The Knapsack Problem with Conflict Graphs

Ulrich Pferschy [*] Joachim Schauer [*]

Abstract

We extend the classical 0-1 knapsack problem by introducing disjunctive constraints for pairs of items which are not allowed to be packed together into the knapsack. These constraints are represented by edges of a conflict graph whose vertices correspond to the items of the knapsack problem. Similar conditions were treated in the literature for bin packing and scheduling problems. For the knapsack problem with conflict graph exact and heuristic algorithms were proposed in the past. While the problem is strongly NP-hard in general, we present pseudopolynomial algorithms for three special graph classes, namely trees, graphs with bounded treewidth and chordal graphs. From these algorithms we can easily derive fully polynomial time approximation schemes (FPTAS).

Keywords: knapsack problem, conflict graph.

1 Introduction

In this paper we consider an extension of the standard 0-1 knapsack problem. In addition to the usual weight constraint there exist incompatibilities for certain pairs of items. This means that from each such conflicting pair at most one item can be packed into the knapsack. It is natural to represent these symmetric conflict relations by means of an undirected *conflict graph* $G = (V, E)$, where every vertex corresponds uniquely to one item and an edge $(i, j) \in E$ indicates that items i and j can not be packed together.

For a formal definition of this *knapsack problem with conflict graph* (KCG), which is sometimes also referred to as *disjunctively constrained knapsack problem*, let n be the number of items, each of them with profit p_j and weight w_j, $j = 1, \ldots, n$, and c the capacity of the knapsack. The conflict graph $G = (V, E)$ with $|V| = n$ is not necessarily connected and may contain isolated vertices (i.e. items which can be combined with every other item). Then we state the

[*]University of Graz, Department of Statistics and Operations Research, Universitaetsstr. 15, A-8010 Graz, Austria, `{pferschy, joachim.schauer}@uni-graz.at`

following ILP formulation:

$$(KCG) \qquad \max \quad \sum_{j=1}^{n} p_j x_j \tag{1}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{2}$$

$$x_i + x_j \leq 1 \quad \forall \ (i,j) \in E \tag{3}$$

$$x_j \in \{0,1\} \quad j = 1, \ldots, n. \tag{4}$$

W.l.o.g. we can restrict KCG to connected conflict graphs. Indeed, we can introduce a dummy item $n+1$ with weight $w_{n+1} = c$ and profit $p_{n+1} = 0$ and insert edges from vertex $n+1$ to every other vertex thereby making every given conflict graph connected without changing the set of feasible solutions with positive profit.

As KCG is a generalization of the 0-1 knapsack problem it is easy to see that this problem is NP-hard (for a given instance of the knapsack problem introduce a star graph as a conflict graph centered at the above mentioned dummy vertex $n+1$).

From a graph theoretical perspective, KCG can also be seen as a generalization of the independent set (or stable set) problem which asks for a maximal set of vertices which are not adjacent to each other. For every given instance of the independent set problem we can superimpose an instance of KCG by introducing trivial items for every vertex with profit and weight equal to 1 and capacity $c = n$. It follows immediately, that KCG for general graphs is strongly NP-hard (cf. [13]) and does not permit pseudo-polynomial algorithms (under $P \neq NP$).

Motivated by this complexity status and following a line of research extensively pursued for the independent set problem, it is our main task in this paper to identify graph classes for which we can prove the existence of a pseudo-polynomial time and space algorithm for KCG. Furthermore, we will use these algorithms to attain fully polynomial time approximation schemes (FPTAS). In the following sections we will show that trees, graphs of bounded treewidth and chordal graphs (including interval graphs as a subclass [13]) used as conflict graphs in KCG do admit pseudo-polynomial algorithms as well as FPTASs.

Note that for unconnected graphs the special properties of some of these graph classes would be no longer valid after the extension of the graph by a dummy vertex as described above. However, all our algorithms are based on dynamic programming and compute optimal solutions for every capacity value $\leq c$. Hence we can in a first step process the components of the graph independently and then merge the solutions of all components. Obviously, the corresponding items from different components are all compatible with each other. To avoid technicalities we will restrict our considerations to KCG with connected conflict graphs.

For simplicity of presentation all our algorithms will be designed to compute only the optimal *solution value* of KCG. To derive also the corresponding *solution set of items* one has to store for every entry of a dynamic programming array also the corresponding partial solution set. Using a binary encoding this would increase all running time and space complexities for our algorithms by a factor of $\log n$ (see also Section 6).

1.1 Related Literature

The first paper dealing with KCG that we are aware of is Yamada et al. [23] from 2002. The authors apply the classical greedy algorithm to KCG by considering the items in decreasing order of their profit to weight ratio and packing an item into the knapsack if it is not in conflict with any previously packed item and if it does not violate the capacity constraint. As an extension a local search procedure based on a 2-opt exchange operation is introduced. Beside these lower bounds on the objective value, upper bounds are given first by the LP-relaxation and then by further relaxing the conflict conditions (3) in a Lagrangean way and iteratively computing better Lagrangean multipliers. Based on these bounds a branch-and-bound algorithm is constructed which uses the conflict relations and standard dominance criteria to reduce the search space.

Since the lower bound derived by the heuristic may be quite weak, it is suggested in [23] to run the branch-and-bound algorithm with an estimated lower bound computed as a convex combination of the originally computed upper and lower bounds. If the algorithm finds a feasible solution it will finally deliver an optimal solution much faster than with the weaker original lower bound. If the algorithm fails to find a feasible solution, the estimated value is a new and stronger upper bound and the procedure is iterated.

Further studies on KCG were recently pursued in two papers by Hifi and Michrafy. In [15] they present a metaheuristic, namely a sophisticated reactive local search algorithm combined with a tabu list. While the neighborhood structure is based on a pairwise exchange operation and the removal resp. insertion of single items, more involved degrading steps are used to escape local optima and for diversification. Computational experiments illustrate the successful behavior of this approach.

An interesting exact algorithm is presented by Hifi and Michrafy in [16] who also give pointers to applications of KCG. In fact, their paper contains three different exact algorithms which are subject of a computational study and all compare very favorably to CPLEX. The LP-relaxation is applied as a straightforward upper bound. Lower bounds are derived from an adaptation of the heuristic in [15]. After a reduction procedure to eliminate some items from further consideration a branch-and-bound approach is performed.

To tackle also instances with a large number of disjunctive constraints an equivalent model is introduced where for each item all conflicting items are combined

into a single constraint. Thereby the number of constraints can be reduced to n. Furthermore, dominating constraints and covering constraints are introduced to accelerate the branch-and-bound algorithm (see [16] for further details).

Conflict graphs were also considered for other combinatorial optimization problems related to the knapsack problem. In particular, several papers deal with the *bin packing problem with conflicts* (BPCG). In this case, the edges of the conflict graph define pairs of items which are not allowed to be packed into the same bin, i.e. the packing of every bin must be an independent (stable) set with respect to the given graph.

The first paper dealing with BPCG seems to be due to Jansen and Öhring [18], although they introduce the problem in a scheduling context. They state seven different approximation algorithms and analyze worst-case approximation ratios for special graph classes, e.g. a 2.7 ratio for perfect graphs. Some of their results were further improved by Epstein and Levin [8] who give a 2.5 approximation ratio for perfect graphs and also consider the on-line case of the problem. In [17] Jansen gives an asymptotic fully polynomial approximation scheme for BPCG if the underlying conflict graph is a $d-$inductive graph. Gendreau et al. [12] introduce six heuristics for BPCG and perform extensive computational experiments. For reasons of comparison, also two lower bounds are developed. An extension of BPCG to the two-dimensional case of packing squares was recently treated by Epstein et al. [9]. Again approximation algorithms for special graph classes were stated and analyzed.

To the best of our knowledge the only exact algorithm for BPCG is given in the recent paper by Muritiba et al. [21]. Their paper introduces non-trivial lower bounds based on the clique structure, on a matching and on a surrogate relaxation over all constraints (3). Upper bounds are computed by a population based heuristic framework with a tabu list. With these ingredients a branch-and-price algorithm is developed based on a set covering model for bin packing. The associated pricing problem is an instance of KCG which is solved by a parametric greedy-type heuristic in [21]. Detailed computational experiments illustrate the effectiveness of this approach.

Scheduling problem are closely related to bin packing problems although conflict structures can be imposed in various ways. Bodlaender and Jansen [4] introduce a scheduling problem where edges of a conflict graph represent jobs which have to be assigned to different machines. The goal is to minimize the makespan. For unit-length jobs they give complexity results for different graph classes. Bodlaender et al. [5] consider the same problem with arbitrary processing times and give approximation algorithms for various special graph classes.

A different setup is considered by Baker and Coffman [1]. Their *mutual exclusion scheduling* considers unit-length jobs where the edges of the conflict graph indicate pairs of jobs which are not allowed to be executed in the same time interval (but possibly on the same machine). They show that the problem can

be solved in polynomial time if the conflict graph is a forest and give a number of heuristics and approximation results.

1.2 Preliminaries

Given an undirected graph $G = (V, E)$ the degree $d(v)$ of a vertex $v \in V$ is defined as the number of vertices i $(i \neq v)$ in G that are adjacent to v. The distance of two vertices i and j in G is defined by the number of edges of the shortest path in G starting in i and ending in j $(dist(i,j) = dist(j,i))$. A subset $S \subset V$ is called vertex separator of G if there are two vertices a and b in one component C of G, such that the removal of the vertices in S from G separates a and b, i.e. a and b are in different components of $G - S$. S is then called ab-separator. By $V(G)$ we denote all the vertices of G $(V(G) = V)$.

A tree T is a connected graph with n vertices and $n-1$ edges. For our purpose it is necessary to consider some vertex r as root vertex of T $(T = T(r))$. A leaf vertex v of T is a vertex with degree 1 $(d(v) = 1)$. The vertex i is parent vertex of vertex j (child vertex) if the following properties are fulfilled: $dist(r,i) = dist(r,j) - 1$ and $(i,j) \in E$. The set children(i) denotes all vertices $j \in G$ with the property, that i is parent vertex of j. The height of $T(r)$ is defined as $\max_i\{d(r,i) : i \in T(r)\}$. For a tree T with root r, $T(i)$ defines the induced subtree of T with root i in which all vertices $j \in T(i)$ fulfill $d(j,r) \geq d(i,r)$ in T. $|T(i)|$ furthermore denotes the number of vertices in $T(i)$. Since we create the FPTAS with a dynamic programming formulation based on profits we define a trivial upper bound P on the total profit of the knapsack as $P = \sum_{i=1}^{n} p_i$.

2 Trees as Conflict Graphs

In this section we introduce a dynamic programming algorithm that solves KCG with a tree T as conflict graph in $O(nP^2)$ time using $O(\log(n)P + n)$ space. If we consider any vertex $i \in T$, by the property of trees as conflict graphs, when including i into the knapsack solution, it is not allowed to include the parent vertex p of i as well as any of the k child vertices $c_1 \dots c_k$ of i. Indeed these vertices are the only vertices in T that are in conflict with i. The main idea of our algorithm AlgTr presented in this section is to process T in depth-first order starting at some vertex r, which we consider as root vertex of T. Reaching a leaf vertex l with parent p, we distinguish two cases:

- Including l into the knapsack solution and as a consequence excluding p.
- Excluding l from the knapsack and as a consequence keeping the decision concerning p open.

We call this procedure *merging of l with p*. After all children of the vertex p are merged with p, p itself can be seen as new leaf vertex and the above idea can be applied recursively. We need some more notation to state Algorithm 1. $z_i(d)$ describes the solution with minimal weight found in the subtree $T(i)$ of $T = T(r)$ that leads to a profit of d with item i *necessarily included* into the knapsack solution. $y_i(d)$ describes the solution with minimal weight found in the subtree $T(i)$ of $T = T(r)$ that leads to a profit of d with item i *excluded* from the knapsack solution.

Algorithm 1 AlgTr

AlgTr($T(r)$): (a)

$$z_r(d) = \begin{cases} w(r) & d = p(r) \\ c+1 & d \neq p(r) \end{cases} \qquad \forall\, d \leq P$$

$$y_r(d) = \begin{cases} c+1 & 1 \leq d \leq P \\ 0 & d = 0 \end{cases}$$

for $j \in$ children(r): (b)

AlgTr($T(j)$) (a)

for $d \in [0, P]$:

$$y_r(d) = \min_k\{y_r(d-k) + \min\{z_j(k), y_j(k)\} \mid k \in [0, d]\} \qquad (c)$$

for $d \in [p(r), P]$:

$$z_r(d) = \min_k\{z_r(d-k) + y_j(k) \mid k \in [0, d - p(r)]\} \qquad (d)$$

Theorem 1 *The Algorithm AlgTr solves KCG with a tree T as conflict graph to optimality.*

Proof.
We will show the correctness of AlgTr by induction on the height h of T with root vertex r.

Assume that $h = 0$. Then T has just one vertex r, which is itself a leaf. Including r into the knapsack solution is the only possibility, the correctness follows.

Assume that the algorithm calculates the optimal solution for trees with height $h-1$. Then we will show that this is also true for trees with height h.

Case 1. The root r of T with height h has only one child vertex c_1. So the induced subtree $T(c_1)$ has height $h-1$ and is processed optimally by AlgTr (by assumption). $y_r(d)$ ((c) in AlgTr) is determined by an optimal combination with a solution computed in $T(c_1)$ which is optimal by assumption.

For calculating $z_r(d)$ AlgTr compares all possible combinations of profits leading to a total profit of d ((d) in AlgTr), where $y_j(k)$ is by assumption optimal and $z_r(p(r))$ was properly initialized to a weight of $w(r)$. So *Case 1* follows.

Case 2. The root of T with height h has j child vertices $c_1 \dots c_j$. By *Case 1* the merging of r with c_1 was done optimally. So in a second induction let us *assume* that for some $l \in [1, j-1]$ the merging procedure of $c_1 \dots c_l$ was done optimally too (*). Then also the merging of c_{l+1} with r is done optimally: $y_r(d)$ is calculated as the minimum over all weights k as $y_r(d) = \min_k\{y_r(d - k) + \min\{z_{c_{l+1}}(k), y_{c_{l+1}}(k)\} \mid k \in [0, d]\}$ leading to a total profit of d. The part described by $y_r(d-k)$ is optimal by (*), where r itself is not included into in the knapsack solution. The part described by $z_{c_{l+1}}(k)$ and $y_{c_{l+1}}(k)$ is optimal by the height of $T(c_{l+1})$, so the overall optimality of $y_r(d)$ follows by taking the minimum over all feasible solutions of the two components leading to a total profit of d. Clearly by the properties of trees the items described by the two parts are not in conflict with each other.

$z_r(d)$ is calculated as the minimum over all weights k in $z_r(d) = \min_k\{z_r(d - k) + y_j(k) \mid k \in [0, d - p(r)]\}$ leading again to a total profit of d. Here the same argument as for $y_r(d)$ works, with the difference, that now r is included, so all children of r have to be excluded. As in this case r adds a profit of $p(r)$ and a weight of $w(r)$ to the knapsack in calculating the minimum over all feasible solutions this has to be taken into account (in the range of k and d). So the merging of all children of r is done optimally and finally one can find the optimal solution as $\min\{z_r(d'), y_r(d')\}$ where d' has the property to be the largest profit that is reachable by a weight that is smaller than $c+1$. □

Theorem 2 *Algorithm AlgTr can be implemented to run in $O(nP^2)$ time and $O(nP)$ space.*

Proof.

Time Complexity. The recursive call of AlgTr described by (a) in Algorithm 1 is executed n times, once for each vertex $v \in T$. In the for loop described by (b) each vertex v is exactly once in the role of being child vertex over all recursive calls of AlgTr (with the exception of r), so (b) is executed exactly $n-1$ times during the algorithm. Combining these parts leads to a time complexity of $O(n)$. But in the part described by (c), P^2 combinations of feasible weights are considered. Since these P^2 array evaluations dominate all other calls involving weights an overall time complexity of $O(nP^2)$ follows.

Space Complexity. $z_j(d)$ and $y_j(d)$ have to be stored for all the n vertices j of T and for each possible profit $d \leq P$ leading to an overall space complexity of $O(nP)$. □

In the remainder of this section we show a general space reduction technique applicable to algorithms characterized by certain properties. This technique is useful for the algorithm AlgTr, as well as for the other algorithms we present in the following sections, since they can easily be adopted to fulfill the following three properties.

> **Property 1.** A tree T is processed in depth-first order combined with the following rule: at deciding which of the k child vertices $j_1 \dots j_k$ of parent i is used next in depth-first order, the algorithm takes the child vertex j_l, which fulfills
>
> $$|T(j_l)| = \max_i \{|T(j_i)| : \ i \in 1 \dots k\}.$$

This information can be gained by using depth-first search as a preprocessing step in order to calculate $|T(i)|$ for each vertex $i \in T$, which can be done in $O(n)$ time.

> **Property 2.** $O(k)$ storage space is used for the processing of each vertex i of T.

> **Property 3.** When a child vertex j of parent vertex i was processed, the information gained is merged to i, which is characterized by the fact that for vertex i again $O(k)$ space is used and all the storage space used for processing vertex j can be deallocated after the merging, leading to a total space of $2 * O(k)$ for the merging procedure.

Lemma 1 *An algorithm A fulfilling Property 1, Property 2 and Property 3 uses at most $(ld(n) + 1) * O(k)$ space.*

Proof.
For $n = 2$ the tree T contains two vertices, namely the root r with one child i. Then A needs $O(k)$ space for the processing of i and $O(k)$ space for r, by Property 3 a total space requirement of $2 * O(k)$ follows.

Assume that the statement is true for all trees with less than n vertices. We will show that the statement also holds for n vertices by considering the largest subtree of the root r of arbitrary size. Obviously, all other subtrees must contain less than $n/2$ vertices.

More formally, r has k children $i_1 \dots i_k$, $k \geq 1$, and w.l.o.g. $|T(i_j)| \leq \frac{n}{2}$ for all $j \in \{2 \dots k\}$ whereas $|T(i_1)| \leq n - 1$. Then by assumption the processing of $T(i_1)$ is done by using at most $(\text{ld}(n-1)+1) * O(k)$ space. After merging $T(i_1)$ to r this space can be deallocated, but $O(k)$ space is used at vertex r, which has to be kept until A has finished. Then the processing of $T(i_j)$, $j \geq 2$, is done using by assumption at most $(\text{ld}(\frac{n}{2}) + 1) * O(k) = \text{ld}(n) * O(k)$ space. After merging each of these subtrees to r this space can be deallocated, but the $O(k)$ space used at vertex r has to be kept until A has finished. This yields a total space requirement of $(\text{ld}(n) + 1) * O(k)$. □

AlgTr already fulfills Property 2 with $O(P)$ space for processing each vertex and Property 3, so by adapting (b) in Algorithm 1 according to Property 1, the

overall space complexity of AlgTr can be reduced to $O(\log(n) * P)$. For the sake of clarity this selection rule was not incorporated in Algorithm 1. Clearly, an additional $O(n)$ space is needed for processing a tree T with n vertices (even for storing these vertices), as well as for performing the depth-first search for Property 1 and for AlgTr itself (the longest path in T has to be stored for deciding if a vertex $v \in T$ has already been visited), leading to an overall space complexity for AlgTr of $O(\log(n) * P + n)$.

3 Graphs of Bounded Treewidth

In this section we treat graphs of bounded treewidth, for example series-parallel graphs, outerplanar graphs or Halin graphs ([3]). We show that given a tree-decomposition with constant treewidth k, there exists a dynamic programming algorithm solving KCG with a conflict graph of bounded treewidth k in $O(nP^2)$ time using $O(\log(n)P + n)$ space.

3.1 Definitions

In [7] a tree-decomposition is defined in the following way: Let $G = (V, E)$ be a graph, T a tree, and let $\mathcal{V} = (V_I)_{I \in V(T)}$ be a family of vertex sets $V_I \subseteq V(G)$ indexed by the vertices I of T. By capital letters we refer to vertices from T, whereas by lower case letters we refer to vertices from G. The pair $(T, \mathcal{V})$ is called a *tree-decomposition* if it satisfies the following three properties:

1. $V(G) = \bigcup_{I \in T} V_I$;
2. for every edge $e \in G$ there exists a $I \in T$ such that both ends of e lie in V_I;
3. $V_{I_1} \cap V_{I_3} \subseteq V_{I_2}$ whenever I_2 lies on the path from I_1 to I_3 in T.

The width of $(T, \mathcal{V})$ is defined as $\max\{|V_I| - 1 : \; I \in T\}$ and the *treewidth* of G is the smallest width of any tree-decomposition of G ([7]).

By [6] deciding whether a tree-decomposition of treewidth at most k exists, and if so, finding a tree-decomposition of width at most k can be done in linear time (if k is seen as a constant and not as part of the input).

For algorithmic purposes it is useful to consider a specially structure tree-decomposition, namely a *nice tree-decomposition*. In this case one vertex is considered to be the root vertex of T and each vertex $I \in T$ is of one of the following four types ([6]):

- Leaf: vertex I is a leaf of T and $|V_I| = 1$
- Join: vertex I has exactly two children, say J_1 and J_2, and $T_I = T_{J_1} = T_{J_2}$
- Introduce: vertex I has exactly one child, say J, and there is a vertex $v \in V$ with $V_I = V_J \cup \{v\}$
- Forget: vertex I has exactly one child, say J, and there is a vertex $v \in V$ with $V_J = V_I \cup \{v\}$

Furthermore by [6] a nice tree-decomposition with $O(n)$ tree vertices ($n = |V|$) with width at most k can be found in linear time, given a (not nice) tree-decomposition. For some vertex $I \in T$ we denote the tree-decomposition limited to the subtree $T(I)$ of T by $(T(I), \mathcal{V})$. Clearly $(T(I), \mathcal{V})$ is not any longer a tree-decomposition of G. Let furthermore G_I be the subgraph of G that is induced by $(T(I), \mathcal{V})$, more precisely by $\bigcup_{J \in T(I)} V_J$.

3.2 Algorithm AlgTDC

Let $G = (V, E)$ be a graph of bounded treewidth k, $(T, \mathcal{V})$ a nice tree-decomposition of G of width k, and R the root of T. Let U_J be the set of subsets S of vertices from V_J with the property in G that S is an independent set (IS) in G and $\sum_{i \in S} w(i) \leq c$ (U_J includes the empty set $\emptyset$). We define $f_d^S(J)$ as the minimum weight of the knapsack including the items $S \subseteq V_J$ with total profit equal to d, while considering only the limited tree-decomposition $(T(J), \mathcal{V})$. Then KCG is solved by algorithm AlgTDC which processes the tree-decomposition in depth-first order. A set of vertices in G that is not an independent set is abbreviated by DS. AlgTDC follows an idea presented in [6].

Theorem 3 *Algorithm AlgTDC solves KCG with a conflict graph G of bounded treewidth k to optimality.*

Proof.
Let $(T, \mathcal{V})$ be the nice tree-decomposition of G with root vertex $R \in T$ given. We will show that for each vertex $I \in T$ the algorithm computes an optimal solution for the subgraph G_I of G. This will be done by an induction like procedure: First the optimality is proved for leaf vertices of T. Then for each inner vertex $J \in T$, given that for the at most two children I_1 and I_2 of J the induced subgraphs G_{I_1} and G_{I_2} are calculated optimally, the optimality of G_J will be proved. Since $G = G_R$ the result follows.

Leaf vertices. Some leaf vertex I of T is the first vertex processed by AlgTDC. By definition of a nice tree-decomposition, V_I consists of exactly one vertex $v \in G$, so G_I equals a subgraph containing only v. By (a) in Algorithm 2 when including v into the knapsack solution (constrained to G_I) the only possible profit $d = p(v)$ has minimal weight $w(v)$.

Algorithm 2 AlgTDC

AlgTDC($(T(r), \mathcal{V})$): (e)

  if R is Leaf with vertex $v \in V_R$: (a)

$$f_d^v(R) = \begin{cases} w(v) & d = p(v) \\ c+1 & d \neq p(v) \end{cases} \qquad \forall\ d \leq P$$

$$f_d^{\emptyset}(R) = \begin{cases} c+1 & 1 \leq d \leq P \\ 0 & d = 0 \end{cases} \qquad \forall\ d \leq P$$

  else:

    for $J \in$ children(R):

      AlgTDC($(T(J), \mathcal{V})$) (e)

      if R is Introduce $(V_R = V_J \cup \{v\})$: (b)

        for $d \in [0, P]$:

          $f_d^S(R) = f_d^S(J)\ \ \forall S \in U_J$

          $f_d^{S\cup\{v\}}(R) = w(v) + f_{d-p(v)}^S(J)$

            $\forall S \in U_J :\ (S \cup \{v\}$ IS in $G) \wedge (w(v) + \sum_{i\in S} w(i) \leq c)$

      else if R is Forget $(V_J = V_R \cup \{v\})$: (c)

        for $d \in [0, P]$:

          $f_d^S(R) = \min\{f_d^S(J), f_d^{S\cup\{v\}}(J)\}$

            $\forall S \in U_J :\ (S \cup \{v\}$ IS in $G) \wedge (w(v) + \sum_{i\in S} w(i) \leq c)$

          $f_d^S(R) = f_d^S(J)$

            $\forall S \in U_J :\ (S \cup \{v\}$ DS in $G) \vee (w(v) + \sum_{i\in S} w(i) > c)$

      else if R is Join : (d)

        for $d \in [0, P]$:

          if J is the first child of R being processed:

            $f_d^S(R) = f_d^S(J)\ \ \forall S \in U_J$

          else:

            $f_d^S(R) = \min_k\{f_{d-k}^S(R) + f_k^S(J) \mid k \in [0, d]\}\ \ \forall\ S \in U_J$ (f)

Inner vertices.

Introduce with respect to vertex v. Let I be an Introduce (part (b) in AlgTDC) with child vertex J and let us *assume* that AlgTDC computed the optimal solution for G_J. We first consider all feasible subsets S from G_J. These subsets are also in G_I so the algorithm takes the optimal solution calculated so far from the limited tree-decomposition $(T(J), \mathcal{V})$ which is optimal by assumption. The only difference between G_I and G_J lies in vertex v and edges adjacent to v, but so far only solutions excluding v were considered.

In a next step all subsets $S \cup \{v\}$ of G_I that are independent sets in G and therefore in G_I are considered if they fulfill the capacity constraint. But S is subset of G_J and by the property of tree-decompositions v was not in $(T(J), \mathcal{V})$. As v is included in the knapsack solution, a profit of $d - p(v)$ is taken with minimal weight from G_J ($f_d^{S\cup\{v\}}(I) = w(v) + f_{d-p(v)}^S(J)$). Furthermore v is compatible with all vertices that lead to $f_{d-p(v)}^S(J)$: for the set S this is true by explicit testing. So let us *assume* that there is a vertex $i \in G_J$ packed that is not in S but adjacent to v. By combining the properties 1 and 2 of tree-decompositions a contradiction follows. The optimum for $f_d^{\emptyset}(I)$ follows by the same arguments.

Forget with respect to vertex v. Let I be a Forget with child vertex J and let us *assume* that AlgTDC computed the optimal solution for G_J. Then $G_I = G_J$ and in part (c) of AlgTDC we compute the optimal solution for all feasible subsets S of V_I by using solutions from $(T(J), \mathcal{V})$ that are by assumption optimal.

Join. Let I be a Join with children J_1 and J_2. This corresponds to part (d) in AlgTDC. For each feasible set $S \in V_I$ AlgTDC calculates the minimum weight of a knapsack solution leading to a profit of d by taking the minimum over all possible combinations of weights from the subgraphs G_{J_1} and G_{J_2} ($f_d^S(I) = \min_k\{f_{d-k}^S(I) + f_k^S(J_i) \mid k \in [0, d]\}$). Clearly by assumption both of these parts are optimal.

It remains to show that this combination of vertices coming from two different subgraphs of G is feasible. Clearly when restricting the knapsack solution leading to the optimal solution of $f_d^S(I)$ to G_{J_1}, all these items are feasible by assumption (they are calculated in the limited tree-decomposition $(T(J_1), \mathcal{V})$. The same is true for G_{J_2}. So let us *assume* that there is a vertex $v_1 \in G_{J_1}$ and a vertex $v_2 \in G_{J_2}$ which both belong to the knapsack solution leading to a minimal weight of $f_d^S(I)$ for profit d and $v_i \notin V_I : \ i \in \{1, 2\}$ so that v_1 and v_2 are not allowed to be packed together. Then they are adjacent, so there has to be some vertex $L \in T$ with the property that $\{v_1, v_2\} \subseteq V_L$: w.l.o.g if $L \in T(J_1)$ then property 3 of the definition of tree-decompositions implies that $v_2 \in I$, a contradiction. If $L \notin T(I)$ then a contradiction follows with the same argument. $\square$

Theorem 4 *Algorithm AlgTDC can be implemented to run in $O(nP^2)$ time and $O(\log(n)P + n)$ space given a nice tree-decomposition $(T, \mathcal{V})$ with $O(n)$ vertices.*

Proof.

Time Complexity.

Since the nice tree-decomposition has $O(n)$ vertices, AlgTDC consists of $O(n)$ recursive calls (e) in Algorithm 2. Since for each vertex V_I, $I \in T$, at most 2^{k+1} subsets of vertices in G have to be considered, all the checks in AlgTDC that evaluate if a subset $S \subseteq V_I$ describes a feasible knapsack solution can be performed in constant time (k is a constant). The other relevant part for the time complexity is described by (f). In this statement P^2 combinations of weights are used for calculating the minimum weight of a solution leading to a profit of d. Combining these parts, the time complexity follows.

Space Complexity.

For each vertex $I \in T$, each feasible subset $S \subseteq V_I$ and each profit d the minimum weight is stored, leading to a space complexity of $O(n * P)$. By the bounded treewidth the number of independent sets at each vertex I is constant. By adapting the algorithm according to Lemma 1 this can be reduced to $O(\log(n)P + n)$. □

4 Chordal Graphs as Conflict Graphs

4.1 Definitions

A graph $G = (V, E)$ is called *chordal graph*, if it does not contain induced cycles other than triangles ([7]). A clique of a graph G is a complete subgraph of G, a maximal clique is a clique, that is not properly contained in any other clique. A *clique tree* $T = (\mathcal{K}, \mathcal{E})$ of a chordal graph G is a tree that has all the maximal cliques K of G as vertices and for each vertex $v \in G$ all the cliques K containing v induce a subtree in T ([2]). When using a capital letter we will always denote a vertex in the clique tree T corresponding to a maximal clique in G, when using a lowercase letter we refer to a vertex in G. It has to be mentioned that the clique tree of a chordal graph can be computed using $O(n + m)$ time and space ([11]) where m describes the number of edges in G.

Having a clique tree T and choosing two adjacent vertices K and K', then $T_K^{(KK')}$ denotes the subtree that results from T when removing the edge between K and K' and including K. $S_{(KK')} \subset V$ is defined as the intersection between the cliques K and K' ($S_{(KK')} = K \cap K' = S_{(K'K)}$). Furthermore, by summing up over all cliques C in $T_K^{(KK')}$ we define the vertex set $V_K^{(KK')} \subset V$ by

$$V_K^{(KK')} = \left(\bigcup_{C \in T_K^{(KK')}} \{v \in C\} \right) - S_{(KK')}.$$

$V_K^{(KK')}$ therefore denotes the vertices in G that are in the cliques represented by vertices of the subtree $T_K^{(KK')}$, but excluding all the vertices that are in $S_{(KK')}$. These definitions refine [2].

4.2 Algorithm AlgCh

The basic idea for treating chordal graphs as conflict graphs in KCG lies in utilizing the special separation properties of the clique-tree of a chordal graph. The algorithm presented in this section uses these properties by means of the following two lemmas, that can be found with detailed proofs in [2].

Lemma 2 *The sets $V_K^{(KK')}$, $V_{K'}^{(KK')}$ and $S_{(KK')}$ form a partition of the vertices V in G.*

Lemma 3 *$S_{(KK')}$ is a minimal vw-separator for every pair of vertices $v \in V_K^{(KK')}$ and $w \in V_{K'}^{(KK')}$.*

Let $G = (V, E)$ be a chordal graph, $T(R)$ a clique-tree of G with vertex R as root vertex (by definition R is a maximal clique of G). Furthermore let $T(I)$ be the induced subtree of $T(R)$ with root I for some clique I (as defined in Section 1.2). $f_d^v(I)$ is defined as the minimum weight of the knapsack including item $v \in I$ with total profit equal to d, while considering only the subtree of the clique tree of G that has I as its root. Then a recursive algorithm that solves KCG for chordal graphs as conflict graphs is given by Algorithm 3. If the algorithm is executed with some vertex R' seen as root vertex of some clique tree T of G (AlgCh($T(R')$)), the optimal solution of KCP with the chordal conflict graph G is computed and stored in one of the $f_d^v(R')$ with $v \in R'$ or in $f_d^{\emptyset}(R')$.

Theorem 5 *Algorithm AlgCh solves KCG with a chordal conflict graph to optimality.*

Proof.
By Lemma 3, for two maximal cliques I and J which are adjacent in a clique tree representation T of G, $S_{(IJ)}$ is a separator for all vertices $a \in (I \setminus S_{(IJ)})$ and $b \in (J \setminus S_{(IJ)})$ in G. If we consider a leaf vertex L_1 of T and its parent vertex P_1, by Lemma 2 the graph G can be decomposed into three parts (not necessarily components), namely $V_{L_1}^{(L_1P_1)}$, $S_{(L_1P_1)}$ and $V_{P_1}^{(L_1P_1)}$(seen as induced subgraphs). Obviously when including a vertex $v \in V_{L_1}^{(L_1P_1)}$ in the knapsack, this vertex cannot be in conflict with any vertex $\overline{v} \in (G \setminus L_1)$. When considering any parent vertex J of T with k child vertices $(I_1 \dots I_k)$, then the graph G can be decomposed into $k+2$ parts, namely $V_{I_1}^{(I_1J)}, \dots, V_{I_k}^{(I_kJ)}$, $(S_{(I_1J)} \cup \dots \cup S_{(I_kJ)})$ and $(G \setminus (T_{I_1}^{(I_1J)} \cup \dots \cup T_{I_k}^{(I_kJ)}))$ (here the subtrees are seen as induced subgraphs of G). By recursively applying this decomposition idea in the processing order

Algorithm 3 AlgCh

```
AlgCh(T(R)):
  if R is leaf:                                                          (a)
      f_d^v(R) = { w(v)   d = p(v)
                 { c+1    d ≠ p(v)          ∀ v ∈ R   ∧   ∀ d ≤ P
      f_d^∅(R) = { c+1    1 ≤ d ≤ P
                 { 0      d = 0             ∀ d ≤ P
  else:
    for J ∈ children(R):
      AlgCh(T(J))
      if J is the first child of R being processed:                     (b)
        for v ∈ R
          if v ∈ S_(RJ):
            for d ∈ [0, P] :
              f_d^v(R) = f_d^v(J)
          else:
            for d ∈ [0, p(v) − 1] :
              f_d^v(R) = c + 1
            for d ∈ [p(v), P] :
              f_d^v(R) = w(v) + min_i{f_{d−p(v)}^i(J) | i ∈ (J \ S_(RJ) ∪ ∅)}
        f_d^∅(R) = min_i{f_d^i(J) | i ∈ (J \ S_(RJ) ∪ ∅)}     ∀ d ≤ P        (d)
      else:                                                              (c)
        for v ∈ R:
          if v ∈ S_(RJ) :
            for d ∈ [p(v), P] :
              f_d^v(R) = min_k{f_k^v(R) + f_{d−k+p(v)}^v(J) | k ∈ [p(v), d]}
              f_d^v(R) = f_d^v(R) − w(v)
          else:
            for d ∈ [p(v), P] :
              f_d^v(R) = min_{i,k}{f_k^v(R) + f_{d−k}^i(J) | k ∈ [p(v), d],     (d)
                                   i ∈ (J \ S_(RJ) ∪ ∅)}
        for d ∈ [0, P] :
          f_d^∅(R) = min_{i,k}{f_k^∅(R) + f_{d−k}^i(J) | k ∈ [p(v), d],
                               i ∈ (J \ S_(RJ) ∪ ∅)}
```

of AlgCh on the maximal cliques of T (depth-first), we can consider G as being iteratively constructed by the resulting parts.

In the remainder of the proof, we show that the algorithm computes the optimum for each subgraph of G that is induced by $T(I)$, for some clique I. This is done by an induction like procedure, i.e. we first show the optimality of the leaf vertices. Afterwards we show the optimality of the subgraph induced by $T(I)$ under the assumption of optimality of all trees $T(J_i)$ with J_i being child vertex of I. Since this procedure finishes at the subgraph induced by $T = T(R)$, which obviously equals G, the theorem follows.

Leaf vertices. The first vertices completed by the algorithm are leaf vertices of T, namely $(L_1 \dots L_m)$ for some m with parent vertex P. The induced subgraphs $(T_{L_1}^{(L_1 P)} \dots T_{L_m}^{(L_m P)})$ are exactly $(L_1 \dots L_m)$. So by (a) in Algorithm 3 $f_d^v(I)$ represents the optimal solution for all profits d and all $v \in I$ given that $I \in \{L_1 \dots L_m\}$.

Inner vertices. *Assume* that AlgCh computes the optima for all subtrees $(T_1^{h-1} \dots T_l^{h-1})$ with height up to $h-1$. We will show that also the subtrees $T_1^h \dots T_k^h$ with height h, being supergraphs of some of the $(T_1^{h-1} \dots T_l^{h-1})$, will be solved to optimality.

Case 1. *Assume* that the tree $T^h(I)$ with root vertex I is a supergraph of exactly one tree with height up to $h-1$ and root J $(T^{h-1}(J) = T_J^{IJ})$. Clearly this means that I has J as its only child vertex (Figure 1).

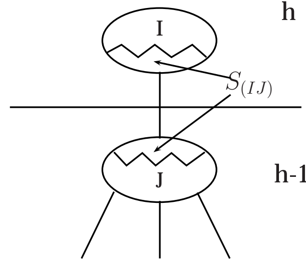

Figure 1: Case 1 in the proof of Theorem 5

In the algorithm in this case we are in the "if part" corresponding to (b). There we calculate $f_d^v(I)$ for each profit d and vertex $v \in I$. If $v \in S_{(IJ)}$ by the optimality of $f_d^v(J)$ also $f_d^v(I)$ has to be optimal (v was already considered in the clique J and is in conflict with all other vertices in I). If $v \notin S_{(IJ)}$, $f_d^v(I)$ is calculated by $f_d^v(I) = w(v) + \min_i\{f_{d-p(v)}^i(J) \mid i \in (J \setminus S_{(IJ)} \cup \emptyset)\}$. By Lemma 2, v cannot be in $T^{h-1}(J)$. By definition of $f_d^v(I)$ we include item v into the knapsack, so we have to add the weight $w(v)$ to $f_d^v(I)$. As v adds a value equal to $p(v)$ to the profit d, we take the best solution so far (by assumption)

represented by i of $(J \setminus S_{(IJ)} \cup \emptyset)$ with minimum weight $f^i_{d-p(v)}(J)$ leading to a profit of $d - p(v)$. The same argument works with $f^{\emptyset}_d(I)$, which means that we do not pack any item included in the maximal clique I. So *Case 1* is proved.

Case 2. *Assume* that the tree $T^h(I)$ with root vertex I is supergraph of k trees with height at most $h-1$ $(T^{h-1}(J_1)\dots T^{h-1}(J_k))$. This means that I has $(J_1 \dots J_k)$ as its child vertices. The algorithm merges $T^h(I)$ with its subtrees $(T^{h-1}(J_1)\dots T^{h-1}(J_k))$. The merging of $T^{h-1}(J_1)$ to $T^h(I)$ is optimal by *Case 1*, so in AlgCh we are in the part denoted by (c). Now we *assume* that the merging procedure is done optimally for the trees $(T^{h-1}(J_1)\dots T^{h-1}(J_{l-1}))$ for some $l \geq 2$ (P_1 in Figure 2).

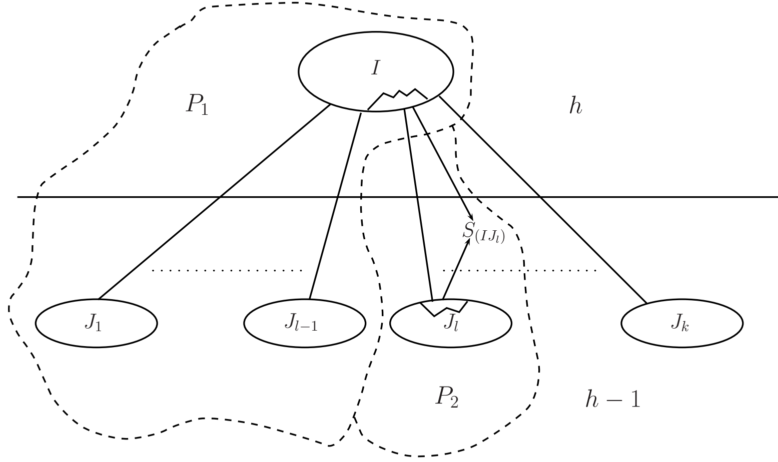

Figure 2: Case 2 in the proof of Theorem 5

By Lemma 3, $V^{(J_l I)}_{J_l} = (T^{h-1}(J_l) \setminus (S_{(J_l I)})$ (seen as induced subgraph of G) is separated by $S_{(J_l I)}$ from all vertices that were considered in the merging procedure so far. Furthermore all $f^v_d(J_l)$ were calculated optimally by assumption (P_2 in Figure 2). If $v \in S_{IJ_l}$, $f^v_d(I)$ is calculated as the minimum over the set $\{f^v_k(I) + f^v_{d-k+p(v)}(J_l)\}$ with all combinations of profits that lead to a total profit of d. By assumption both expressions in this set are optimal. If $v \notin S_{IJ_l}$, $f^v_d(I)$ is calculated as the minimum over all combinations of profits and feasible vertex combinations leading to a total profit of d, i.e. $\min_{i,k}\{f^v_k(R) + f^i_{d-k}(J) \mid k \in [p(v), d], i \in (J \setminus S_{(IJ)} \cup \emptyset)\}$. Again by assumption both expressions in this calculation are optimal. The same argument works with $f^{\emptyset}_d(I)$. So *Case 2* is proved. □

Theorem 6 *Algorithm AlgCh can be implemented to run in $O((n+m)P^2)$ time and $O(\min\{m, n\log n\} * P + m)$ space.*

Proof.

Time Complexity. By [11] the following holds:

$$\sum_{K \in T} |K| \leq n + m \qquad (**)$$

K denotes the maximal cliques of G represented as vertices of T and $|K|$ the number of vertices in clique K.

As the algorithm traverses each vertex $K \in T$ and each vertex $v \in K$ once, by (**) no more than $n + m$ steps are executed in AlgCh. Furthermore at each of these steps $P * P$ combinations of profits are considered and in part (d) of AlgCh the minimum over $O(n)$ vertices of the corresponding child clique is computed, resulting in a time complexity of $O((n + m) * n * P^2)$.

But this time complexity can be reduced to $O((n + m) * P^2)$ by the following observation: Referring to (d) we argued that each vertex of a clique K in T is combined with $O(n)$ vertices in its child clique K'. But each vertex $v \in G$ has the property in T to be in some clique J with parent clique I, so that $v \notin S_{(IJ)}$ and this happens exactly once for each vertex (with the exception of vertices in the root clique). Therefore during the whole algorithm in the part described by (d), each v is used at most once for updating the parent clique with its child clique, where v is seen as part of the child clique.

Space Complexity. For every induced subtree $T(K)$ of T with root K and each vertex $v \in K$ the algorithm stores the optimal solution calculated so far, given that $v \in K$ is included in the knapsack, namely $f_d^v(K)$. Thereby exactly $P + 1$ different profits are considered. By $(**)$ an overall space complexity of $O((n + m) * P)$ follows.

On the other hand, to apply the space reduction technique of Lemma 1, one has to consider that the tree T has at most n vertices K, each of them corresponding to a clique consisting of at most n vertices of G. By the same arguments as before a space complexity of $O(n^2 * P)$ follows. By Lemma 1 this can be reduced to $O(n \log(n) * P)$. It has to be pointed out that now the m edges of G, resulting simply from storing a clique tree, are missing.

Taking the minimum of these two expressions and considering that $m \geq n$ for a connected graph, the result follows.

□

Remark 1 AlgCh also solves the maximum weight independent set problem for chordal graphs by setting the profits and weights of each item to 1 and the capacity of the knapsack to n in KCG. But it has to be mentioned that the time and space complexity of AlgCh is outperformed by the classical approach described in [10].

5 Fully Polynomial Time Approximation Schemes

The three algorithms described in this paper all do admit FPTASs. To see this we first briefly review the crucial parts of the standard FPTAS for the classical 0-1 knapsack problem, i.e. an approximation algorithm with a performance guarantee of $(1-\varepsilon)$ and a running time polynomial in n and $1/\varepsilon$, which can be found in [19].

Following this approach, the dynamic programming algorithms are executed on an instance characterized by *scaled profits*, i.e. p_j is replaced by $\tilde{p}_j := \lfloor \frac{p_j}{K} \rfloor$, for some K. Let $\tilde{X}$ be the solution set representing an optimal solution on this scaled instance. Generally, this set will be different from the solution set X^* which optimizes the original instance with a solution value of z^*. Furthermore, let z^A be the solution value of the set $\tilde{X}$ on the original instance. Clearly $z^A \leq z^*$. Then completely analogous to [19] one gets the following inequality:

$$z^A = \sum_{j \in \tilde{X}} p_j \geq \sum_{j \in \tilde{X}} K \lfloor \frac{p_j}{K} \rfloor \geq \sum_{j \in X^*} K \lfloor \frac{p_j}{K} \rfloor \geq \sum_{j \in X^*} K (\frac{p_j}{K} - 1) =$$
$$= \sum_{j \in X^*} (p_j - K) = z^* - |X^*| \, K$$

As a consequence one gets the following bound on the relative error represented by ε:

$$\frac{z^* - z^A}{z^*} \leq \frac{|X^*| \, K}{z^*} \leq \varepsilon.$$

If now K is chosen to be less or equal to $\varepsilon \frac{z^*}{|X^*|}$ the desired performance guarantee of $1-\varepsilon$ follows. Setting $K := \varepsilon \frac{p_{\max}}{n}$ this obviously can be achieved.

Going back to the three algorithms of the current paper it can be seen easily that the only ingredients required for the above construction are: (i) an exact dynamic programming algorithm, (ii) an upper bound on the cardinality of the optimal solution set, (iii) a lower bound on the optimal solution value. All three aspects are trivially fulfilled by Algorithm 1, 2 and 3 since $|X^*| \leq n$ and $z^* \geq p_{\max}$ always hold.

Furthermore, every occurrence of the trivial upper bound P in the running time and space complexities of the presented algorithms can be replaced for the scaled instance in the following way: the optimal solution value $\tilde{z}$ of the scaled problem instance is bounded by

$$\tilde{z} \leq n \tilde{p}_{\max} \leq n \frac{p_{\max}}{K} = \frac{n^2}{\varepsilon} \, .$$

Therefore, in the running time bound for each of the three algorithms one can replace the factor P by $\frac{n^2}{\varepsilon}$ for the scaled instances and thus the required complexity of an FPTAS is attained.

6 Conclusion

After considering chordal graphs the natural next step would be the more general class of *perfect graphs*, since the maximum weighted independent set problem is efficiently solvable on perfects graphs (cf. [14]) as well as on all the classes we considered. However, this question can be settled by a result due to Milanič and Monnot [20].

Theorem 7 *KCG is strongly NP-hard on perfect graphs.*

Proof. It was shown in [20] that the exact weighted independent set problem (EWIS) for perfect graphs is strongly NP-complete. In fact it was shown, that EWIS is already strongly NP-complete for cubic bipartite graphs. Having an instance of EWIS one asks if a given independent set with weight exactly w exists where each vertex j has weight w_j. Now consider an instance of KCG that results by setting the profits p_j equal to w_j and the capacity c to w. Then by solving this KCG-instance one can immediately answer the corresponding EWIS-instance. □

Clearly the result of Milanič and Monnot also implies, that KCG is strongly NP-hard on general bipartite graphs.

It was pointed out in the Introduction that our algorithms report only the optimal solution values and keeping track of the corresponding solution sets would require an additional factor of $\log n$ for all complexity results. However, this factor can be avoided by applying an adaption of the general recursive storage reduction scheme given by Pferschy [22] (see also [19, ch. 3.3]).

Going into the technical details of this modification is beyond the scope of this paper where we concentrate on identifying polynomially solvable special cases. The crucial point is that a given problem instance can be partitioned into two instances of roughly equal size which are then solved recursively and their solutions combined to an overall optimal solution. Such a bipartitioning can be achieved for all three graph classes by splitting the respective tree into two parts after computing the median vertex of the tree.

Acknowledgment

We would like to thank Martin Milanič for pointing out reference [20] and giving valuable comments on our work.

References

[1] B. S. Baker and E. G. Coffman. Mutual exclusion scheduling. *Theoretical Computer Science*, 162(2):225–243, 1996.

[2] J. R. S. Blair and B. Peyton. An introduction to chordal graphs and clique trees. In A. George, J. R. Gilbert, and J. H. U. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 1–29, New York, 1993. Springer.

[3] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[4] H. L. Bodlaender and K. Jansen. On the complexity of scheduling incompatible jobs with unit-times. In *MFCS '93: Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science*, pages 291–300, London, UK, 1993. Springer.

[5] H. L. Bodlaender, K. Jansen, and G. J. Woeginger. Scheduling with incompatible jobs. *Discrete Applied Mathematics*, 55(3):219–232, 1994.

[6] H. L. Bodlaender and A. M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[7] R Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, Heidelberg, 2006.

[8] L. Epstein and A. Levin. On bin packing with conflicts. In *Approximation and Online Algorithms, Springer Lecture Notes in Computer Science 4368*, pages 160–173. Springer, 2007.

[9] L. Epstein, A. Levin, and R. van Stee. Two-dimensional packing with conflicts. *Acta Informatica*, 45(3):155–175, 2008.

[10] A. Frank. Some polynomial algorithms for certain graphs and hypergraphs. In *Proceedings of the Fifth British Combinatorial Conference, Aberdeen*, pages 211–226. Utilitas Mathematica Publishing, 1975.

[11] P Galinier, M. Habib, and C. Paul. Chordal graphs and their clique graphs. In *WG '95: Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 358–371, London, UK, 1995. Springer.

[12] M. Gendreau, G. Laporte, and F. Semet. Heuristics for the bin packing problem with conflicts. *Computers and Operations Research*, 31:347–358, 2004.

[13] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, 2004.

[14] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[15] M. Hifi and M. Michrafy. A reactive local search-based algorithm for the disjunctively constrained knapsack problem. *Journal of the Operational Research Society*, 57:718–726(9), 2006.

[16] M. Hifi and M. Michrafy. Reduction strategies and exact algorithms for the disjunctively constrained knapsack problem. *Computers and Operations Research*, 34(9):2657–2673, 2007.

[17] K. Jansen. An approximation scheme for bin packing with conflicts. In *SWAT '98: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, pages 35–46, London, UK, 1998. Springer.

[18] K. Jansen and S. Öhring. Approximation algorithms for time constrained scheduling. *Information and Copmutation*, 132(2):85–108, 1997.

[19] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[20] M. Milanič and J. Monnot. *Combinatorial Optimization - Theoretical Computer Science : interfaces and Perspectives*, chapter The complexity of the exact weighted independent set problem, pages 393–432. Wiley-ISTE, 2008.

[21] F. Muritiba, M. Iori, E. Malagut, and P. Toth. Algorithms for the bin packing problem with conflicts. Technical Report OR-08-7, DEIS - University of Bologna.

[22] U. Pferschy. Dynamic programming revisited: improving knapsack algorithms. *Computing*, 63:419–430, 1999.

[23] T. Yamada, S. Kataoka, and K. Watanabe. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. *Information Processing Society of Japan Journal*, 43:2864–2870, 2002.