UKP5: Solving the Unbounded Knapsack Problem

Henrique Becker , Luciana S. Buriol

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{hbecker,buriol}@inf.ufrgs.br

Abstract. *In this extended abstract we present UKP5, an algorithm for solving the unbounded knapsack problem. UKP5 is based on dynamic programming, but implemented in a non traditional way: instead of looking backward for stored values of subproblems, it stores incremental lower bounds forward. UKP5 is considerably simpler than EDUK2, the state-of-the-art algorithm for solving the problem. We run UKP5 and EDUK2 on a benchmark of 4540 hard instances proposed by the authors of EDUK2. The results reveal that UKP5 outperforms EDUK2, being 47 times faster on the average.*

Resumo. *Nesse resumo extendido nós apresentamos o UKP5, um algoritmo para solucionar o* unbounded knapsack problem *(problema da mochila com repetições). O UKP5 é baseado em programação dinâmica, mas implementado de uma forma não-tradicional: ao invés de olhar para trás para usar soluções de subproblemas previamente computados, ele armazena limites inferiores a frente. O UKP5 é consideravelmente mais simples que o EDUK2, o algoritmo do estado da arte para solucionar o problema. Nós executamos o UKP5 e o EDUK2 em uma bateria de testes contendo 4540 instâncias consideradas difíceis pelos autores do EDUK2. Os resultados mostram que o UKP5 é, em média, 47 vezes mais rápido que o EDUK2.*

1. Introduction

The unbounded knapsack problem (UKP) is a simpler variation of the well-known bounded knapsack problem (BKP). UKP allows the allocation of an unbounded quantity of each item type. The UKP is NP-Hard, and thus has no known polynomial-time algorithm for solving it. However, it can be solved by a pseudo-polynomial dynamic programming algorithm.

Two techniques are often used for solving UKP: dynamic programming (DP) [Andonov et al. 2000], [Garfinkel and Nemhauser 1972, p. 214], and branch and bound (B&B) [Martello and Toth 1990]. The state-of-the-art solver for the UKP, introduced by [Poirriez et al. 2009], is a hybrid solver that combines DP and B&B. The solver's name is PYAsUKP, and it is an implementation of the EDUK2 algorithm.

An UKP instance is composed by a capacity c, and a list of n items. Each item can be referenced by its index in the item list $i \in \{1 \ldots n\}$. Each item i has a weight value w_i, and a profit value p_i. A solution is an item multiset, i.e, a set that allows multiple copies of the same element. The sum of the items weight, or profit, of a solution s is denoted by w_s, or p_s. A valid solution s has $w_s \leq c$. An optimal solution s^* is a valid solution with the greatest profit among all valid solutions. The UKP objective is to find an optimal solution for the given UKP instance. The mathematical formulation of UKP is:

$$maximize \sum_{i=1}^{n} p_i x_i \quad (1) \qquad subject\ to \sum_{i=1}^{n} w_i x_i \leq c \quad (2) \qquad x_i \in \mathbb{N}_0 \quad (3)$$

The quantities of each item i in an optimal solution are denoted by x_i, and are restricted to the non-negative integers, as (3) indicates. The efficiency of an item i is the ratio $\frac{p_i}{w_i}$. We use w_{min} and w_{max} to denote the smallest item weight, and the biggest item weight, respectively.

2. UKP5: The Proposed Algorithm

UKP5 is inspired by the DP algorithm described by Garfinkel and Nemhauser (we will reference it as G&N) [Garfinkel and Nemhauser 1972, p. 221]. The name "UKP5" is due to five improvements applied over that algorithm: **Symmetry pruning**: symmetric solutions are pruned in a more efficient fashion than in G&N; **Sparsity**: not every position of the optimal solutions value array has to be computed; **Dominated solutions pruning**: we never generate some solutions if they are worse than solutions already generated (bigger weight and smaller profit); **Time/memory tradeoff**: the test $w_i \leq y$ from G&N was removed in cost of more O(w_{max}) memory; **Periodicity**: the periodicity check suggested in [Garfinkel and Nemhauser 1972] (but not implemented there) was adapted and implemented.

The g is a sparse array where we store solutions profit. If $g[y] > 0$ then there exists a non-empty solution s with $w_s = y$ and $p_s = g[y]$. The d array stores the index of the last item used on a solution. If $g[y] > 0 \wedge d[y] = i$ then the solution s with $w_s = y$ and $p_s = g[y]$ has at least one copy of item i. Our first loop (lines 4 to 9) stores all solutions comprised of a single item in the arrays g and d. After this setup, we simply iterate g and, when we find a stored solution, we create new solutions combining the current solution with single items. We only prune symmetric or dominated solutions, all other valid solutions are generated. Consequently, one of those solutions is guaranteed to be an optimal solution, and *opt* will end with the optimal solution profit value.

With the intent of making easier to the reader to undestand the core ideia of the UKP5 algorithm, the pseudocode presented at Algorithm 1 was stripped of many small optimizations. Some of them are: all the items are sorted by non-increasing efficiency; the y^* periodicity bound is computed as in [Garfinkel and Nemhauser 1972, p. 223], and used to reduce the c value; an UKP5-specific periodicity check is used, we don't describe it here because of the page limit. The solution assemble phase also isn't described here, but it's similar to the one used by the DP method described in [Garfinkel and Nemhauser 1972, p. 221, Steps 6-8].

3. Computational Results, Analysis and Conclusions

The computer used on the experiments was an ASUS R552JK-CN159H (Intel Core i7-4700HQ Processor, 6M Cache, 3.40 GHz). The operating system used was Linux 4.3.3-2 x86_64. The number of instances of each different class are: Subset-Sum (400); Strong Correlation (240); Postponed Periodicity (800); No Collective Dominance (2000);

Algorithm 1 UKP5 – Computation of *opt*

```
 1: procedure UKP5($n, c, w, p, w_{min}, w_{max}$)
 2:     $g \leftarrow$ array of $c + w_{max}$ positions each one initialized with $0$
 3:     $d \leftarrow$ array of $c + w_{max}$ positions each one initialized with $n$
 4:     for $i \leftarrow 1, n$ do                                   ▷ Stores one-item solutions
 5:         if $g[w_i] < p_i$ then
 6:             $g[w_i] \leftarrow p_i$
 7:             $d[w_i] \leftarrow i$
 8:         end if
 9:     end for
10:     $opt \leftarrow 0$
11:     for $y \leftarrow w_{min}, c$ do               ▷ Can end early because of periodicity check
12:         if $g[y] \leq opt$ then        ▷ Handles sparsity and pruning of dominated solutions
13:             continue                          ▷ Ends current iteration and begins the next
14:         end if
15:         $opt \leftarrow g[y]$
16:         for $i = 1, d[y]$ do                  ▷ Creates new solutions (never symmetric)
17:             if $g[y + w_i] < g[y] + p_i$ then
18:                 $g[y + w_i] \leftarrow g[y] + p_i$
19:                 $d[y + w_i] \leftarrow i$
20:             end if
21:         end for
22:     end for
23:     return $opt$
24: end procedure
```

SAW (1100). These datasets aim to reproduce the ones described in [Poirriez et al. 2009] (see the paper for more information). The same tool was used to generate the datasets (PYAsUKP). The capacity of the instances vary between 10^6 and 10^8, and the number of items vary between 10^3 and 10^5. The sources can be found at `https://github.com/henriquebecker91/masters/tree/v0.1`[1].

As we can see in Figure 3, for many instances PYAsUKP is faster than UKP5. However, PYAsUKP can also be one or more orders of magnitude slower than UKP5 on some instances. Our tests have shown when PYAsUKP is faster than UKP5 this is often because of its B&B phase, that solves some instances almost instantly. When B&B fails to solve the instance in a short time, PYAsUKP fallbacks to a DP algorithm that seems to be many times slower than UKP5. It's important to note that both algorithms have a pseudo-polynomial worst-case ($O(c \times n)$). Integrating a B&B phase to UKP5 seems promising, and will be the theme for future works.

[1]The UKP5 implementation is at **codes/cpp/** and two versions of PYAsUKP are at **codes/ocaml/**. The *pyasukp_site.tgz* is the version used to generate the instances, and was also available at `http://download.gna.org/pyasukp/pyasukpsrc.html`. A more stable version was provided by the authors. This version is in *pyasukp_mail.tgz* and it was used to solve the instances the results presented in Figure 3. The *create_*_instances.sh* scripts inside **codes/sh/** were used to generate the instance datasets.

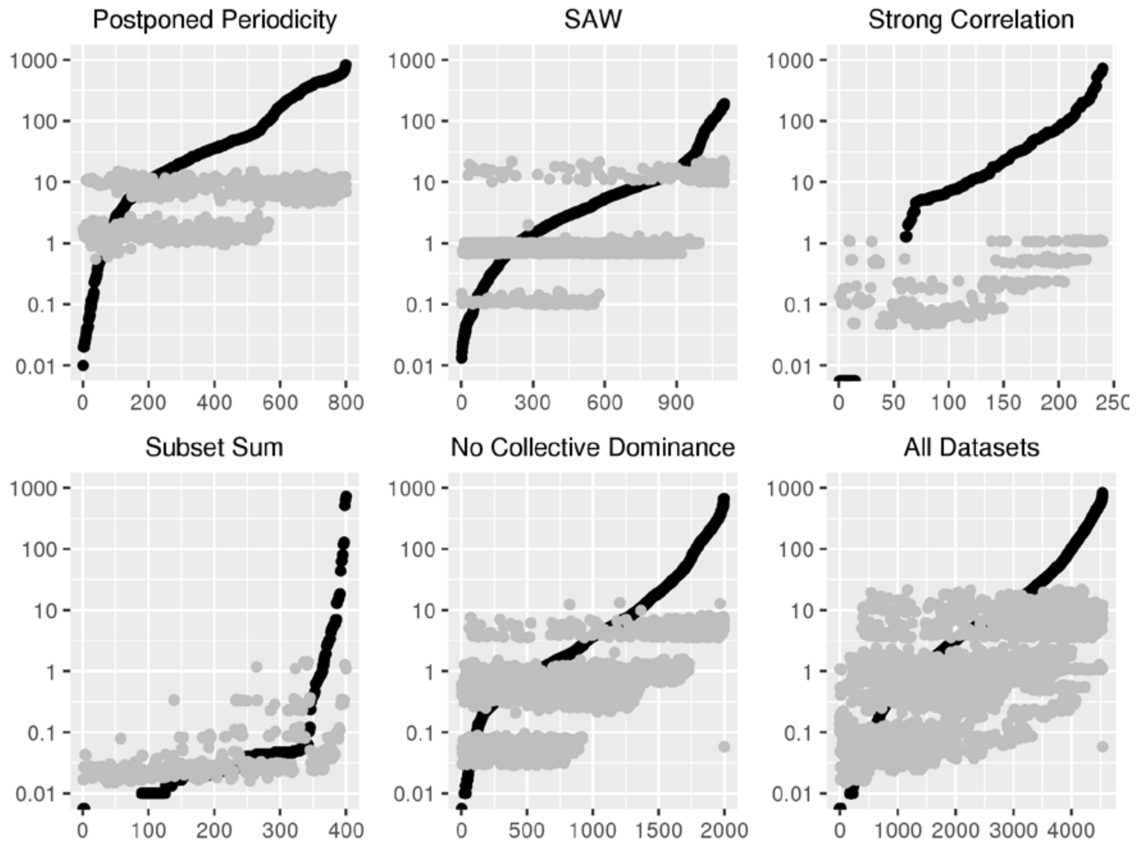

Figure 1. The times used by UKP5 and PYAsUKP for each instance of each class. The black dots represent PYAsUKP times. The gray dots represent UKP5 times. The y axis is the time used to solve an UKP instance, in seconds. The x axis is the instance index when the instances are are sorted by the time PYAsUKP took to solve it. Note that the y axis is in logarithmic scale.

Acknowledgments We are thankful to the CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) for the financial support.

References

Andonov, R., Poirriez, V., and Rajopadhye, S. (2000). Unbounded knapsack problem: Dynamic programming revisited. *European Journal of Operational Research*, 123(2):394–407.

Garfinkel, R. S. and Nemhauser, G. L. (1972). *Integer programming*, volume 4. Wiley New York.

Martello, S. and Toth, P. (1990). An exact algorithm for large unbounded knapsack problems. *Operations research letters*, 9(1):15–20.

Poirriez, V., Yanev, N., and Andonov, R. (2009). A hybrid algorithm for the unbounded knapsack problem. *Discrete Optimization*, 6(1):110–124.