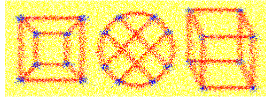
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 13, no. 2, pp. 233–249 (2009)

The Knapsack Problem with Conflict Graphs

Ulrich Pferschy Joachim Schauer

University of Graz, Department of Statistics and Operations Research,
Universitaetsstr. 15, A-8010 Graz, Austria

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 graphs, exact and heuristic algorithms were proposed in the past. While the problem is strongly NP-hard in general, we present pseudopolynomial algorithms for two special graph classes, namely graphs of bounded treewidth (including trees and series-parallel graphs) and chordal graphs. From these algorithms we can easily derive fully polynomial time approximation schemes (FPTAS).

Submitted:	Reviewed:	Revised:	Accepted:
October 2008	May 2009	July 2009	July 2009
	Final:	Published:	
	August 2009	October 2009	

Article type: Regular paper

Communicated by: D. Wagner

E-mail addresses: pferschy@uni-graz.at (Ulrich Pferschy) joachim.schauer@uni-graz.at (Joachim Schauer)

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. We define a trivial upper bound P on the total profit of the knapsack as $P = \sum_{i=1}^{n} p_i$. Now we state the following ILP formulation of KCG:

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

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

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

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

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). However, w.l.o.g. we can restrict KCG to connected conflict graphs by introducing a dummy item $n + 1$ with weight $w_{n+1} = c$ and profit $p_{n+1} = 0$ and inserting edges from vertex $n + 1$ to every other vertex. This makes 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 $\mathcal{NP}$-hard (for a given instance of the knapsack problem introduce a star graph as a conflict graph centered at the above mentioned dummy vertex).

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 $\mathcal{NP}$-hard (cf. [7]) and does not permit pseudo-polynomial algorithms (under $\mathcal{P} \neq \mathcal{NP}$).

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. From these algorithms we can immediately attain fully polynomial time approximation schemes (FPTAS). In the following sections we will show that graphs of bounded treewidth (including trees) and chordal graphs (including interval graphs [7]) used as conflict graphs in KCG admit pseudo-polynomial algorithms as well as FPTASs. However, for perfect conflict graphs KCG can be shown to be strongly $\mathcal{NP}$-hard and hence does not permit an FPTAS.

Note that for unconnected graphs the special properties of some of these graph classes would be no longer valid after adding the 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. The computation of the corresponding *solution set of items* and other storage issues will be discussed in Section 4.

The first paper dealing with KCG that we are aware of is Yamada et al. [15] from 2002. The authors construct a branch-and-bound algorithm with lower bounds based on a heuristic and upper bounds derived from a Lagrangean relaxation of the conflict conditions (3). Hifi and Michrafy [9] recently presented a metaheuristic approach consisting of a reactive local search algorithm combined with a tabu list. The same authors develop three different exact algorithms in [10] invoking reduction and constraint combination. They all compare very favorably to CPLEX.

Conflict graphs were also considered for other combinatorial optimization problems such as bin packing and scheduling problems, but we refrain from giving a full review of these related problems.

2 Graphs of Bounded Treewidth

In this section we treat graphs of bounded treewidth, for example trees, series-parallel graphs, outerplanar graphs or Halin graphs ([2]). We show that for KCG on a conflict graph with a given tree-decomposition of constant treewidth k, there exists a dynamic programming algorithm with $O(nP^2)$ running time.

In [4] 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 $I \in T$ 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 $\max\{|V_I| - 1 : \; I \in T\}$. The *treewidth* of G is the smallest width of any tree-decomposition of G ([4]). By [3] deciding whether a tree-decomposition of treewidth at most k exists, and if so, finding such a tree-decomposition can be done in linear time (if k is seen as a constant and not as part of the input).

2.1 Trees as Conflict Graphs

Since all algorithms in this paper are based on the exploration of a tree, we briefly discuss explicitly the special case of trees T as conflict graphs, i.e. graphs of treewidth one. This should facilitate the understanding of the algorithmic ideas of the paper and in particular the space reduction described in Section 4.

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 a dynamic programming algorithm for trees as conflict graphs 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. For applying *dynamic programming by reaching* we use two arrays: $z_i(d)$ describes a solution with minimal weight found in the subtree $T(i)$[1] 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 $T(i)$ that leads to a profit of d with item i *excluded* from the knapsack solution. The update operations applied to these arrays when merging a leaf to its parent is obvious. Without going into details the following statement can be derived from the dynamic programming scheme for the standard knapsack problem.

Theorem 1 *KCG with a tree as conflict graph can be solved in $O(nP^2)$ time and $O(nP)$ space.* □

[1] $T(i)$ defines the induced subtree of T with root i, where for every $j \in T(i)$, i lies on the unique path from j to r.

2.2 The Algorithm for the General Case

For algorithmic purposes it is useful to consider a specially structured tree-decomposition, namely a *nice tree-decomposition*. This tree-decomposition has the property of being a binary tree in which adjacent vertices differ by at most one vertex in the underlying graph G. One vertex R is considered to be the root of T and each vertex $I \in T$ is of one of the following four types ([3]):

- 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 $V_I = V_{J_1} = V_{J_2}$
- 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 [3] a nice tree-decomposition with $O(n)$ tree vertices 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 no 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 $\bigcup_{J \in T(I)} V_J$.

Let $(T, \mathcal{V})$ be a nice tree-decomposition of G of bounded treewidth k. Let U_J be the set of subsets S of vertices from V_J with the property that S is an independent set (IS) in G and $\sum_{i \in S} w(i) \leq c$ (U_J includes the empty set $\emptyset$). A set of vertices in G that is not an independent set is abbreviated by DS. 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, following an idea presented in [3], KCG is solved by algorithm AlgTDC which processes the tree-decomposition in depth-first order.

Theorem 2 *Algorithm AlgTDC solves KCG with a conflict graph G of bounded treewidth k to optimality.*

Proof: Given a nice tree-decomposition we will show by an induction like procedure that for each vertex $I \in T$ AlgTDC computes an optimal solution for the subgraph G_I of G. 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 1 when including v into the knapsack solution (constrained to G_I) the only possible profit $d = p(v)$ has minimal weight $w(v)$.

Inner vertices.
Introduce with respect to vertex v. Let I be an Introduce (part (b) in AlgTDC)

Algorithm 1 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)

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. Since these subsets are also in G_I the algorithm takes the optimal solution calculated 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 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 $(T(J), \mathcal{V})$ which are optimal by assumption.

Join. Let I be a Join with children J_1 and J_2 ((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 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. The same is true for G_{J_2}. Now assume that there are vertices $v_1 \in G_{J_1}$ and $v_2 \in G_{J_2}$ both belonging to the knapsack solution leading to a minimal weight of $f_d^S(I)$ for profit d and v_1 and v_2 are not in V_I and furthermore they 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. □

Theorem 3 *Algorithm AlgTDC can be implemented to run in $O(nP^2)$ time and $O(nP)$ 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 1. Since for each vertex V_I, $I \in T$, at most 2^{k+1} subsets of vertices in G have to be considered, AlgTDC performs a constant number of feasibility tests for subsets $S \subseteq V_I$. The other relevant part for the time complexity is described by (f) where 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 yielding a space complexity of $O(nP)$. By the bounded treewidth the number of independent sets at each vertex I is constant. □

3 Chordal Graphs as Conflict Graphs

3.1 Definitions

A graph $G = (V, E)$ is called *chordal graph*, if it does not contain induced cycles other than triangles ([4]). 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 ([1]). 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 ([6]) 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 [1].

3.2 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. 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 ($V(G) = V$).

The algorithm presented in this section uses these properties by means of the following two lemmas, that can be found with detailed proofs in [1].

Lemma 1 *The sets $V_K^{(KK')}$, $V_{K'}^{(KK')}$ and $S_{(KK')}$ form a partition of the vertices V in G.*

Lemma 2 *$S_{(KK')}$ 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. $f_d^v(I)$ 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 2. 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 4 *Algorithm AlgCh solves KCG with a chordal conflict graph to optimality.*

Proof: By Lemma 2, 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 1 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 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, starting with the leaf vertices and moving upwards in the tree thus showing 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. The proof finishes with the subgraph induced by $T = T(R)$, which obviously equals G.

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 $(T_{L_1}^{(L_1P)} \dots T_{L_m}^{(L_mP)})$ are exactly $(L_1 \dots L_m)$. So by (a) in Algorithm 2 $f_d^v(I)$ represents the optimal solution for all profits d and all $v \in I$ given that $I \in \{L_1 \dots L_m\}$.

Algorithm 2 AlgCh

AlgCh($T(R)$):

  if R is leaf: (a)

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

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

  else:

    for $J \in$ children(R):

      AlgCh($T(J)$)

      if J is the first child of R being processed: (b)

        for $v \in R$

          if $v \in S_{(RJ)}$:

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

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

          else:

            for $d \in [0, p(v) - 1]$:

              $f_d^v(R) = c + 1$

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

              $f_d^v(R) = w(v) + \min_i\{f_{d-p(v)}^i(J) \mid i \in (J \setminus S_{(RJ)} \cup \emptyset)\}$

        $f_d^{\emptyset}(R) = \min_i\{f_d^i(J) \mid i \in (J \setminus S_{(RJ)} \cup \emptyset)\} \qquad \forall\, d \leq P$ (d)

      else: (c)

        for $v \in R$:

          if $v \in S_{(RJ)}$:

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

              $f_d^v(R) = \min_k\{f_k^v(R) + f_{d-k+p(v)}^v(J) \mid k \in [p(v), d]\}$

              $f_d^v(R) = f_d^v(R) - w(v)$

          else:

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

              $f_d^v(R) = \min_{i,k}\{f_k^v(R) + f_{d-k}^i(J) \mid k \in [p(v), d],$ (d)
                $i \in (J \setminus S_{(RJ)} \cup \emptyset)\}$

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

          $f_d^{\emptyset}(R) = \min_{i,k}\{f_k^{\emptyset}(R) + f_{d-k}^i(J) \mid k \in [p(v), d],$
            $i \in (J \setminus S_{(RJ)} \cup \emptyset)\}$

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).

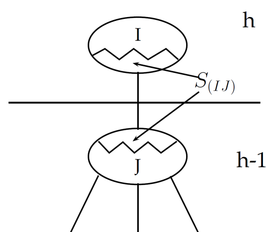

Figure 1: Case 1 in the proof of Theorem 4

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 1, 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 $f_{d-p(v)}^i(J)$ leading to a profit of $d - p(v)$. The same argument works with $f_d^{\emptyset}(I)$, 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).

By Lemma 2, $V_{J_l}^{(J_l I)} = (T^{h-1}(J_l) \setminus (S_{(J_l I)})$ (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 $f_d^v(J_l)$ were calculated optimally by assumption (P_2 in Figure 2). If $v \in S_{IJ_l}$, $f_d^v(I)$ is calculated as the minimum over the set $\{f_k^v(I) + f_{d-k+p(v)}^v(J_l)\}$ with all combinations of profits that lead to a total profit of d. By assumption both expressions in this set are optimal. If $v \notin S_{IJ_l}$, $f_d^v(I)$ is calculated as the minimum over all combinations of profits and feasible vertex combinations leading to a total profit of d, i.e.

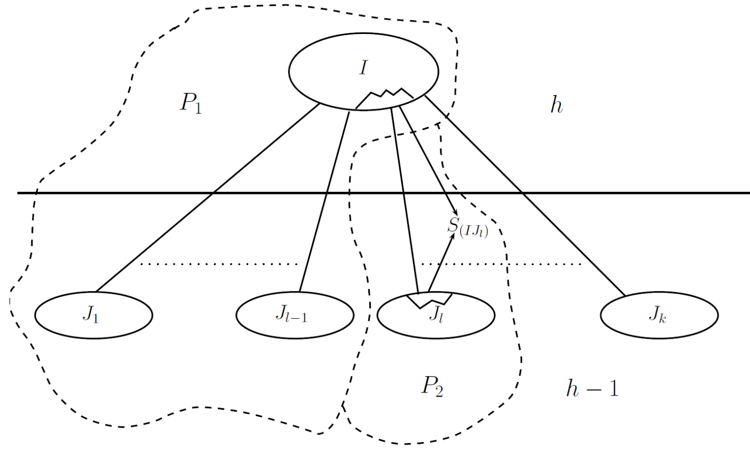

Figure 2: Case 2 in the proof of Theorem 4

$\min_{i,k}\{f_k^v(R) + f_{d-k}^i(J) \mid k \in [p(v), d], i \in (J \setminus S_{(IJ)} \cup \emptyset)\}$. Again by assumption both expressions in this calculation are optimal. The same argument works with $f_d^{\emptyset}(I)$. So *Case 2* is proved. □

Theorem 5 *Algorithm AlgCh can be implemented to run in $O((n+m)P^2)$ time and $O((n+m)P)$ space.*

Proof:

Time Complexity. Denoting by K the maximal cliques of G represented as vertices of T and by $|K|$ the number of vertices in clique K the following inequality holds ([6]):

$$\sum_{K \in T} |K| \leq n + m \tag{5}$$

As the algorithm traverses each vertex $K \in T$ and each vertex $v \in K$ once, by (5) no more than $n+m$ steps are executed in AlgCh. Furthermore at each of these steps $P \cdot P$ 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 $O((n+m)nP^2)$.

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 (5) an overall space complexity of $O((n+m)P)$ follows. □

Remark. 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. However, the complexity of AlgCh for this case is outperformed by the classical approach described in [5].

4 Space Requirements

Excessive memory consumption is frequently pointed out as a major drawback of dynamic programming approaches and approximation schemes based on these. Hence, we briefly discuss the space requirements of our algorithms and point out two general improvement techniques.

It was pointed out in the Introduction that our algorithms report only the optimal solution values. Keeping track of the corresponding solution sets would require an additional factor of $\log n$ for all complexity results if we used a binary encoding for each subset. However, this factor can be avoided by applying an adaption of the general recursive storage reduction scheme given by Pferschy [13] (see also [11, 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 both graph classes by splitting the respective tree into two parts after computing the median vertex of the tree.

Beside this technical aspect of set representation, we also sketch a general space reduction technique applicable to all algorithms in this paper, in fact to all algorithms following a similar depth first search strategy on a tree T. Assume that in every vertex of the tree we generate an array of length P. Merging a child vertex to its parent as in Section 2.1 requires $2O(P)$ space and the storage space used for the child vertex can be deallocated after the merging. This yields a straightforward space complexity of $O(nP)$.

The crucial point of the argument is the selection among the k child vertices $j_1 \dots j_k$ of parent i to be explored next in the depth-first order. Choosing a child vertex j_l with the largest induced subtree, i.e.

$$|T(j_l)| = \max_i \{|T(j_i)| : \; i \in 1 \dots k\} \tag{6}$$

yields the following improvement.

Lemma 3 *The tree T can be fully explored in depth-first order using at most $(\log_2(n)+1)O(P)$ space.*

Proof: For $n = 2$, the tree T contains two vertices which can be merged within $2O(P)$ space by assumption. Assuming 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 $T(i_1)$ of the root r of T. Obviously, all other subtrees must contain less than $n/2$ vertices.

Then by assumption the processing of $T(i_1)$ is done by using at most $(\log_2(n-1)+1)O(P)$ space. After merging $T(i_1)$ to r this space can be deallocated, but $O(P)$ space is used at vertex r, which has to be kept until the algorithm has finished. Then the processing of $T(i_j)$, $j \geq 2$, is done using by assumption at most $(\log_2(\frac{n}{2})+1)O(P) = \log_2(n)O(P)$ space. After merging each of these subtrees to r this space can be deallocated, but the $O(P)$ space used at vertex r has to be kept until completion of the algorithm. This yields a total space requirement of $(\log_2(n)+1)O(P)$. □

Since all algorithms in this paper follow in principle an exploration of a tree in depth-first order, condition (6) can be easily incorporated as a selection criterion. We do not explicate the full technical details but summarize the resulting improved space complexities in Table 1.

problem	space
KCG on trees	$O(n + P\log(n))$
KCG on graphs of bounded treewidth	$O(n + P\log(n))$
KCG on chordal graphs	$O(\min\{m, n\log(n)\}P + m)$

Table 1: Improved space complexities.

5 Approximation Results

The algorithms described in this paper all admit an FPTAS, i.e. an approximation algorithm with a performance guarantee of $(1-\varepsilon)$ and a running time polynomial in n and $1/\varepsilon$. Such an approximation scheme could be derived in a straightforward way from the standard FPTAS for the classical 0-1 knapsack problem which can be found in [11]. However, an FPTAS can be also immediately deduced from a more general result due to Pruhs and Woeginger [14]. Roughly speaking, they define the following subset selection problem:

Given a ground set X with n elements each of them with positive profit $p(x)$ for $x \in X$, we are looking for a feasible subset of X with maximum total profit. Assume that the feasibility of a subset can be decided in polynomial time. Then it is shown in [14] in a more general setting that if there exists an exact algorithm for this problem with running time polynomial in n and $P := \sum_{x\in X} p(x)$, then there exists also an FPTAS.

It is easy to see that KCG belongs to this family of subset selection problems. Moreover, all the algorithms of this paper fulfill the required condition of pseudo-polynomial running time. Hence we conclude:

Theorem 6 *There exists an FPTAS for KCG on conflict graphs of bounded treewidth and on cordal conflict graphs.* □

After deriving dynamic programming schemes and FPTASs for 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. [8]) as well as on all the classes we considered. However, this question can be settled by a result due to Milanič and Monnot [12].

Theorem 7 *KCG is strongly $\mathcal{NP}$-hard on perfect graphs.*

Proof: It was shown in [12] that the exact weighted independent set problem (EWIS) for perfect graphs is strongly $\mathcal{NP}$-complete. In fact it was shown, that EWIS is already strongly $\mathcal{NP}$-complete for bipartite graphs of degree at most 3. 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 $\mathcal{NP}$-hard on general bipartite graphs and rules out the existence of an FPTAS on perfect conflict graphs.

Acknowledgments

We would like to thank Martin Milanič for pointing out reference [12] and giving valuable comments on our work. We also thank Gerhard J. Woeginger for fruitful discussions. We are grateful for the comments of the referees which helped a lot to improve the exposition of the paper.

References

[1] 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*. Springer.

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

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

[4] R. Diestel. *Graph Theory*. Springer, 2006.

[5] 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.

[6] 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. Springer, 1995.

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] K. Pruhs and G.J. Woeginger. Approximation schemes for a class of subset selection problems. *Theoretical Computer Science*, 382(2):151–156, 2007.

[15] 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.