INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Approximation Schemes for a Class-Constrained Knapsack Problem

E. C. Xavier *F. K. Miyazawa*

Technical Report - IC-03-012 - Relatório Técnico

April - 2003 - Abril

Approximation Schemes for a Class-Constrained Knapsack Problem *

E.C. Xavier† F.K. Miyazawa‡

April 23, 2003

Abstract

We consider approximation algorithms for a class-constrained version of the knapsack problem: Given an integer K, a set of items S, each item with value, size and a class, find a subset of S of maximum total value such that items are grouped in compartments. Each compartment must have only items of one class and must be separated by the subsequent compartment by a wall division of size d. Moreover, two subsequent wall divisions must stay a distance of at least $d_{\min}$ and at most $d_{\max}$. The total size used by compartments and by wall divisions must be at most K. This problem have practical applications on cutting-stock problems.

Key Words: Approximation schemes, knapsack problem, class-constrained.

1 Introduction

We consider approximation algorithms for a class-constrained version of the knapsack problem which we call *Generalized-Knapsack* (G-KNAPSACK). Given an integer K, a set of items $S = \{1, \ldots, n\}$, each item i with value v_i, size s_i and a class c_i, wall divisions of size d, find a set $M \subseteq S$ of maximum total value and a partition of M into compartments $C_1, \ldots, C_k$, each compartment with size $\sum_{e \in C_i} s_e$ where the following conditions are valid: *(i)* items in the same compartment have the same class, *(ii)* two subsequent compartments are separated by a wall division, *(iii)* the total size used by compartments and by wall divisions must be at most K, *(iv)* each compartment size must be at least $d_{\min}$ and at most $d_{\max}$.

This problem has a practical motivation in the roll cutting in iron and steel industry. Ferreira et al. [3] present a cutting problem where a raw material roll must be cut into final rolls grouped by certain properties after two cutting phases. The rolls obtained after the first phase, called primary rolls, are submitted to different processing operations (tensioning, tempering, laminating, hardening etc.) before the second phase cut. Due to technological limitations, primary rolls have a maximum and minimum allowable width and each cut generate a loss in the roll width.

[1]This research was partially supported by FAPESP project 01/04412-4, MCT/CNPq under PRONEX program (Proc. 664107/97-4) and CNPq (Proc. 470608/01–3, 464114/00–4, 300301/98–7).

[2]Instituto de Computação — Universidade Estadual de Campinas, Caixa Postal 6176 — 13084–971 — Campinas–SP — Brazil, eduardo.xavier@ic.unicamp.br.

[3]Instituto de Computação — Universidade Estadual de Campinas, Caixa Postal 6176 — 13084–971 — Campinas–SP — Brazil, fkm@ic.unicamp.br.

In table 1, we present some common characteristics for final rolls. We consider three classes in this problem, one for each different thickness. The hardness interval of items with the same thickness are overlapped in a common interval to satisfy all hardness requirements. If there are items for which hardness cannot be assigned to the same thickness class, a new class must be defined for these ones.

Width (mm)	Hardness ($kg \cdot mm^{-2}$)	Thickness (mm)
50	50 to 70	4.50
74	60 to 75	4.50
93	32 to 39	3.50
35	32 to 41	3.50
20	20 to 30	2.50
100	24 to 35	2.50

Table 1: Characteristics of final rolls

In the example of table 1, we have a raw material roll of size K (1040 mm) which is first cut in primary rolls according to the different classes. In the example, the roll is first cut in three primary rolls. Each primary roll is processed by different operations to acquire the required thickness and hardness before obtaining the final rolls. Each cutting in the roll material generates a loss due to the width of the cutter knife. The cuts done at the first phase corresponds to the size of the wall division of our problem. The cuts of the second phase can be associated to the size of the items. This way, we only worry about the loss generated by the first phase cut. The figure 1 show the process.

Each processing operation has a high cost which implies items to be grouped before each processing operations, where each group corresponds to one compartment. In the above example we can consider three different classes and six different items. The size of the raw roll material corresponds to the size of the knapsack and the size of the wall division corresponds to the loss generated by the first cutting phase. The distances $d_{\min}$ and $d_{\max}$ are the minimum and maximum allowable width of the primary rolls.

Given a family of algorithms A_ϵ, for any $\epsilon > 0$, and an instance I for some problem P we denote by $A_\epsilon(I)$ the value of the solution returned by algorithm A_ϵ when executed on instance I and by $\mathrm{OPT}(I)$ the value of an optimal solution for this instance. We say that A_ϵ is a polynomial time approximation scheme (PTAS) for a maximization problem if for any $\epsilon > 0$ and any instance I we have $A_\epsilon(I) \geq (1-\epsilon)\mathrm{OPT}(I)$. If the algorithm is also polynomial in $1/\epsilon$ we say that A_ϵ is a fully polynomial time approximation scheme (FPTAS).

Results : In this paper we present a FPTAS and a PTAS for two restricted versions of the G-KNAPSACK. We also present a proof that a less restricted version of the G-KNAPSACK problem cannot be approximated unless $P = NP$. The algorithms we present are developed in two phases. In the first phase we have to solve a small problem for each class. In the second phase we use the results of each class to solve the fully problem using a dynamic programming approach.

For the FPTAS we consider at most a constant k of different item sizes for each class. The algorithm is simple and uses an algorithm to find an optimal bin packing of items. The bins have size $d_{\max}$ and must be filled by at least $d_{\min}$.

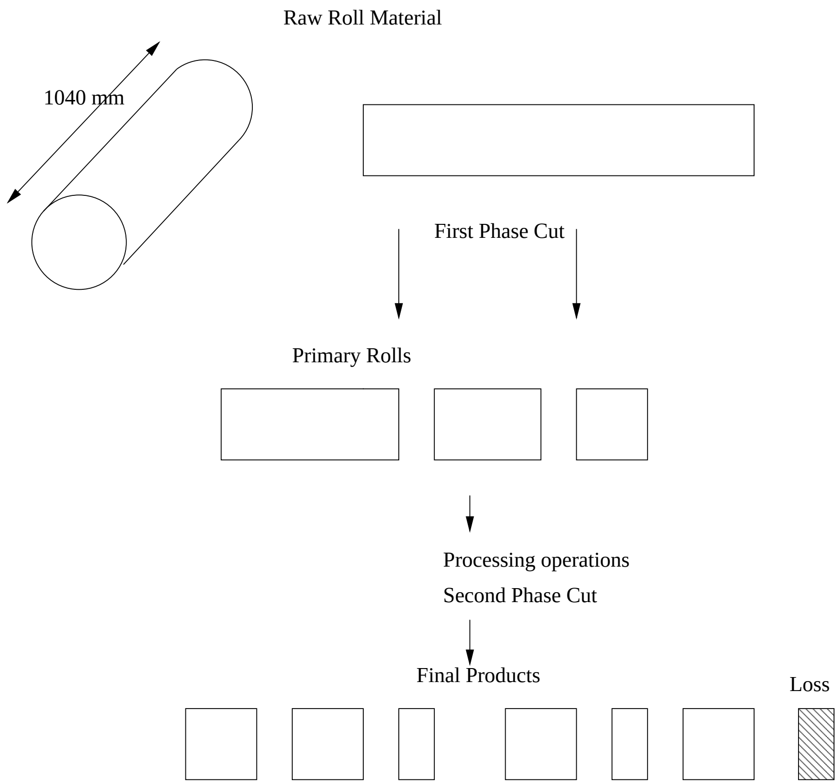

Figure 1: The two-phase cutting stock problem.

The PTAS is for the case when $d_{\min} = 0$. The algorithm is more complex and uses some nice ideas proposed by Chekuri and Khanna [2]. In this case, the algorithm find a set of items that can be packed into bins of size $d_{\max}$ and have a corresponding value very close to the optimal. The items are packed using known algorithms for bin packing.

Related Work : The knapsack problem is a well studied problem in the literature and a FPTAS was shown by Ibarra and Kim [5]. In [1], Caprara et al. present approximation schemes for two restricted versions of the knapsack problem, κKP and E-κKP. The κKP is the knapsack problem where the number of items chosen in the solution is at most k and the E-κKP has the number of items chosen in the solution exactly k. Chekuri and Khanna [2] present a PTAS for the Multiple Knapsack Problem (MKP). In this problem we have a set B of bins, each bin $j \in B$ with capacity b_j, and a set S of items, each item i with size s_i and profit p_i. The objective is to find a subset $U \subseteq S$ of maximum profit such that U has a feasible packing in B. In [6], Schachnai and Tamir present a PTAS to a class-constrained multiple knapsack problem (CCMK) and class-constrained bin-packing problem (CCBP). In both problems we have N bins, each bin j with C_j compartments and size V_j. We also have a set I of items, each item $u \in I$ have class c_u, size s_u and profit p_u. In problem CCMK, we have to find a packing of items into the bins B, such that the value of the items packed is maximized and each bin j has at most C_j items of different classes. In problem CCBP the bins have size 1 and C compartments. The goal is to find a legal placement of all items in a minimal number of bins. If we consider that we do not have the restrictions *(ii)* and *(iv)* of

G-KNAPSACK and the size of the wall is 0 then we can solve G-KNAPSACK using the algorithm to CCMK developed by Schachnai and Tamir.

Organization : In section 2, we present a generic algorithm to solve the G-KNAPSACK problem. This generic algorithm depends on the solution of another problem that we call SMALL. In sections 3 and 4 we present algorithms to solve problem SMALL for the FPTAS and the PTAS. Finally, in section 5 we present an inaproximality result to G-KNAPSACK problem.

2 Generic Algorithm

In this section we present some notation and formally define the problem G-KNAPSACK. Furthermore, we present a general approximation scheme considering the existence of an algorithm to a restricted problem.

For most approximation schemes, it is sufficient to prove that the solution generated is $O(\epsilon)$ from the optimum solution, since we can obtain a solution that is ϵ from the optimum solution simple rescaling the parameter ϵ. Therefore, the following claim is valid.

Claim 2.1 *Given a constant $\epsilon > 0$, if A_ϵ is a polynomial time algorithm for a maximization problem P, such that for any instance I of P, we have $A_\epsilon(I) \geq (1 - O(\epsilon))\mathrm{OPT}(I)$, then there is a polynomial time algorithm B_ϵ such that $B_\epsilon(I) \geq (1 - \epsilon)\mathrm{OPT}(I)$.*

An *instance* I for the G-KNAPSACK is a tuple $(S, c, s, v, K, d, d_{\max}, d_{\min})$ where S is a set of items, c, s and v are class, size and value functions over S, respectively; d is the size of wall divisions, and $d_{\max}$ and $d_{\min}$ are bounds for the compartments size. All values are non-negatives. We denote by n and m, the number of items in the set S and the number of classes, respectively.

The *size of a solution* $Q = (S', P')$, where $P' = (P'_1, \ldots, P'_k)$, is equal to $s(Q) := \sum_{i=1}^{k}(s(P'_i)+d)$ and the *value* of the solution is equal to $v(Q) := \sum_{i=1}^{k} v(P'_i)$.

A *solution* Q of an instance I for problem G-KNAPSACK is a pair (S', P') where $S' \subseteq S$ and P' is a partition of S', $P'_1, \ldots, P'_k$, each set P'_i has only items of one class and are such that $s(P') \leq K$ and $d_{\min} \leq s(P'_i) \leq d_{\max}$.

The problem G-KNAPSACK can be defined as follows: Given an instance I, find a solution Q of maximum value.

The crucial point for the algorithm of this section is a subroutine for a problem we call SMALL. The algorithm is the same for the two problems we consider in the next two sections, differing only on the way problem SMALL is solved.

PROBLEM SMALL: Given an instance I for G-KNAPSACK, where all items are of the same class and given a value w, find a solution Q of I with value w and smallest size.

We say that an algorithm SS_ϵ is *ϵ-relaxed* for problem SMALL if given an instance I and value w the algorithm generates a solution Q with $(1 - \epsilon)w \leq v(Q) \leq w$ and $s(Q) \leq s(O)$, where O is a solution with value w and smallest size. Such solution Q is called an *ϵ-relaxed* solution.

In figure 2 we present an approximation scheme for G-KNAPSACK using a subroutine to solve problem SMALL. The algorithm uses a rounding idea presented by Ibarra and Kim [5] for the knapsack problem.

In steps 1–3 the original instance is reparameterized in such a way the item values are non-negative integer values bounded by $\lceil n/\epsilon \rceil$. Therefore, the value of any solution is bounded by

ALGORITHM $G_\epsilon(I)$ where $I = (S, c, s, v, K, d, d_{\max}, d_{\min})$
Subroutine: SS_ϵ (ϵ-relaxed algorithm for problem SMALL).

1. *% reparameterize the instance by value*
2. let $P \leftarrow \max\{v_e : e \in S\},\ L \leftarrow \epsilon P/n$ and $V \leftarrow \lceil n^2/\epsilon \rceil$, where $n = |S|$
3. for each item $e \in S$ do $v'_e \leftarrow \lfloor \frac{v_e}{L} \rfloor$
4. *% generate an ϵ-relaxed solution for each class*
5. for class $j \leftarrow 1$ to m do
6.   for value $w \leftarrow 1$ to V do
7.   let S_j be the set of items in S with class j
8. $A_{j,w} \leftarrow SS_\epsilon(S_j, s, v', K, d, d_{\max}, d_{\min}, w)$
9. *% for each possible value w, find solution for* G-KNAPSACK *with classes* $\{1, \ldots, j\}$
10. for class 1 do
11.   for value $w \leftarrow 1$ to V do
12. $T_{1,w} \leftarrow A_{1,w}$
13. for class $j \leftarrow 2$ to m do
14.   for value $w \leftarrow 1$ to V do
15. $T_{j,w} \leftarrow$ (solution in $\{T_{j-1,w}, A_{j,w}, \min_{1 \le k < w}\{T_{j-1,k} + A_{j,w-k}\}\}$
of value in $[(1-\epsilon)w, w]$ and minimum size)
16. let Q be the solution $T_{m,w}$, $1 \le w \le V$ with maximum value w and size $s(Q) \le K$
17. return Q.

Figure 2: Generic algorithm for G-KNAPSACK using subroutine for problem SMALL

$V = O(n^2/\epsilon)$. This leads to a polynomial time algorithm using a dynamic programming approach with only $O(\epsilon)$ loss on the total value of the solution found by the algorithm.

In steps 4–8, the algorithm generates ϵ-relaxed solutions for each problem SMALL obtained from the reparameterized instances of each class and each possible value w. The solutions are stored in variables $A_{j,k}$, for each class j and each possible value k.

In steps 9–15 problem G-KNAPSACK is solved using dynamic programming. We have table $T_{j,w}$ indexed by classes j and all possible values w. It stores the smaller solution using items of classes $\{1, \ldots, j\}$ that have value w. The basic idea is to solve the following recurrence:

$$T_{j,w} := \min\{T_{j-1,w}, A_{j,w}, \min_{1 \le k < w}\{T_{j-1,k} + A_{j,w-k}\}\}$$

Finally, given that there are m classes, in steps 16–17 a solution generated with maximum value w is returned.

To prove that G_ϵ is an approximation scheme we consider that algorithm SS_ϵ, used as subroutine, is an ϵ-relaxed algorithm for problem SMALL.

Lemma 2.2 *If algorithm G_ϵ uses an ϵ-relaxed algorithm as subroutine and if $O_{j,w}$ is a solution using classes $\{1, \ldots, j\}$, with $w := v'(O_{j,w})$ and minimum size, then $T_{j,w}$ exists and $v'(T_{j,w}) \ge (1-\epsilon)w$ and $s(T_{j,w}) \le s(O_{j,w})$.*

Proof. First, note that if a solution $O_{j,w}$ with value $w := v'(O_{j,w})$ using items $\{1, \ldots, j\}$ exists, then a solution for $T_{j,w}$ also exists. We can prove this fact by induction on the number of classes.

The base case consider only items with class 1 and can be proved from the fact that subroutine SS_ϵ is an ϵ-relaxed algorithm, (that is, $T_{1,w} = A_{1,w}$).

Consider a solution $O_{j,w}$ with value $w := v'(O_{j,w})$ using items $\{1, \ldots, j\}$.

If $O_{j,w}$ uses only items of class j, then we have a solution $A_{j,w}$ which is obtained from subroutine SS_ϵ, which by assumption is an ϵ-relaxed algorithm. Therefore, $v'(A_{j,w}) \geq (1-\epsilon)v'(O_{j,w})$ and $s(A_{j,w}) \leq s(O_{j,w})$.

If $O_{j,w}$ uses only items of classes $1, \ldots, j-1$, by induction, $T_{j-1,w}$ exists and $v'(T_{j-1,w}) \geq (1-\epsilon)v'(O_{j,w})$ and $s(T_{j-1,w}) \leq s(O_{j,w})$.

If $O_{j,w} = (S, P)$ uses items of class j and items of other classes, denote by $O_1 = (S_1, P_1)$ and $O_2 = (S_2, P_2)$ two solutions obtained partitioning $O_{j,w}$ such that P_1 contains all parts of P with items of class different than j and P_2 contains all parts with items of class j. Let $k := v'(O_1)$. By induction, there are solutions $T_{j-1,k}$ and $A_{j,w-k}$ such that

$$v'(T_{j-1,k}) + v'(A_{j,w-k}) \geq (1-\epsilon)k + (1-\epsilon)(w-k) = (1-\epsilon)v'(O_{j,w}) \quad \text{and}$$

$$s(T_{j-1,k}) + s(A_{j,w-k}) \leq s(O_1) + s(O_2) = s(O_{j,w}).$$

□

Theorem 2.3 *If I is an instance for* G-KNAPSACK *and SS_ϵ is an ϵ-relaxed polynomial time algorithm for* SMALL *then the algorithm G_ϵ with subroutine SS_ϵ is a polynomial time approximation scheme for* G-KNAPSACK*. Moreover, if SS_ϵ is also polynomial time in $1/\epsilon$, the algorithm G_ϵ is a fully polynomial time approximation scheme.*

Proof. Let O be an optimum solution for instance I. Let w be the value of an optimal solution considering the rounding items.

$$\begin{aligned}
v(T_{m,w}) &= \sum_{e \in T_{m,w}} v_e \;\geq\; \sum_{e \in T_{m,w}} v'_e L \;=\; L v'(T_{m,w}) \\
&\geq L(1-\epsilon)w \;\geq\; L(1-\epsilon)\sum_{e \in O} v'_e \\
&\geq L(1-\epsilon)\sum_{e \in O}\left(\frac{v_e}{L} - 1\right) \;\geq\; L(1-\epsilon)\left(\sum_{e \in O}\frac{v_e}{L} - n\right) \\
&= (1-\epsilon)\left(\sum_{e \in O} v_e - nL\right) = (1-\epsilon)(\mathrm{OPT} - \epsilon P) \\
&\geq (1-\epsilon)(\mathrm{OPT} - \epsilon \mathrm{OPT}) \\
&\geq (1-2\epsilon)\mathrm{OPT}
\end{aligned}$$

Since the solution returned by the algorithm G_ϵ has value at least $v(T_{m,w})$, we have that $G_\epsilon(I) \geq (1-2\epsilon)\mathrm{OPT}$.

If m and n are the number of classes and the number of items, respectively, the time complexity of the algorithm G_ϵ is $O(mn^4/\epsilon^2 + mn^2/\epsilon \cdot T_{SS})$, where T_{SS} is the time complexity of subroutine SS_ϵ. Therefore, if SS_ϵ is polynomial time in n (and in $1/\epsilon$) then algorithm G_ϵ is a (fully) polynomial time approximation scheme. □

In the next two sections, we present two approximations schemes for restricted versions of problem G-KNAPSACK. The approximations schemes are based on algorithm G_ϵ and differ only in the way problem SMALL is solved. From now on we consider the items after the rounding process.

3 The FPTAS

In this section, we consider the G-KNAPSACK problem restricted to k different item sizes in each class. We present a fully polynomial time approximation scheme. The algorithm is the same presented in the previous section. In this case, we only need to present an ϵ-relaxed algorithm, used as subroutine by the algorithm G_ϵ, that is polynomial time both in the input size and in $1/\epsilon$. In fact, we show that an algorithm for problem SMALL does not need to compute solutions for every value w to obtain a fully polynomial time approximation scheme for problem G-KNAPSACK. Given such a subroutine, the approximation result follows from theorem 2.3.

The next result state the difficulty of the case we are considering.

Theorem 3.1 *The problem with the restriction that we have at most a constant k of different sizes in each class is still NP-hard.*

Proof. The theorem is valid since the knapsack problem is a particular case when each item is of a different class and $d_{\max} = \infty$, $d_{\min} = 0$ and $d = 0$. □

3.1 The k-PACK problem

Before presenting the algorithm to solve problem SMALL, consider the problem, denoted by k-PACK, which consists in packing n one-dimensional items with at most k different item sizes into the minimum number of bins of size $d_{\max}$, each bin filled by at least $d_{\min}$.

The algorithm to solve problem k-PACK uses a dynamic programming strategy combined with the generation of all configurations of packing items into one bin. In the figure 3 we present the algorithm that generates a function B that returns the minimum number of bins to pack an input list, under the restrictions of the problem k-PACK. For our purposes, we also need that the function B returns the partition of the input list into bins. For simplicity, we let to the interested reader its conversion to an algorithm that also returns the partition of the input list into bins.

In steps 1–4, the algorithm generates all possible subset of items that can be packed in one bin. In each iteration of the while command, steps 5–11, the algorithm uses the knowledge of instances that uses i bins to compute instances that uses $i + 1$ bins.

The following theorem is straightforward.

Theorem 3.2 *The algorithm P_k generates a function that returns the minimum number of bins to pack any sublist of the input list L of problem k-PACK. Moreover, the algorithm P_k has a polynomial time complexity.*

3.2 Solving problem SMALL

The following lemma states the relationship of a solution for problem SMALL and the problem k-PACK.

Lemma 3.3 *If $O = (L, P)$ is an optimum solution of an instance $I = (S, s, v', K, d, d_{\max}, d_{\min}, w)$ for the problem SMALL, $L \subseteq S$ and $P = (P_1, \ldots, P_k)$ then $k \geq B(L)$, where $B(L)$ is the minimum number of bins to pack L into bins of size $d_{\max}$, filled by at least $d_{\min}$.*

ALGORITHM $\mathrm{P}_k(L, d_{\min}, d_{\max})$

1. let $s_1, \ldots, s_k$ the k different sizes occurring in list L,
2. let d_i be the number of items in L of size s_i, $i = 1, \ldots, k$,
3. let Q_1 be the set of all tuples $(q_1, \ldots, q_k)$ such that $0 \leq q_i \leq d_i$, $i = 1, \ldots, k$
4. and $d_{\min} \leq \sum_{i=1}^{k} q_i s_i \leq d_{\max}$.
5. let $i \leftarrow 1$
6. while $(d_1, \ldots, d_k) \notin Q_i$ do
7. $Q_{i+1} \leftarrow \emptyset$
8. for each $q' \in Q_1$ and $q'' \in Q_i$ do
9. $q \leftarrow q' + q''$
10. if $q \notin Q_i$ then $Q_{i+1} \leftarrow Q_{i+1} + q$
11. $i \leftarrow i + 1$
12. let $B(q) \leftarrow j$ for all $q \in Q_j$, $1 \leq j \leq i$
13. return B

Figure 3: Algorithm to find the minimum number of bins to pack any subset of L

Proof. Note that items packed in the optimum solution O are separated by wall divisions of size d and two subsequent wall divisions defining a compartment must be a distance between $d_{\min}$ and $d_{\max}$. Items in compartments can be considered as a packing into bins of size $d_{\max}$ occupied by at least $d_{\min}$. Since $B(L)$ is the minimum number of such bins to pack L, the number of compartments of L is at least $B(L)$. ∎

Corollary 3.4 *If $O = (L, P)$ is an optimum solution of an instance $I = (S, s, v', K, d, d_{\max}, d_{\min}, w)$ for problem* SMALL, *then $s(O) \geq s(L) + B(L)d$.*

In figure 4, we present an algorithm for solving a relaxed version of problem SMALL, which is sufficient to our purposes. The algorithm first consider all possible configurations of solutions for problem SMALL, without considering the value of each item. This step is performed by a subroutine to solve problem k-PACK. Instead of finding each possible attribution of values for each configuration, the algorithm only generates valid configurations with maximum value. For a given value w, the algorithm only returns a solution if the value is a maximum value for some configuration. Notice that we return the smallest packing that have the given value.

Theorem 3.5 *If I is an instance for* G-KNAPSACK *with at most k different item sizes and algorithm G_ϵ is executed with subroutine k-SS then $G_\epsilon(I) \geq (1 - \epsilon)\mathrm{OPT}(I)$.*

Proof. Consider an optimum solution O for the instance I with the rounded items . Let Q_c be the set of items of class c used in this optimal solution. This set of items corresponds to a configuration q_c that is packed optimally by corollary 3.4. In algorithm k-SS we return items of maximum value corresponding to this configuration. If k-SS does not return this optimal solution to G_ϵ, is because it finds another configuration with the same value but with smaller size in line 10 of the algorithm. It follows from theorem 2.3 that the optimal solution found by algorithm G_ϵ with k-SS is a FPTAS to G-KNAPSACK. ∎

ALGORITHM k-$SS(S, s, v', K, d, d_{\max}, d_{\min}, w)$
Subroutine: P_k (Subroutine to solve problem k-PACK).

let $B \leftarrow P_k(S, d_{\min}, d_{\max})$
let $s_1, \ldots, s_k$ the k different sizes occurring in list L,
let d_i be the number of items in L of size s_i, $i = 1, \ldots, k$,
let Q be the set of all tuples $(q_1, \ldots, q_k)$ such that $0 \leq q_i \leq d_i$, $i = 1, \ldots, k$
for each $q = (q_1, \ldots, q_k) \in Q$ do
    let $P(q)$ the packing obtained using function $B(q)$ placing for each size s_j,
        i_j items of S with size s_j and greatest values
let $Q_w \leftarrow \{q \in Q : w = v(P(q))\}$
if $Q_w \neq \emptyset$ then
    return $q \in Q_w$ such that $s(q)$ is minimum.
else
    return $\emptyset$

Figure 4: Algorithm to find the minimum number of bins to pack any subset of L

4 The PTAS

In this section, we present an algorithm to solve the problem SMALL with the restriction that $d_{\min} = 0$. This restriction is based in our weakness to find packings with a minimum filling. In section 5 we present an inaproximality result that show that the full version of G-KNAPSACK problem can not be approximated unless $P = NP$.

The algorithm of this section uses some nice ideas proposed by Chekuri and Khanna [2]. Given an instance $I = (S, s, v', K, d, d_{\max}, d_{\min}, w)$ for the problem SMALL, the algorithm performs two basic steps. First, the algorithm finds a set $U \subseteq S$ with $v'(U) \geq (1 - O(\epsilon))w$ such that its packing has size no bigger than an optimal packing of value w. This is shown in the next subsection. In the second step, the algorithm packs a set $U' \subseteq U$ such that $v(U') \geq (1 - O(\epsilon))v'(U)$.

4.1 Finding the Items

Given an instance I we first have to find a subset of the items with total value very close to w. The algorithm, denoted by $Find$, is presented in figure 5. It finds a polynomial number of sets, such that at least one has value close to w and its packing size is at most the size of the packing of the optimal solution.

In the first step, the algorithm $Find$ round down each item value to the nearest power of $(1+\epsilon)$. From now on, consider the items with the new values. Given a set O, with $v'(O) = w$ and smallest size, we have with the rounding that $\frac{w}{(1+\epsilon)} \leq v''(O) \leq w$. With the rounding specified we have $h = \lfloor \log \frac{n}{\epsilon} \rfloor$ different values of items. The items are grouped by their values in sets $S_1, \ldots, S_h$, such that items in the same set have the same value.

Instead of looking for the sets $O_i = O \cap S_i, i = 1, \ldots, h$, the algorithm $Find$, finds subsets U_i with values very close to $v'(O_i)$ whose packing is not larger than an optimal packing of O.

For each index i, the algorithm finds an integer value $k_i \in \{0, \ldots, \frac{h}{\epsilon}\}$ such that $k_i(\frac{\epsilon w}{h}) \leq$

ALGORITHM $Find(S, \epsilon, w)$

Subroutine: $Round$ (Subroutine to round the values of the items).

1. for each $e \in S$ do
2. $\quad v''_e \leftarrow (1+\epsilon)^k$ where $(1+\epsilon)^k \leq v'_e < (1+\epsilon)^{k+1}$
3. let $h \leftarrow \lfloor \log \frac{n}{\epsilon} \rfloor$
4. for each $i \in \{0, \ldots, h\}$ do
5. $\quad$ let S_i be the set of items with value $(1+\epsilon)^i$ in S
6. $\quad$ let $(e^i_1, \ldots, e^i_{n_i})$ the items in S_i sorted in non-decreasing order of size
7. let T be the set of all possible tuples $(k_1, \ldots, k_h)$
such that $k_i \in \{0, \ldots, \frac{h}{\epsilon}\}, 0 \leq i \leq h$ and $\sum_{i=0}^{h} k_i = \frac{h}{\epsilon}$.
8. for each value w' in the interval $[(1-\epsilon)w, w]$ do
9. $\quad$ for each tuple $(k_1, \ldots, k_h)$ in T do
10. $\quad\quad$ for each $i \in \{0, \ldots, h\}$ do
11. $\quad\quad\quad$ let $U_i = \{e^i_1, \ldots, e^i_j\}$ such that $v(\{e^i_1, \ldots, e^i_{j-1}\}) < k_i(\frac{\epsilon w'}{h}) \leq v(\{e^i_1, \ldots, e^i_j\})$
12. $\quad\quad$ $U \leftarrow (U_1 \cup \ldots \cup U_h)$
13. $\quad\quad$ $Q \leftarrow Q + U$.
14. return Q.

Figure 5: Algorithm to find sets with value very close to a given value w

$v''(O_i) < (k_i+1)(\frac{\epsilon w}{h})$. The algorithm generate all possible tuples $(k_1, \ldots, k_h)$ and one will satisfy this condition. First we show that such values k_i exist in such a way that they are very close to the optimal.

Lemma 4.1 *If $O \subseteq S$ is a set with $v'(O) = w$ and smallest size, then there exists a valid tuple $(k_1, \ldots, k_h)$ such that for each i we have $k_i \in \{0, \ldots, \frac{h}{\epsilon}\}$ and $\sum_{i=1}^{k} k_i(\frac{\epsilon w}{h}) \geq (1-\epsilon)w$.*

Proof. Let $O = O_1 \cup \ldots \cup O_h$ where $O_i = S_i \cap O, 1 \leq i \leq h$. For each i, let $k_i = \lfloor \frac{v''(O_i)h}{\epsilon w} \rfloor$ and we have:

$$k_i(\frac{\epsilon w}{h}) = \lfloor \frac{v''(O_i)h}{\epsilon w} \rfloor (\frac{\epsilon w}{h}) \geq (\frac{v''(O_i)h}{\epsilon w} - 1)\frac{\epsilon w}{h} = v''(O_i) - \frac{\epsilon w}{h}.$$

Thus, for each i we loose at most $\frac{\epsilon w}{h}$, so the total loss is ϵw.

□

The first idea is to test all possible values k_i, but this does not leads to a polynomial time algorithm. In this case we have $O((\log n)^{\log n})$ possibilities to test. To limit the number of tests polynomially, we use the fact that $\sum_i k_i \leq \frac{h}{\epsilon}$. Note that if this condition is not satisfied we would obtain sets with values bigger than w. The next two lemmas, presented by Chekuri and Khanna [2], limit the number of different tuples to be considered.

Lemma 4.2 *Let f be the number of g-tuples of non negative integer values such that the sum of the values of the tuple is d. Then*

$$f = \binom{d+g-1}{g-1}$$

Lemma 4.3 *The number of different tuples of index k_i that we have to test is $O(n^{\frac{1}{\epsilon}})$.*

Proof.
Let $d = \frac{h}{\epsilon}$ and $g = h$. From lemma 4.2 we know that the number of possibilities to test is $f = \binom{d+g-1}{g-1}$.
That is,

$$f = \frac{(d+g-1)!}{(g-1)!d!} = \frac{(d+g-1)\dots(d+1)}{(g-1)!} \leq \frac{(d+g-1)^{g-1}}{(g-1)!}.$$

Using the Stirling approximation for the factorial we have $(g-1)! \geq (\frac{(g-1)^{g-1}}{e})$. Therefore,

$$f \leq (\frac{e(d+g-1)}{g-1})^{g-1}.$$

Since $h + \frac{h}{\epsilon} \leq \alpha(\frac{h}{\epsilon} - 1)$ for some constant α, we have the new limit:

$$f \leq (\frac{e\alpha(g-1)}{g-1})^{g-1} = (e\alpha)^{g-1} = O(n^{\frac{1}{\epsilon}}).$$

□

The enumeration of the tuples is done in step 7 of the algorithm and is used in the loop of step 9. Notice that we have a polynomial number of tuples. Now we show how the algorithm get the items. Given a set S_i and a value k_i the algorithm take items as follows:

1. In step 6 the algorithm sort the set S_i in non-decreasing order of items size.

2. Items are taken in non-decreasing order of size until the total value of the items becomes bigger or equal than $k_i(\frac{\epsilon w}{h})$ in step 11.

The next lemma state that at least one of the sets obtained this way have value very close to the optimal and that its packing size is not bigger than the packing size of the optimal set O.

Lemma 4.4 *Let O be a solution with value w and smallest size and $O_1, \dots, O_h$ the partition of O such that $O_i = O \cap S_i, i = 1, \dots, h$. There are values $k_1, \dots, k_h$ and sets $U_1, \dots, U_h$ obtained by the algorithm $Find$, such that $v(U_1 \cup \dots \cup U_h) \geq v(O)(1 - O(\epsilon))$ and that the packing size of the set U is not larger than the packing size of O.*

Proof. Consider an integer $i, 0 \leq i \leq h$. From lemma 4.1 there exists an integer k_i such that $k_i(\frac{\epsilon w}{h}) \leq v''(O_i) < (k_i + 1)(\frac{\epsilon w}{h})$. Since the algorithm $Find$ tests all possibilities of k_i, one tuple satisfy the above condition. In step 11 the algorithm take items $e_i \in S_i$ that can fall in one of this two cases:

1) Suppose that $v''_{e_i} \leq \frac{\epsilon w}{h}$. The algorithm take items until their total value become greater or equal than $k_i(\frac{\epsilon w}{h})$. Taking items this way the loose in value is not more than $\frac{\epsilon w}{h}$ because $v''(O_i) < (k_i + 1)(\frac{\epsilon w}{h})$. If all the sets O_i fall in this case the total loss is at most ϵw.

2) Suppose that $v''_{e_i} > \frac{\epsilon w}{h}$. The algorithm take items until their total value become greater or equal than $k_i(\frac{\epsilon w}{h})$. We know that $v''_{e_i} > \frac{\epsilon w}{h}$, and in this case there is no loss because the algorithm take exactly the value of $v''(O_i)$.

Note that all items in the set O_i have the same value and so $v''(O_i)$ is multiple of the value of the items. The algorithm never take more items than the set O_i, i.e, $|U_i| \leq |O_i|$. The algorithm takes the items in non-decreasing order of size obtaining a packing with size not larger than the packing of the items in O_i. □

Remind that O corresponds to a set such that $v'(O) = w$ before the rounding in step 1 of algorithm $Find$ and that its packing size is minimum. After the rounding we have that $\frac{w}{(1+\epsilon)} \leq v''(O) \leq w$ wich implies that $v''(O) \in [(1-\epsilon)w, w]$ since

$$w - \frac{w}{(1+\epsilon)} = \frac{\epsilon w}{1+\epsilon} \leq \epsilon w.$$

If we test all integer possible values w' in the interval $[(1-\epsilon)w, w]$ we have at most ϵw possibilities. Since the maximum value of w is $O(\frac{n^2}{\epsilon})$ we have at most $O(n^2)$ possible tests. If we take $w' = \lceil v''(O) \rceil$ this value satisfy:

$$k_i \frac{\epsilon w'}{h} \leq v''(O_i) < (k_i + 1)\frac{\epsilon w'}{h}, \quad 0 \leq i \leq h.$$

According to lemma 4.4 we have a loss of at most $\epsilon w'$ which corresponds to a small fraction of the value of O.

$$\epsilon w' \leq \epsilon(v''(O) + 1) \leq 2\epsilon v''(O)$$

The search for the value of w' corresponds to the loop in step 8 of the algorithm.

At last, the algorithm generate a polynomial number of sets, at least one with value $v = (1 - O(\epsilon))w$ and that can be packed optimally. In the next section we see how these sets can be packed.

4.2 Packing the Items

In the previous section we have obtained at least one set U of items such that its total value is very close to the optimal O and that its packing size is not large than the packing size of O. The packing is obtained using known algorithms for bin packing. Notice that an optimal bin packing using bins of size $d_{\max}$ is a limit for the optimal packing in compartments. This is because we choose items of smaller size and possibly less items than the optimal set. Also notice that packing the items with this algorithm we can respect only the maximum limit of a compartment, $d_{\max}$.

Fernadez de la Vega and Lueker [7] have shown how to build an algorithm that pack items using at most $(1+\epsilon)OPT + 1$ bins. Frieze and Clarke [4] found a PTAS for the problem of packing a set of items in m bins maximizing the total value of the packed items, where m is a constant.

We call by A_{FVL} the algorithm of Fernadez de la Vega and Lueker, and by A_{FC} the algorithm of Frieze and Clarke. The algorithm $Pack$, to pack the set U, is presented in figure 6. It first uses the algorithm A_{FVL} to pack the items in U. If the number of bins used is smaller than $\frac{1}{\epsilon} + 2$ it just forget the packing and pack the set using the algorithm A_{FC}. In the other case we just take the OPT' bins with biggest value. We denote by $size(N)$ the number of bins used in packing N.

ALGORITHM $Pack(U)$

Subroutine: A_{FVL} and A_{FC} (Subroutines to pack the items).

1. let $N \leftarrow A_{FVL}(U)$
2. let $OPT' \leftarrow \frac{size(N)-1}{1+\epsilon}$
3. if $OPT' \geq \frac{1}{\epsilon}$ then
4. let P be the OPT' bins with biggest values in N
5. else
6. $P_{FC} \leftarrow \emptyset$
7. for $j \leftarrow 1$ to $\frac{1}{\epsilon}+2$ do
8. $P_{FC} \leftarrow P_{FC} + A_{FC}(U, j)$
9. Let P be the smaller packing in P_{FC} such that $v(P) \geq (1-\epsilon)v(U)$
10. return P

Figure 6: Algorithm to pack the items

Lemma 4.5 *If P is the solution corresponding to the packing generated by the algorithm $Pack$ over the set U then $v(P) \geq (1 - O(\epsilon))v(U)$ and its size is smaller than the optimal packing.*

Proof.

If $OPT' \geq \frac{1}{\epsilon}$ then we loose the value of the $\epsilon OPT' + 1$ bins. Of course $OPT' \leq OPT$, where OPT is the minimum number of bins to pack the items in U. Each one of the bins have value at most $\frac{v(U)}{size(N)}$. Then we loose at most

$$\frac{v(U)}{size(N)}(\epsilon OPT' + 1) \leq \frac{v(U)}{OPT}(\epsilon OPT + 1) = \epsilon v(U) + \frac{v(U)}{OPT}.$$

Since $OPT \geq OPT' > \frac{1}{\epsilon}$ an upper bound for the loose is $2\epsilon v(U)$.

Therefore, $v(P) \geq (1 - 2\epsilon)v(U)$.

If $OPT' < \frac{1}{\epsilon}$ then we have that $size(N) \leq \frac{1}{\epsilon} + 2$. In this case P is obtained running the algorithm A_{FC} with j bins $1 \leq j \leq \frac{1}{\epsilon} + 2$. We take the smaller packing of the items such that the total loose is at most $\epsilon v(U)$. □

4.3 The ϵ-relaxed algorithm

In this section we present the ϵ-relaxed algorithm for the problem SMALL with the restriction that $d_{\min} = 0$. The algorithm is presented in figure 7. It is very simple and just use the algorithms presented in sections 4.1 and 4.2.

Given a value w, the algorithm $Find$ generates a polynomial number of sets U as shown in section 4.1. At least one of the sets have value very close to the value of the optimal set O and have a packing of size at most the size of the optimal. For all possibilities of U we pack it with algorithm $Pack$. The algorithm $Find$ generate a loss of at most 5ϵ. The algorithm $Pack$ generate a loss of at most 2ϵ. The solution returned by the algorithm is the packing of smaller size such that the loose is at most $(2 + 5)\epsilon w$. We can conclude with the theorem:

ALGORITHM $Small(S, s, v, K, d, d_{\max}, 0, w)$
Subroutine: $Find$ and $Pack$ (Subroutines of the previous sections).

let $Q \leftarrow Find(S, \epsilon, w)$
$MIN \leftarrow \infty$
for each $U \in Q$ do
    $M \leftarrow Pack(U)$
    if $size(M) < MIN$ and $v(M) > (1 - (5+2)\epsilon)w$ then
        $M2 \leftarrow M$
        $MIN \leftarrow size(M)$
return $M2$

Figure 7: Algorithm to solve Small Problem

Theorem 4.6 *Algorithm G_ϵ is a PTAS to G-Knapsack if $d_{\min} = 0$, when executed with the ϵ-relaxed algorithm $Small$ of this section.*

5 Inaproximality of the G-KNAPSACK problem

In this section we present an inaproximality result for the G-KNAPSACK problem. We prove that the full version of the problem where $0 < d_{\min} \leq d_{\max}$ can not be approximated to any factor unless $P = NP$. The proof is made reducing the partition problem, with instance I_1 and items with total size K, to an instance of the G-KNAPSACK problem with the same set of items and $d_{\min} = d_{\max} = \frac{K}{2}$. It is not hard to see that with this instance, G-KNAPSACK problem have only two possible solutions, one with size 0 and other with size $\frac{K}{2}$. The last solution is obtained if and only if instance I_1 has a solution.

We can also prove that the G-KNAPSACK can not be approximated when $0 < d_{\min} < d_{\max}$. The next theorem states this result.

Theorem 5.1 *There is a gap-introducing reduction transforming instance I_1 of the Partition Problem (PP) to an instance I_2 of the* G-KNAPSACK *problem such that:*

- *If instance I_1 is satisfiable then $OPT(I_2) = d_{\min}$, and*
- *if I_1 is not satisfiable, $OPT(I_2) = 0$.*

Proof. Let $I_1 = (S, s)$ be the instance of Partition Problem where each item $e \in S$ has size s_e. We construct an instance I_2 such that $OPT(I_2) \in \{0, d_{\min}\}$. Let $I_2 = (S, c, s', v, K, 0, d_{\max}, d_{\min})$ be the instance of G-KNAPSACK problem. For each item $e \in S$ we have that $v_e = s_e \cdot \alpha$ and $s'_e = s_e \cdot \alpha$, where α is a constant integer. Of course the partition problem is satisfiable with instance (S, s) if and only if it is satisfiable with instance (S, s'). All items in S belongs to the same class. Let $d_{\min} = \frac{\sum_{e \in S} s'_e}{2}$ and $d_{\max} = d_{\min} + (\alpha - 1)$. Let $K = d_{\max}$. Notice that there is no wall divisions.

First note that any size s'_e is multiple of α and therefore, any solution of I_2 is also multiple of α. Since $d_{\min}$ is multiple of α, we have

$$d_{\min} < d_{\max} < d_{\min} + \alpha$$

and we conclude that there is no solution to instance I_2 with size greater than $d_{\min}$.

If instance I_1 is satisfiable then the optimal solution of instance I_2 has value $d_{\min}$ and the knapsack is filled until $d_{\min}$. If instance I_2 is not satisfiable then the only solution with size multiple of α that respects the limits $d_{\min}$ and $d_{\max}$ has value 0 and it packs no items.

□

Corollary 5.2 *There is no r-approximation algorithm for* G-KNAPSACK *problem when* $0 < d_{\min} \leq d_{\max}$, $r > 0$*, unless* $P = NP$.

6 Conclusion

We present approximation algorithms to some restricted versions of G-knapsack. We also prove that the full version of G-KNAPSACK problem cannot be approximated. The problem has a practical motivation in the iron and steel industry where a problem of cutting rolls appears.

References

[1] A. Caprara, H. Kellerer, U. Pferschy and D. Pisinger. Approximation Algorithms for Knapsack Problems with Cardinality Constraints. *European Journal of Operational Research*, 123:333–345, 2000.

[2] C. Chekuri and S. Khanna. A ptas for the multiple knapsack problem. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 213–222, 2000.

[3] J.S. Ferreira, M.A. Neves, P. Fonseca, and P.F. Castro. A two-phase roll cutting problem. *European Journal of Operational Research*, 44:185–196, 1990.

[4] A. M. Frieze and M. R. B. Clarke. Approximation algorithms for the m-dimensional 0-1 knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 15:100–9, 1984.

[5] O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the Association for Computing Machinery*, 22:463–468, 1975.

[6] H. Shachnai and T. Tamir. Polynomial time approximation schemes for class-constrained packing problems. *Proc. of the 3rd International Workshop on Approximation Algorithms for Combinatorial Optimization*, (APPROX) 2000.

[7] W. Fernandez de la Vega and G.S. Lueker. Bin packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.