

To evaluate the running time of the algorithm for each of the distributions L2, L3, L4, L6 and regions we varied the number of bids in the range from 20 to 200 with step 20, the number of items in the range from 30 to 300 with step 30. For the distributions matching and scheduling we varied both the number of bids and the number of items in the range from 100 to 900 with step 200. Note that we considered the actual number of items and bids that was given as an input to the CATS generator: due to the nature of some of the distributions the generated instances could have a different number of items or bids than the number given as input to CATS - we assumed

that, if there were less items generated than we actually asked, then non-generated items were dummy in the sense that each of them belonged only to bids with prices equal to zero. For every experimental case, i.e. for every pair of numbers of bids and items, we executed the algorithm on three problem instances and took the average of the running times over those instances. The time required for parsing the test data and for building the hypertree decomposition was not measured, since we were only interested in the time required to solve the constraint satisfaction problem represented by the hypertree decomposition.

Throughout the experiments we estimated how running time varies with the width of the hypertree decomposition.

Results and Analysis

Figure 5.1 presents the dependence of the running time on the width of hypertree decompositions for different distributions. As we can see this dependence is exponential.

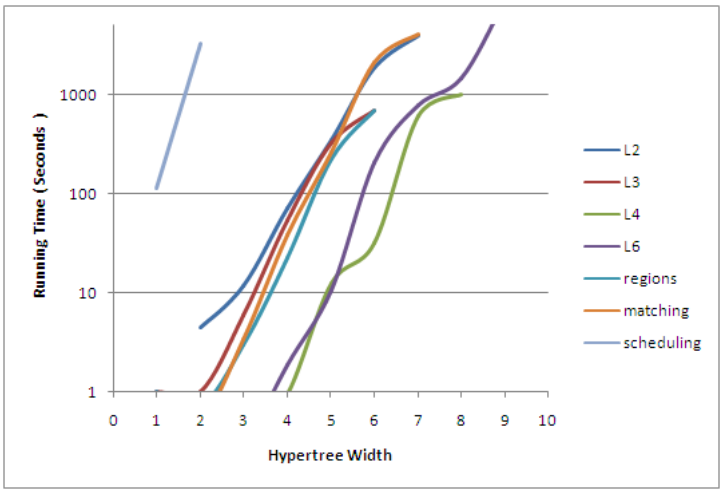


Figure 5.1: $\mathsf{ComputeSetPacking}_k$ running time dependence on the width of hypertree decompositions for distributions L2, L3, L4, L6, regions, matching and scheduling

It is interesting that, for a fixed width, the running time varies significantly among some of the distributions. For example, if we consider a hypertree decomposition with width equal to 2, the corresponding problem instance from the scheduling distribution was solved in 3263.4 seconds in average. An instance from the L2 distribution was solved in 4.45 seconds in average. An instance from the L4 distribution was solved in 0.12 seconds in average. This suggests that even though the hypertree width is an interesting and important measure of the difficulty of the problem, it is not the

only notion that must be considered.

By our experiments we observed that the size (the number of nodes) of the hypertree also influences the running time and the memory used by the algorithm. For example, given a fixed number of items and bids, the hypertree constructed for the instance of scheduling distribution has normally the smallest width among all distributions. However, it is much harder to solve this instance because of the large size of the corresponding hypertree. There is a tradeoff between the width of the hypertree decompositon and the size of the hypertree: the lower the width of the hypertree decompositions, the larger their sizes.

A possible direction for future research could be to estimate how the performance of the algorithm depends on the size of the hypertree decomposition, and to extend the algorithm considering the tradeoff between the width and the size of the hypertree decomposition. One of the approaches could be to use heuristics that would help to find the best combination of values for the size and the width of the hypertree decomposition for a given combinatorial auction instance.

Figure 5.2 shows the values of the width of the hypertree decompositions for which the algorithm terminated without causing memory overflow. For some of the instances having hypertree decompositions with widths larger than the maximal shown in Figure 5.2, the algorithm led to the depletion of memory.

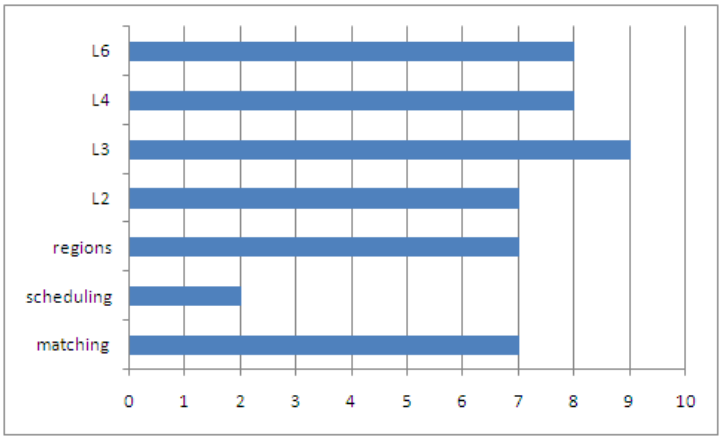


Figure 5.2: Values of the hypertree width for which $\mathsf{ComputeSetPacking}_k$ terminates for different distributions

As we can see from Figure 5.2, the distribution for which the algorithm manages to give an exact answer for instances with largest widths is L3. On the other extreme, the distribution that is the hardest to solve for $\mathsf{ComputeSetPacking}_k$ is scheduling.

5.2.3 Dependencies of the Width of Hypertree Decompositions

The experiments described in this section were designed to answer how the widths of hypertree decompositions vary while varying the number of bids and the number of items for different distributions.

Additionally, these experiments show the ranges of values for the numbers of items and bids for which the Bucket Elimination method was able to built hypertree decompositions with the width below the threshold that was determined with the previously described experiments.

Experimental Procedure

To make the observation of experiments more visual, we split distributions in two groups, so that the change of the widths of hypertree decompositions with the change of the numbers of bids and items is easily comparable with other distributions inside the group. The first group consisted of distributions L3, L4, L6, L7 and regions; for these distributions we varied both the number of items and the number of bids in the range from 20 to 400 with a step of 20. To the second group we attributed the scheduling distribution, which is the hardest distribution for $\mathsf{ComputeSetPacking}_k$; for scheduling distribution we varied the number of items and the number of bids in the range from 75 to 1500 with a step of 75. For each experimental case we executed the algorithm on one problem instance.

Throughout the experiments we evaluated:

1. How the width of the hypertree decomposition varies with the number of items.

2. How the width of the hypertree decomposition varies with the number of bids.

3. How the width of the hypertree decomposition varies with the distribution for fixed numbers of bids and items.

Results and Analysis

Figures 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8 show how the width of a hypertree decomposition varies with the number of items and the number of bids for distributions L3, L6, regions, L4, L7 and scheduling respectively. Examining them together with the data presented in Figure 5.2, we can estimate for which problem instance size, i.e. for which range of the values of the number of items and the number of items, the problem can be solved by $\mathsf{ComputeSetPacking}_k$ with the computational resources that we used. For example, if we consider the distribution L3 we can see that in Figure 5.3 the instances having the hypertree decomposition with the width at most equal

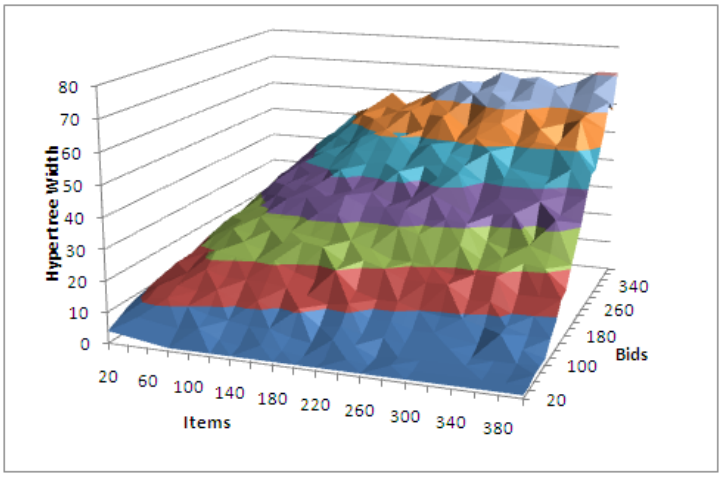


Figure 5.3: Dependence of the width of hypertree decompositions on the numbers of bids and items for distribution L3

to 9 are approximately in the range from 0 to 400 of the number of items and from 0 to 100 of the number of bids.

Below we give a comparison of the dependence of the width of hypertree decompositions on the numbers of items and bids for different distributions.

As we can see from Figures 5.3, 5.4 and 5.5, the width of the hypertree decompositions grows in a similar manner for the distributions L3, L6 and regions with the increase of the number of items and bids: for all three of them the value of the width of the hypertree decompositions does not depend much on the number of items, but depends substantially on the number of bids, increasing with the increase of the number of bids.

Nevertheless, some important differences can also be observed. First of all, L3 is the most difficult among these three distributions: for fixed numbers of items and bids, it results in a hypertree decomposition having width almost twice larger than for the distribution L6 and more than twice larger than for the distribution regions. In the range of the numbers of items and bids that we considered for these experiments (i.e. from 20 to 400 for both) the width of hypertree decompositions varies for these distributions in the following ranges: from 1 to 71 for L3; from 1 to 44 for L6; and from 2 to 27 for regions.

The fact that we considered only one instance for every experimental case allows us to observe one more interesting property of these distributions. As can be easily seen from Figures 5.3, 5.4 and 5.5, the surface for the distribution L3 is "smoother" than the surface for the distribution L6, while the surface for L6 is "smoother" than the surface for regions. Hence, we can conclude that, among them, the regions distribution behaves most randomly with respect to the width of the generated hypertree decomposition, and L3

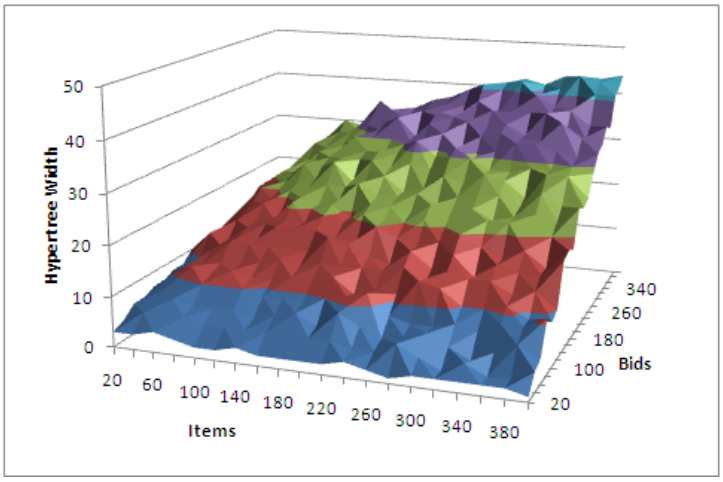


Figure 5.4: Dependence of the width of hypertree decompositions on the numbers of bids and items for distribution L6

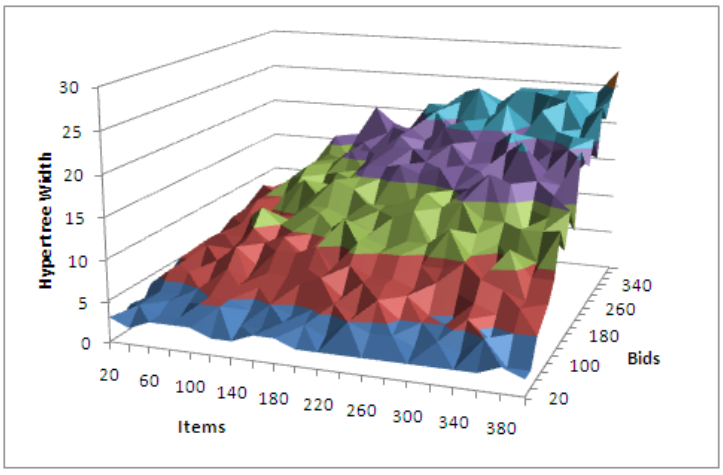


Figure 5.5: Dependence of the width of hypertree decompositions on the numbers of bids and items for regions distribution

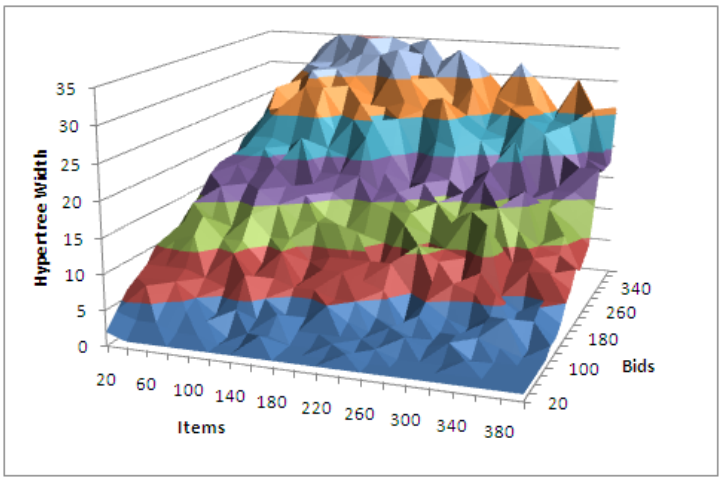


Figure 5.6: Dependence of the width of hypertree decompositions on the numbers of bids and items for distribution L4

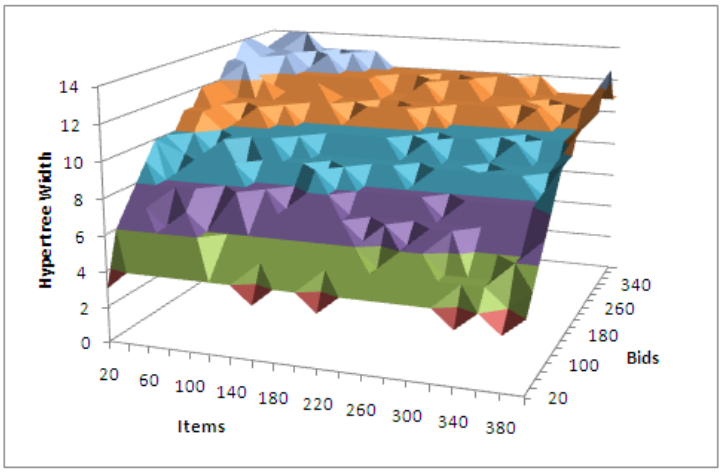


Figure 5.7: Dependence of the width of hypertree decompositions on the numbers of bids and items for distribution L7

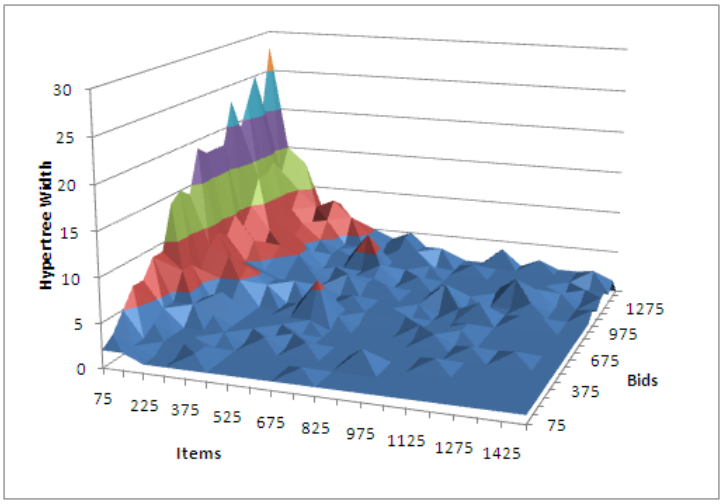


Figure 5.8: Dependence of the width of hypertree decompositions on the numbers of bids and items for scheduling distribution

behaves least randomly with this respect.

For the distribution L4 (Figure 5.6) the dependence of the width of hypertree decompositions on the number of items is the opposite in comparison to the dependence for other distributions in the group: it decreases with the increase of the number of items. As for the dependence on the number of bids, the distribution L4 behaves similarly to the other distributions: the width of hypertree decompositions increases with the number of bids.

Figure 5.7 shows the variation of the width of hypertree decompositions for the distribution L7. It increases with the number of bids with decreasing rate, in contrast to other distributions in the group. Moreover, this distribution is the easiest among them because the width of hypertree decompositions is the smallest for the fixed numbers of items and bids.

The dependence of the width of hypertree decompositions on the numbers of items and bids for the scheduling distribution behaves differently from the dependencies presented above. As we see from Figure 5.8, the value of the width of a hypertree decomposition depends substantially both on the number of bids and on the number of items: the width of hypertree decompositions increases with the number of bids, the highest widths occur for small numbers of items and they reduce drastically with the increase of the number of items. This is due to the nature of the scheduling distribution: with the increase of the number of items the number of cycles of overlapping bids decreases, causing the decrease of the width of a hypertree decomposition. In contrast, for the other distributions, the increase of the number of items does not influence as much the number of cycles of overlapping bids, hence the width of a hypertree decomposition does not change significantly

or remains the same.

5.2.4 Experimental Conclusions

To conclude the experimental work, we classify the distributions that we tested by their hardness to be solved by $\mathsf{ComputeSetPacking_k}$ with respect to the values of the width of hypertree decompositions built by Bucket Elimination and the influence of this width to the running time.

Considering the length of the running time for fixed widths of hypertree decompositions that were built in our experiments, we can separate the distributions in three groups of hardness according to Figure 5.1. In the first group we include the scheduling distribution that requires much longer time to be solved already for hypertree decompositions with very small width. L2, L3, matching and regions distributions form the second group. To the third and easiest group, we attribute the L4 and L6 distributions since they require less time to find the solution even for hypertree decompositions with large widths.

With regard to the speed of increase of the width of hypertree decompositions with the numbers of items and bids (Figures 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8), we can say that the L3 distribution is the most rapid, hence the hardest to be solved for a larger number of items and bids. Then follow L6, regions, L4, L7, concluding with scheduling.

However, as we already mentioned before, the width of the hypertree decomposition is not the only factor influencing the hardness of the problem for the algorithm, and future research could be aimed on identifying other factors, such as the tradeoff between the width and the size of a hypertree decomposition, and estimating the behavior of the algorithm with respect to these factors.

5.2.5 Comparison of Distribution's Difficulties between Algorithms

In this subsection we give a brief comparison of the distributions with respect to the difficulty to be solved by CABOB, CASS, CPLEX and $\mathsf{ComputeSetPacking_k}$ algorithms.

L2 is known to be an easy distribution for the CABOB, CASS and CPLEX algorithms. For these algorithms this distribution is shown to be much easier than L3 and L4 distributions, while L3 is known to be harder than L4 [19, 27]. On the contrary, for $\mathsf{ComputeSetPacking_k}$ L2 and L3 distributions have a similar difficulty, and the distribution L4 is easier than both of them.

Moreover, according to [19], L6 and L7 are among the hardest distributions for CASS. For $\mathsf{ComputeSetPacking_k}$ the distribution L6 is one of the easiest among the distributions that we tested for the hypertree decompo-

sitions with small width. L7 is easy for $\mathsf{ComputeSetPacking}_k$ because the speed of the width growth with the numbers of items and bids is relatively low.

All CATS distributions, apart from scheduling which is of medium difficulty, are shown to be the easiest for CABOB and CPLEX [27]. For CASS these distributions are hard [19]. As our experiments have shown, the scheduling distribution is the hardest distribution for $\mathsf{ComputeSetPacking}_k$, and the matching and the regions distributions are much easier than scheduling.

Chapter 6

Conclusion

In this thesis we formally presented the main notions of combinatorial auctions theory, giving an overview of previous research targeted on solving the winner determination problem for combinatorial auctions. Moreover, we described the concepts involved in solving the winner determination problem by means of the $\mathsf{ComputeSetPacking}_k$ algorithm which is a polynomial time algorithm that uses the technique of hypertree decompositions.

We implemented the $\mathsf{ComputeSetPacking}_k$ algorithm in a way that uses an existing implementation of the Bucket Elimination algorithm with Min-Induced-Width and Max-Cardinality heuristics to prepare the required hypertree decompositions. We experimentally tested our implementation on L2, L3, L4, L6, regions, matching and scheduling benchmark distributions involving up to 900 items and bids. Our experiments have shown that the algorithm could be successfully applied to the problem instances having the width of hypertree decompositions below a certain threshold that is specific for each distribution. Therefore, we analyzed how the width varies with the numbers of bids and items and we experimentally estimated the threshold for each distribution. As another result of our experiments, we observed that the size of a hypertree decomposition built for a problem also influences the performance of the algorithm.

Our experiments show that the efficiency of the algorithm could be significantly improved by making it or the underlying heuristics for hypertree decompositions more specific to particular distributions. Extending it with techniques aimed to reduce the memory consumption would allow a general increase in the solvability threshold for the width of the hypertree decompositions. These are possible directions for future research.

Bibliography

[1] Arne Andersson, Mattias Tenhunen, and Fredrik Ygge. Integer programming for combinatorial auction winner determination. In *ICMAS '00: Proceedings of the Fourth International Conference on MultiAgent Systems (ICMAS-2000)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[2] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[3] Vincent Conitzer, Jonathan Derryberry, and Tuomas Sandholm. Combinatorial auctions with structured item graphs. In *AAAI*, pages 212–218, 2004.

[4] Peter Cramton, Yoav Shoham, and Richard Steinberg. *Combinatorial Auctions*. The MIT Press, 2006.

[5] Sven de Vries and Rakesh V. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on Computing*, (3):284–309, 2003.

[6] Rina Dechter. Constraint Networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992.

[7] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[8] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Ben McMahan, Nysret Musliu, and Marko Samer. Heursitic methods for hypertree decompositions. Technical report, Technische Universität Wien, DBAI-TR-2005-5, 2005.

[9] Elaine M. Eschen and Jeremy P. Sinrad. An o(n2) algorithm for circular-arc graph recognition. In *SODA '93: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 128–137, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[10] Yuzo Fujishima, Kevin Leyton-Brown, and Yoav Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 548–553, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[11] Georg Gottlob and Gianluigi Greco. On the complexity of combinatorial auctions: structured item graphs and hypertree decomposition. In *ACM Conference on Electronic Commerce*, pages 152–161, 2007.

[12] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, pages 1–15, 2005.

[13] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural csp decomposition methods. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 394–399, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[14] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32, New York, NY, USA, 1999. ACM.

[15] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On tractable queries and constraints. In *DEXA '99: Proceedings of the 10th International Conference on Database and Expert Systems Applications*, pages 1–15, London, UK, 1999. Springer-Verlag.

[16] Charles L. Jackson. *Technology for spectrum markets*. PhD thesis, Department of Electrical Engineering, MIT, Cambridge, MA, 1976.

[17] Norbert Korte and Rolf H. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput.*, 18(1):68–81, 1989.

[18] Daniel Lehmann, Rudolf Mueller, and Tuomas Sandholm. *Combinatorial Auctions*, chapter The winner determination problem, pages 297 – 317. The MIT Press, 2006.

[19] Kevin Leyton-Brown. *Resource Allocation in Competitive Multiagent Systems*. PhD thesis, Stanford University, August 2003.

[20] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *ACM Conference on Electronic Commerce*, pages 66–76, 2000.

[21] Ben McMahan. Bucket elimination and hypertree decompositions. Implementation report, Institute of Information Systems (DBAI), TU Vienna, 2004.

[22] Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[23] Michael H. Rothkopf, Aleksandar Pekec, and Ronald M. Harstad. Computationally manageable combinational auctions. *Manage. Sci.*, 44(8):1131–1147, 1998.

[24] Tuomas Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1-2):1–54, 2002.

[25] Tuomas Sandholm. *Combinatorial Auctions*, chapter Optimal Winner Determination Algorithms, pages 337 – 368. The MIT Press, 2006.

[26] Tuomas Sandholm and Subhash Suri. Bob: Improved winner determination in combinatorial auctions and generalizations. *Artif. Intell.*, 145(1-2):33–58, 2003.

[27] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. CABOB: a fast optimal algorithm for winner determination in combinatorial auctions. *Management Science*, 51(3):374–391, 2005.

[28] Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, 1984.

[29] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.