A fast dynamic programming multi-objective knapsack problem

Marcos Daniel Valadão Baroni* Flávio Miguel Varejão

September 13, 2017

Abstract

This work addresses... The Multidi Objective knapsack programming. The dynamic programming method... The data structure...

1 Introduction

2 The Multiobjective Knapsack Problem

A general multiobjective optimization problem can be described as a vector function f that maps a decision variable (solution) to a tuple of m objectives. Formally:

$$\begin{aligned} &\min/\max\ \boldsymbol{y} = f(\boldsymbol{x}) = \big(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\big) \\ &\text{subject to } \boldsymbol{x} \in X \end{aligned}$$

where $\boldsymbol{x}$ is the *decision variable*, X denotes the set of feasible solutions, and $\boldsymbol{y}$ is the *objective vector* where each objective has to be minimized (or maximized).

Considering two decision vectors $\boldsymbol{a}, \boldsymbol{b} \in X$, $\boldsymbol{a}$ is said to *dominate* $\boldsymbol{b}$ if, and only if $\boldsymbol{a}$ is at least as good as $\boldsymbol{b}$ in all objectives and better than $\boldsymbol{b}$ in at least one objective. For shortening we will say that $\boldsymbol{a}$ dominates $\boldsymbol{b}$ by saying $dom(\boldsymbol{a}, \boldsymbol{b})$. Formally:

$$dom(\boldsymbol{a}, \boldsymbol{b}) = \begin{cases} \forall i \in \{1, 2, \ldots, m\} : f_i(\boldsymbol{a}) \geq f_i(\boldsymbol{b}) \text{ and} \\ \exists j \in \{1, 2, \ldots, m\} : f_j(\boldsymbol{a}) > f_j(\boldsymbol{b}) \end{cases}$$

A feasible solution $\boldsymbol{a} \in X$ is called *efficient* if it is not dominated by any other feasible solution. The set of all efficient solutions of a multiobjective optimization problem is known as *Pareto optimal*. Solving a multiobjective problem consists in giving its Pareto optimal set.

An instance of a multiobjective knapsack problem (MOKP) with m objectives consists of an integer capacity $W > 0$ and n items. Each item i has a

*Research supported by Fundação de Amparo à Pesquisa do Espírito Santo.

positive weight w_i and nonnegative integer profits $p_i^1, p_i^2, \ldots, p_i^m$. Each profit p_i^k represents the contribution of the i-th item for k-th objective. A solution is represented by a set $\boldsymbol{x} \subseteq \{1, \ldots, n\}$ containing the indexes of the items included in the solution. A solution is feasible if the total weight included in the knapsack does not exceed its capacity. Formally the definition of the problem is:

$$\begin{aligned} \max\ & f(\boldsymbol{x}) = \big(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_m(\boldsymbol{x})\big) \\ \text{subject to}\ & w(\boldsymbol{x}) < W \\ & \boldsymbol{x} \subseteq \{1, \ldots, n\} \end{aligned}$$

where

$$f_j(\boldsymbol{x}) = \sum_{i \in \boldsymbol{x}} p_i^j$$

$$w(\boldsymbol{x}) = \sum_{i \in \boldsymbol{x}} w_i$$

The MOKP is considered a $\mathcal{NP}$-Hard problem since it is a generalization of the well-known $0{-}1$ knapsack problem, in which $m = 1$. It is quite difficult to determine the Pareto optimal set for the MOKP, especially for high dimension instances, in which the Pareto optimal set tends to grow exponentially. Even for the bi-objective case, small problems may prove intractable. For this reason we are interested in developing efficient methods for handling large solution sets, which may bring tractability to previously intractable instances.

Considering two solutions $\boldsymbol{x}, \boldsymbol{y} \subseteq \{1, \ldots, n\}$, $\boldsymbol{y}$ is called *extension* of $\boldsymbol{x}$, denoted as $ext(\boldsymbol{y}, \boldsymbol{x})$, iff $\boldsymbol{x} \subseteq \boldsymbol{y}$. Any set $\boldsymbol{e} \subseteq \{1, \ldots, n\}$ such that $\boldsymbol{x} \cap \boldsymbol{e} = \emptyset$ is called an *extender* of $\boldsymbol{x}$. If $w(\boldsymbol{x}) + w(\boldsymbol{e}) \leq W$ than $\boldsymbol{e}$ is called a *feasible extender* of $\boldsymbol{x}$. A solution $\boldsymbol{x}$ is called *deficient* if it has available space to fit one or more item, i.e., $w(\boldsymbol{x}) + min\{w_i : i \notin \boldsymbol{x}\} \leq W$. We say $\boldsymbol{x}$ *knapsack-dominates* $\boldsymbol{y}$, denoted as $dom_k(\boldsymbol{x}, \boldsymbol{y})$, if $\boldsymbol{x}$ dominates $\boldsymbol{y}$ and does not weight more than $\boldsymbol{y}$. Formally:

$$dom_k(\boldsymbol{x}, \boldsymbol{y}) = \begin{cases} dom(\boldsymbol{x}, \boldsymbol{y}) & \text{and} \\ w(\boldsymbol{x}) \leq w(\boldsymbol{y}) \end{cases}$$

The concept of knapsack dominance was proposed by Weingartner and Ness [10] and is the basis the algorithm considered in this work which will be presented in the next section. Figure 1 illustrates the concept for a problem with $m = 1$. Any solution in the cross-hatched area knapsack-dominates the marked solution.

Theorem 1. *Consider two solutions $\boldsymbol{x}, \boldsymbol{y} \subseteq \{1, \ldots, n\}$ and $\boldsymbol{e}$ an extender for $\boldsymbol{x}$ and $\boldsymbol{y}$. If $dom_k(\boldsymbol{x}, \boldsymbol{y})$ than $dom_k(\boldsymbol{x} \cup \boldsymbol{e}, \boldsymbol{y} \cup \boldsymbol{e})$.*

Proof. □

3 The Dynamic Programming algorithm

The algorithm addressed in this work can be seen as a MOKP specialization of the classical Nemhauser and Ullmann's algorithm proposed in [6] for solving

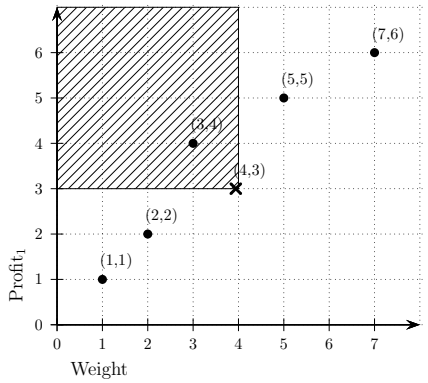


Figure 1: A knapsack-dominated solution.

knapsack problems. A basic multi-objective version of the algorithm will be presented in Section 3.1. Sections 3.2, 3.3 and 3.4 will cover the optimizations proposed by Bazgan considering some singularities of MOKP.

3.1 The Nemhauser-Ullmann algorithm

The Nemhauser and Ullmann's algorithm generically solves knapsack problems by applying the concept of knapsack dominance to remove solutions that will not lead to efficient ones. A basic multi-objective version of the algorithm is presented in Algorithm 1.

Algorithm 1 Nemhauser and Ullmann's algorithm for MOKP

1: **function** DP($\boldsymbol{p}, \boldsymbol{w}, W$)
2: $S^0 = \{\emptyset\}$
3: **for** $k \leftarrow 1, n$ **do**
4: $S^k_* = S^{k-1} \cup \{\boldsymbol{x} \cup k \mid \boldsymbol{x} \in S^{k-1}\}$ ▷ solutions extension
5: $S^k = \{\boldsymbol{x} \mid \nexists \boldsymbol{a} \in S^k_* : dom_k(\boldsymbol{a}, \boldsymbol{x})\}$ ▷ partial dominance filter
6: **end for**
7: $P = \{\boldsymbol{x} \mid \nexists \boldsymbol{a} \in S^n : dom(\boldsymbol{a}, \boldsymbol{x}) \mid w(\boldsymbol{x}) \leq W\}$ ▷ dominance/feasibility
8: **return** P
9: **end function**

The algorithm begins by defining an initial solution set S^0 containing only the empty solution (line 2). At a k-th stage the algorithm receives a set S^{k-1} exclusively containing solutions composed by the first $k-1$ items, i.e., $\forall \boldsymbol{x} \in S^k, \boldsymbol{x} \subseteq \{1, \ldots, k-1\}$. This set is then expanded by adding a copy of its solutions, but now including k-th item (line 4). This new set is defined by S^k_* thence having twice the cardinality of S^{k-1}. Set S^k is then defined by selecting all the solutions from S^k_* that are not knapsack-dominated by any other existing solution (line 5). The last step of the algorithm is selecting the solutions that are

not dominated by any other according to their objective values. Any solution in the context of stages prior than n-th stage will be considered a *partial solution*.

Algorithm 1 is quite powerful considering its simplicity. However... o carater potenciamente exponencial do problema may lead it degraded performance

3.2 Item ordering

An important issue in the MOKP is the ordering of the items. It is well-known that good solutions of knapsack problems are generally composed of items with the best cost-benefit rate. Therefore, the prioritization of those items tends to lead to better solutions.

The profit-weight relation for the single objective case may be directly derived from the profit/weight ratio of the items. However, for the multi-objective case, there is no such natural measure. The method proposed by [1] will be considered in this work, which is based on the ranking of items from their cost-benefit measures in all objectives. Those measures will give us several items orders which will be useful during the development of the algorithm.

We denote $\mathcal{O}^j$ the set of items ordered by ascending order of cost-benefit function regarding objective j. Formally, considering the function $cb^j(a) = p_a^j/w_a$ the cost-benefit function of item a regarding objective j, order $\mathcal{O}^j$ can be defined by:

$$\mathcal{O}^j = (o_1^j, \ldots, o_n^j), \qquad cb^j(o_1^j) \leq cb^j(o_2^j) \leq \cdots \leq cb^j(o_n^j)$$

Let r_i^j be the rank of item i in order $\mathcal{O}^j$. $\mathcal{O}^{sum}$, $\mathcal{O}^{min}$ and $\mathcal{O}^{max}$ denotes an order according to inscreasing values of the sum, minimum and maximum ranks. Formally:

$$\begin{aligned} r_i^j &= \max\{k \mid o_k^j = i\} \\ r_i^{sum} &= \textstyle\sum_{j=1}^{m} r_i^j \\ \mathcal{O}^{sum} &= (o_1, \ldots, o_n), \qquad r_{o_1}^{sum} \leq \cdots \leq r_{o_n}^{sum} \end{aligned}$$

The orders $\mathcal{O}^{min}$ and $\mathcal{O}^{max}$ are conceived in the same manner as $\mathcal{O}^{sum}$, except for the fact that $\frac{r_i^{sum}}{m}$ is added up in their ranks as tie breaking criteria. The notation $\mathcal{O}(\boldsymbol{s})$ will be used to denote the respective ordering of a $\boldsymbol{s}$ restrict set of items.

3.3 Avoiding deficient solutions

The first optimization that can be made on Algorithm 1 is avoiding the generation of deficient solutions. At k-th stage all previous solution is copied to the new solution set without adding k-th item (line 4). However preserving solutions with a lot of space left, concerning the remaining itens, may lead to deficient solutions.

Theorem 2. *Considering the k-th stage of the algorithm and $\boldsymbol{x} \in S^{k-1}$. If $w(\boldsymbol{x}) + \sum_{i \in \{k,\ldots,n\}} w_i \leq W$ than*

Considering the k-th stage, if a partial solution $\boldsymbol{x} \in S^{k-1}$ has enough space to fit all remaining items, i.e., $w(\boldsymbol{x}) + \sum_{i=k}^{n} w_i \leq W$, $\boldsymbol{x}$ may be discarded and only $\boldsymbol{x} \cup \{k\}$ keeped, once keeping $\boldsymbol{x}$ will certainly lead to deficient solutions.

3.4 Removing unpromissing solutions

Another optimization that can be applied on later stages is filtering unpromissing solutions by computing upper bounds for its objectives functions and comparing it with the set of available lower bounds. Considering a given k-th iteration, an upper(lower) bound of a partial solution is an upper(lower) limit each objective value can achieve, given its remaining capacity and the remaining items $(k+1, \ldots, n)$. A solution can be discharged if its upper bound is dominated by an existing lower bound, since it will generate no efficient solution.

A lower bound of a solution can be computed by greedily filling the knapsack with respect to $\mathcal{O}^{max}$ which is a good quality order of items. Formally:

$$lb(\boldsymbol{x}, \boldsymbol{s}) = \boldsymbol{x} \cup \left\{ o_i \;\middle|\; w(\boldsymbol{x}) + \sum_{j=1}^{i} w_{o_j} \leq W \right\}$$

where

$$(o_1, \ldots, o_k) = \mathcal{O}^{max}(\boldsymbol{s})$$

The upper-bound of a partial solution $\boldsymbol{x}$ is computed considering its available capacity and the remaining items on the current algorithm stage. To ensure the upper-bound its computation is done separately for. For the each j-th objective the reversed order $\mathcal{O}^j(\boldsymbol{s})$ on the set $\boldsymbol{s}$ of remaining items is considered to iteratively fill the remaining capacity of $\boldsymbol{x}$.

Algorithm 2 Upper-bound computation for a partial solution.

```
1:  function UB_j(p, w, W, x, s)
2:      w_left ← W − w(x)
3:      (o_1^j, ..., o_k^j) ← O^j(s)
4:      u_j ← f_j(x)
5:      i ← 1
6:      l ← o_i^j
7:      while w_left ≥ w_l and i ≤ k do
8:          u_j ← u_j + p_l^j
9:          w_left ← w_left − w_l
10:     end while
11:     if i ≤ k then
12:         l ← o_i^j
13:         u_j ← u_j + (w_left / w_l).p_l^j
14:     end if
15:     return u_j
16: end function
```

Algorithm 3 Bazgan's DP algorithm for the MOKP

function BazDP($\boldsymbol{p}, \boldsymbol{w}, W$)
  $S^0 = \{\emptyset\}$
  for $k \leftarrow 1, n$ **do**
    $S^k_* = \{\boldsymbol{x} \cup \{k\} \mid \boldsymbol{x} \in S^{k-1} \wedge (w(\boldsymbol{x}) + w_k \leq W)\}$
      $\cup \{\boldsymbol{x} \mid \boldsymbol{x} \in S^{k-1} \wedge (w(\boldsymbol{x}) + w_k + \ldots + w_n) > W\}$
    $S^k = \{\boldsymbol{x} \in S^k_* \mid (\nexists \boldsymbol{a} \in S^k_*)[dom_k(\boldsymbol{a}, \boldsymbol{x}) \vee dom(lb(\boldsymbol{a}), up(\boldsymbol{x}))]\}$
  end for
  return S^n
end function

4 The use of k-d tree

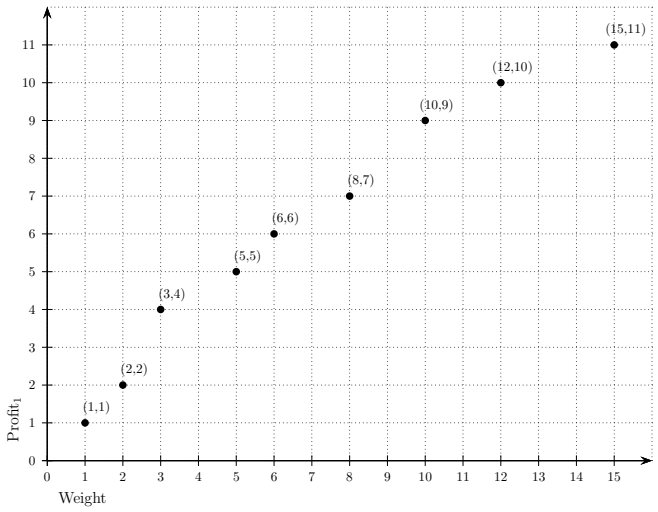
The k-d tree is a type of binary search tree for indexing multidimensional data with simple construction and low space usage. Despite its simplicity it efficiently supports operations like nearest neighbour search and range search [2]. For those reasons k-d tree is widely used on spacial geometry algorithms [8, 3], clustering [5, 4] and graphic rendering algorithms [7].

Like a standard binary search tree, the k-d tree subdivides data at each recursive level of the tree. Unlike a standard binary tree, that users only one key for all levels of the tree, the k-d tree uses k keys and cycles through these keys for successive levels of the tree.

Concerning it's efficiency, it is important to consider the number of dimensions k-d tree is indexing. As a general rule, a k-d tree is suitable for efficiently indexing of n elements if n is much greater than 2^k. Otherwise, when k-d tree are used with high-dimensional data, most of the elements in the tree will be evaluated and the efficiency is no better than exhaustive search [9].

Indexing the solutions and range operations.

Tends to increase the feasibility on problems with higher dimensions.



5 Computational experiments

- Base de dados utilizaca
- Parametros dos algoritmos
- Anlise dos resultados (comparao)

6 Conclusions and future remarks

- Concluses dos resultados
- Trabalhos futuros

References

[1] Cristina Bazgan, Hadrien Hugot, and Daniel Vanderpooten. Solving efficiently the 0–1 multi-objective knapsack problem. *Computers & Operations Research*, 36(1):260–279, 2009.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[4] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[5] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24(7):881–892, 2002.

[6] George L Nemhauser and Zev Ullmann. Discrete dynamic programming and capital allocation. *Management Science*, 15(9):494–505, 1969.

[7] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[8] Franco P Preparata and Michael Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[9] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. *Handbook of discrete and computational geometry*. CRC press, 2004.

[10] H Martin Weingartner and David N Ness. Methods for the solution of the multidimensional 0/1 knapsack problem. *Operations Research*, 15(1):83–103, 1967.