**Multidimensional Multiple-choice Knapsack Problem**

*Algoritmi avanzati,*

*Gabriel Mendonça Gomez*

*Leonardo Babbucci*

*Gabriele Dominelli*

Immagine che contiene diagramma

Descrizione generata automaticamente

Sommario

[Abstract 3](#_Toc133487357)

[Introduction 3](#_Toc133487358)

[Greedy algorithm 5](#_Toc133487359)

[Greedy in steps 5](#_Toc133487360)

[Data preparation: 5](#_Toc133487361)

[Pre-algorithm sorting: 5](#_Toc133487362)

[Initialization of the binary mask: 6](#_Toc133487363)

[First greedy iteration: 6](#_Toc133487364)

[Iterative greedy process: 6](#_Toc133487365)

[Value-based Prioritization and Binary Mask Combinations 7](#_Toc133487366)

[Table of results with standard inputs 8](#_Toc133487367)

[Flow chart 9](#_Toc133487368)

[Pseudo code 10](#_Toc133487369)

[Comparison with other algorithms: 10](#_Toc133487370)

[Trade-offs and considerations: 11](#_Toc133487371)

[Local Search 12](#_Toc133487372)

# Abstract

This report presents three novel algorithms developed to tackle the classical knapsack problem and its variations. The knapsack problem involves determining the optimal combination of items with given weights and values to maximize the total value of the knapsack without exceeding its weight capacity. The developed algorithms aim to address both the standard knapsack problem and the multiple-choice multidimensional knapsack problem (MMKP), which extends the problem to include multiple resource constraints and item classes. The report provides a comprehensive analysis of the proposed algorithms' efficiency, scalability, and performance on various instances of the knapsack problem, demonstrating their effectiveness in delivering optimal solutions.

# Introduction

The knapsack problem is a classical combinatorial optimization problem that has been widely studied in the fields of computer science, operations research, and mathematics. The problem can be defined as follows: given a set of items, each with a specific weight and value, determine the combination of items that maximizes the total value of a knapsack without exceeding its weight capacity. Mathematically, the problem is formulated as an integer (binary) linear programming model, with the objective function representing the maximization of the total value and the constraint ensuring that the total weight does not surpass the knapsack capacity.

Various real-world applications can be modeled using the knapsack problem framework, such as resource allocation, budgeting, and scheduling, making it a highly relevant and practical optimization problem. Additionally, the knapsack problem is classified as NP-hard, meaning that it is computationally challenging to find an exact solution in a reasonable time, especially for large problem instances. In this report, we present the development and analysis of three algorithms specifically designed to address the knapsack problem and its variations: a greedy algorithm, a local search algorithm, and a metaheuristic algorithm.

One of the variations addressed in this report is the multiple-choice multidimensional knapsack problem (MMKP). The MMKP extends the classical knapsack problem by considering multiple resource constraints and dividing the items into disjoint classes. The objective is to maximize the total value while respecting all capacity constraints and selecting exactly one item per class. As with the standard knapsack problem, the MMKP is also formulated as an integer (binary) linear programming model, albeit with additional constraints to account for the multiple resources and item classes.

The greedy algorithm is a simple yet effective approach that makes a series of myopic decisions by selecting items based on a specific criterion, such as the value-to-weight ratio. Although this algorithm does not guarantee an optimal solution, it provides an efficient method for obtaining a good initial solution, especially for large problem instances.

The local search algorithm is an iterative optimization method that starts with an initial solution and explores its neighboring solutions by making small changes to the current solution. The algorithm accepts better solutions and may also accept worse solutions based on a specific criterion, such as a probability function, to avoid getting trapped in local optima. The local search algorithm can provide high-quality solutions by effectively exploring the solution space and refining the initial solution obtained from the greedy algorithm.

Finally, the metaheuristic algorithm builds upon the concepts of both greedy and local search algorithms by incorporating higher-level strategies to guide the search process. Metaheuristic algorithms, such as genetic algorithms, simulated annealing, or tabu search, are designed to balance exploration and exploitation, providing a powerful and flexible framework for solving complex optimization problems, including the knapsack problem and its variations.

The three algorithms developed in this report are thoroughly analyzed in terms of their efficiency, scalability, and performance on various instances of the knapsack problem, including both standard and MMKP instances. Through extensive experimentation, the algorithms demonstrate their ability to deliver high-quality solutions while maintaining reasonable computational times. These novel algorithms not only contribute to the growing body of knowledge on solving the knapsack problem but also provide practical tools for tackling real-world problems that can be modeled using the knapsack problem framework. By integrating the strengths of greedy, local search, and metaheuristic algorithms, we offer a comprehensive and robust approach to solving the knapsack problem and its variations, paving the way for future research and applications in the field of combinatorial optimization.

# Greedy algorithm

The greedy algorithm is an effective and efficient approach to solving optimization problems, including the knapsack problem and its variations. In this particular implementation of the greedy algorithm, various combinations of items are tested in a systematic manner to identify a valid combination that maximizes the total value of the knapsack without violating the constraints. The core idea of the greedy approach is to make local, myopic decisions based on a specific criterion, which in this case is the value of the items.

In this implementation, the algorithm receives an ArrayList of MyClass objects as input, which represents the classes of items. The MyClass objects contain information about the items, such as their weights and values. The algorithm returns an array of integers, representing the position indices of the items chosen as the solution to the problem.

## Greedy in steps

The greedy algorithm can be broken down into several steps, as described below:

### Data preparation:

The algorithm starts by preparing the necessary data structures and variables required for its execution. The input ArrayList of MyClass objects is assigned to a variable named "classes", and an ArrayList named "classesSize" is created to store the number of items in each class. Additionally, an integer array named "chosenItem" is initialized to store the chosen items for the current combination, and another integer array named "usedWeights" is initialized to store the used weights for each resource in the multidimensional knapsack problem.

### Pre-algorithm sorting:

Before diving into the main greedy process, the items in each class are sorted based on their weight sums in descending order. This sorting is performed to prioritize the items with higher weights in the subsequent steps of the algorithm. This prioritization is based on the assumption that items with higher weights are more likely to contribute to a better solution, although it may not always hold true.

### Initialization of the binary mask:

A binary mask represented by a Boolean array named "itemMask" is created and initialized to have the same length as the number of classes. This mask is used to keep track of the items included in the current combination. Initially, all elements of the itemMask are set to false, indicating that no items have been chosen yet.

### First greedy iteration:

The algorithm performs its first greedy iteration by considering only the first item from each class. This is done by calling the "greedyHelper" method with the current itemMask as the input. If a feasible solution is found, it is stored in the "result" array and returned as the output of the algorithm.

### Iterative greedy process:

If the first greedy iteration does not yield a feasible solution, the algorithm proceeds to explore other combinations of items. It does so by calling the "generateCombinationsHelper" method, which generates combinations of items by recursively updating the itemMask. This process is performed for all possible combinations, starting from the second item in each class. The algorithm stops as soon as it finds a feasible solution that satisfies the constraints of the knapsack problem.

The key strength of the greedy algorithm lies in its simplicity and efficiency. By iteratively testing combinations of items based on their sorted order and updating the itemMask, the algorithm systematically explores the solution space and identifies valid solutions that maximize the total value of the knapsack. However, it is important to note that the greedy approach does not guarantee an optimal solution, as it relies on local, myopic decisions rather than a global perspective of the problem.

In summary, this greedy algorithm for the knapsack problem and its variations follows a step-by-step process that involves data preparation, pre-algorithm sorting, initialization of the binary mask, and an iterative greedy procedure. By sorting the items based on their weight sums, prioritizing higher-weight items, and systematically exploring combinations of items, the algorithm efficiently searches for feasible solutions that maximize the total value of the knapsack while adhering to the given constraints. Its performance is contingent upon the nature of the problem and the quality of the input data, but it often delivers satisfactory results in a relatively short amount of time.

## Value-based Prioritization and Binary Mask Combinations

In the developed greedy algorithm, the primary criterion for choosing items is their value, with items prioritized by their weight sums in descending order. To implement this, the algorithm starts with a pre-algorithm sorting stage, where items in each class are sorted based on their weight sums. The rationale behind this ordering is to prioritize items with higher weights, as they are assumed to contribute more to a better solution.

The algorithm generates new combinations of items using a binary mask, which is a Boolean array of the same length as the number of classes. The binary mask is initialized with all elements set to false, indicating no items have been chosen. It then runs a greedy iteration, considering only the first item in each class. If this iteration does not yield a feasible solution, the algorithm proceeds to explore other item combinations using the generateCombinationsHelper method.

This method is responsible for generating a series of binary masks that determine the items to be used in the next iteration. It works by recursively shifting the only present "true" value in the mask until the end of the mask vector, then adding a new "true" value and shifting them until every combination has been evaluated. The last generated mask will contain only "true" values.

The greedyHelper method evaluates a specific set of item indices held by the chosenItem attribute using the binary mask. It checks if the current chosen items combination is valid by verifying whether their weights can fit inside the knapsack. If a valid combination is found, the algorithm stops and returns the position indices of the chosen items as a solution to the problem.

In summary, this greedy algorithm chooses items based on their value and weight sums, orders items by prioritizing those with higher weights, and generates new combinations of items using a binary mask. By systematically exploring the solution space and evaluating different combinations of items, the algorithm efficiently searches for feasible solutions that maximize the total value of the knapsack.

## Table of results with standard inputs

Immagine che contiene tavolo

Descrizione generata automaticamente

## Flow chart

Immagine che contiene diagramma

Descrizione generata automaticamente

## Pseudo code

Immagine che contiene testo

Descrizione generata automaticamente

## Comparison with other algorithms:

While the greedy algorithm offers a fast and straightforward approach to solving the knapsack problem, other algorithms such as local search and metaheuristic methods may yield better results in certain scenarios. Local search algorithms, including hill climbing and simulated annealing, focus on iteratively refining a candidate solution by making small adjustments to it, seeking to find a local optimum in the solution space. These algorithms can sometimes overcome the limitations of the greedy approach by considering a broader range of solutions and escaping local optima.

Metaheuristic algorithms, such as genetic algorithms and particle swarm optimization, employ higher-level heuristics inspired by natural processes to guide the search for an optimal solution. These methods are typically more computationally expensive than greedy algorithms but can often produce higher-quality solutions in complex problem domains. By incorporating elements of both exploration and exploitation, metaheuristic algorithms balance the need to search globally for an optimal solution while also taking advantage of the best solutions found thus far.

## Trade-offs and considerations:

When selecting an algorithm for solving the knapsack problem or a similar optimization problem, it is essential to weigh the trade-offs between different approaches. Greedy algorithms offer the advantage of speed and simplicity, making them suitable for problems with tight time constraints or limited computational resources. However, their reliance on local decisions may limit their ability to find globally optimal solutions.

In contrast, local search and metaheuristic algorithms can often deliver better solutions by exploring a broader range of possibilities and employing more sophisticated search strategies. These algorithms may require more computational resources and a longer runtime but may be better suited to complex problem domains or situations where a higher-quality solution is necessary.

Ultimately, the choice of algorithm depends on the specific requirements of the problem and the desired balance between computational efficiency and solution quality. In some cases, a hybrid approach that combines elements of greedy, local search, and metaheuristic algorithms may offer the best of both worlds, providing an effective means of tackling challenging optimization problems.

In conclusion, the greedy algorithm presented here offers a straightforward and efficient approach to solving the knapsack problem and its variations. By iteratively testing combinations of items, prioritizing higher-weight items, and systematically exploring the solution space, the algorithm efficiently searches for feasible solutions that maximize the total value of the knapsack. While it may not always guarantee an optimal solution, the greedy approach can often deliver satisfactory results in a short amount of time, making it a valuable tool in the arsenal of optimization techniques.

# Local Search