Project - Multidimensional Unitary-profit Precedence-constrained Knapsack Problem

Lucas Guesser Targino da Silva - RA: 203534

July 17, 2022

Abstract

We present a generalization of the classic 0-1 Knapsack Problem. In this problem, the weight of the items are multidimensional vectors and so is the knapsack capacity, the profit of all items is equal to one, and the items are required to be added in a certain order. That problem is refered as *Multidimensional Unitary-profit Precedence-constrained Knapsack Problem (MEPKP)*. Three solution approaches are proposed: an Integer Linear Programming for a exact solution; a greedy algorithm for a fast solution; a Greedy Randomized Adaptive Search Procedures (GRASP) and Tabu Search (TS) for a near optimal solution. The three approaches will be compared in terms of quality of the solution and computational time with randomly generated instances.

1 Problem Statement

1.1 Input

1. a directed acyclic graph $G = \langle V, E \rangle$;
2. a multi-dimensional weight function $w : V \to \mathbb{Z}_{>}^{n_w}$, where $n_w \in \mathbb{N}$;

   We will usually write $w_v = w(v)$
3. a maximum capacity of the knapsack $W \in \mathbb{Z}_{>}^{n_w}$;

Besides that, one requires the input to satisfy the constraints below [1], otherwise the problem would be trivial:

1. $w_v \leqslant W$: the weight of each vertex must be smaller than the knapsack capacity;
2. $\sum_{v \in V} w_v \geqslant W$: the weight of all vertices combined must be greater than the knapsack capacity;

1.1.1 Partial Order

Definition 1 (Partial Order on Directed Acyclics Graph)**.** Given a directed acyclic graph $G = \langle V, E \rangle$, we define the set:

$$\prec = \{\langle v, v' \rangle : \text{there is a path from the second to the first}\} \tag{1}$$

and so $\prec$ is a partial order over the set V.

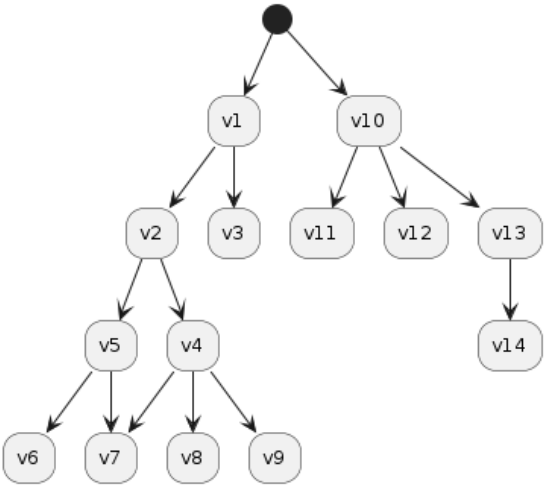

Figure 1: Example of a directed acyclic graph. The black dot indicates the root vertices. For this case, the induced partial order satisfy: $v5 \prec v2$, $v7 \prec v1$, $v14 \prec v10$.

1.2 Output

A subset $S \subseteq V$ of the vertices which satisfy:

$$\sum_{v \in S} w_v \leqslant W \tag{2}$$

$$\forall v \left(v \in S \rightarrow \forall v' \left(v \prec v' \rightarrow v' \in S \right) \right) \tag{3}$$

Equation (2) states the total weight of all vertices in the solution set S must not be greater than the weight limit W. It is called Capacity-constraint.

Equation (3)[1] says that, if a v is included in the solution, then all the v' greater than it (in the sense of the partial order $\prec$) must also be included. It is called Precedence-constraint.

1.3 Objective

Find S that maximizes $|S|$. In other words: find the solution with the maximum number of vertices.

2 State of the Art

In [3], the authors proposes a memory based GRASP for 0-1 quadratic knapsack problem with restart and a simple tabu search algorithm is proposed to overcome the limitations of local optimality in order to find near optimal solutions. In that paper, numerical tests on benchmark instances demonstrate the effectiveness and efficiency of the proposed methodology which outperform the Mini-Swarm heuristic in terms of the success ratio, relative percentage deviation and computational time.

[1] It is a First-order logic expression [2].

In [4], the authors reported the implementation of an efficient TS method based on the oscillation strategy and definition of a promising zone, a zone which englobes all feasible solutions plus all unfeasible solutions bordering the unfeasible solutions, for solving the O-1 MKP which has been tested on standard test problems from [5, 6] and [7]. Optimal solutions were successfully obtained for all instances and the previously best known solutions were improved for five of the last seven instances. These numerical results were claimed to confirm the merit of tabu tunneling approaches to generate solutions of high quality for 0-1 multiknapsack problems. Moreover, these results (like those of [7]) are claimed to establish that the oscillation strategy is efficient to balance the interaction between intensification and diversification strategies of TS.

In [1], the authors used a lagrangean relaxation on the precedence-constrained and the subgradient method to solve the problem faster then use a "pegging" test to guarantee optimality.

2.1 0-1 Knapsack Problem

We present it because of its simplicity, relavance in the literature, and similarity with the problem proposed in this problem. In [8], the authors define the 0-1 Knapsack Problem (0-1KP):

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subjected to} \quad & \sum_{j=1}^{n} w_j x_j \leq c \\ & x_j \in \{0,1\} \quad \forall j \in \{1,\ldots,n\} \end{aligned} \tag{4}$$

in which p_j and w_j are known as the profit and the weight of the item j, respectively.

The problem proposed here is a knapsack problem adapted to satisfy two extra constraints: precedence-constrained and multi-dimensional weights. Besides, its profits are all one.

3 Solving Methodologies

In this section, one presents all the methodologies proposed to solve the MUPKP.

3.1 Integer Linear Programming (ILP)

3.1.1 Decision Variables

For each vertex $v \in V$ of the input graph $G = \langle V, E \rangle$, we define the binary variable $x_v \in \{0,1\}$ which indicates whether v is in the solution S:

$$x_v = \begin{cases} 1 & , v \in S \\ 0 & , v \notin S \end{cases} \tag{5}$$

3.1.2 Mathematical Model

Below, Equation (6) is the objective function: maximize the number of vertices in the solution. Equation (7) is the Capacity-Constraint of Equation (2). Equation (8) is the Precedence-Constraint of Equation (3): if a vertex v is in the solution, then all vertices v' "above" it must also be in the solution.

$$\max_{S \subseteq V} \sum_{v \in S} x_v \tag{6}$$

$$s.t. \sum_{v \in S} x_v w_v \leqslant W \tag{7}$$

$$x_v \leqslant x_{v'} \quad \forall v \prec v' \tag{8}$$

$$x \in \{0,1\}^{n_w} \tag{9}$$

3.2 Greedy Criteria

Since the problem is a unitary-profit, the objective function provides no information about which vertex is better to add to the solution. Being more practical, for designing an algorithm, making decisions based solely on the objective function is useless and meaningless. For that reason, we propose the following auxiliary function g, called Greedy Criteria:

Definition 2 (Greedy Criteria)**.** Let be a vertex and w_v its weight. The Greedy Criteria is the function $g : V \to \mathbb{Z}_>$ which associates a vertex $v \in V$ to the entry of its weight vector $w_v \in \mathbb{Z}_>^{n_w}$ with the highest value:

$$g(v) = \max(w_v) \tag{10}$$

In metaheuristics, we are usually interested in greedy strategies. For the approaches analyzed in this project, the greedy criteria is going to be: select the vertex which minimizes the value of g. The intuition behind such choice is clear: we want to add vertices which weight as little as possible.

Notice that the above definition is not the only one available. One could choose to use the norm 2 or average value of the weight vector w_v. The election of the maximum is relies on the intuition that "averages" might fill too much one of the dimensions of the weight while leaving others free. That would cause early stop of the algorithms. The maximum function, on the other hand, ensures that dominating values of the dimension of the weight are properly handled.

Of course, using the maximum function has drawbacks as well. Since it looks only to the most loaded entry, between two vertices with the same value of the greedy function g, it won't see which one has lower values in the other entries.

3.3 GRASP

The GRASP (Greedy Randomized Adaptative Search Procedure) is presented in [9]. A general pseudocode of it is presented Algorithm 1.

Algorithm 1 GRASP

Require: N_i, α
1: $S^* \leftarrow \varnothing$
2: **for** $i = 1, \ldots, N_i$ **do**
3: $\quad S_0 \leftarrow$ GRASP-Construction(α)
4: $\quad S_+ \leftarrow$ GRASP-Local-Search(S)
5: $\quad$ **if** $\|S_+\| > \|S^*\|$ **then**
6: $\quad\quad S^* \leftarrow S_+$
7: **return** S^*

3.3.1 Number of iterations

We propose to use as the number of iterations N_i the square root of the number of nodes:

$$N_i = \sqrt{|V|} \tag{11}$$

Such criteria is interesting for two main reasons:

1. it grows with the size of the input
2. it does not grow at the same rate as the size of the input grows

As pointed out in 1 above, it is natural to think that, as the size of the problem grows, so should the number of iterations, in order to better cover the space of feasible solutions.

However, the number of iterations must not grow too much with the size of the input. That's exactly why we proposed to use the square root. As it generates new solutions, because of the greedy criteria, it tends to explore similar locations, so more iterations will not provide much more coverage of the space of feasible solutions.

3.3.2 Greedy Randomized Construction

We used Algorithm 2, the same one proposed in [9].

In the algorithm, α is known as Greedy Parameter, it controls the balance between greediness and randomness ($\alpha = 0$ is purely greedy, $\alpha = 1$ is purely random).

Algorithm 2 GRASP-Construction

Require: α
$S \leftarrow \varnothing$
$C \leftarrow \{v \in V \backslash S : S \cup \{v\}$ does not violate the constraints$\}$
while $C \neq \varnothing$ **do**
  $g_{min} = \arg\min_{v \in C} g(v)$
  $g_{max} = \arg\max_{v \in C} g(v)$
  $RC \leftarrow \{v \in C : g(v) \leqslant g_{min} + \alpha \cdot (g_{max} - g_{min})\}$
  $v \leftarrow$ pick an element of RC at random
  $C \leftarrow \{v \in V \backslash S : S \cup \{v\}$ does not violate the constraints$\}$
return S

Notice that $\{v \in V \backslash S : S \cup \{v\}$ does not violate the constraints$\}$ is all the vertices v which:

1. are successors of the vertices in the solution S
2. have all the predecessors in the solution S
3. its weight w_v plus the weight $w_S = \sum_{v' \in S} w_{v'}$ of the solution S does not exceed the capacity W of the knapsack

3.3.3 Local Search

The idea is to find a local optimal. For that, the Algorithm 3 fits as many vertices as possible in the solution. A substitution (step 5 above) might free up enough space for a new vertex to be added, that's why the search is in a **while** loop.

Algorithm 3 GRASP-Local-Search

Require: S

1: **while** S changed in the last iteration **do**
2: attempt to add the vertex to S that minimizes the Greedy Criteria of the sum of the weights (solution + vertex) and satisfy all contraints
3: **if** a vertex has been added **then**
4: **continue**
5: attempt to substitute one vertex of S by one vertex outside S which does not violate the constraints and minimize the Greedy Criteria of the combined weights (solution + vertex to add + vertex to remove)
6: **return** S

3.3.4 Parameters Selection

For the implementation of GRASP of this project, we decided to use the Greedy Criteria g presented in the Subsection 3.2. As for the Greedy Parameter α, we decided to use $\alpha = 0.2$.

3.4 Greedy

It is a simple algorithm: at each iteration, add the vertex which minimizes the Greedy Criteria g and does not violate the Precedence-constraint (Equation (3)). When no more vertex can be added without violating the Capacity-constraint (Equation (2)), stop.

Although being quite easy to define and implement this algorithm, we decided to implement it using GRASP with:

1. Greedy Parameter $\alpha = 0$. As discussed in Subsubsection 3.3.2, it makes the algorithm purely greedy;
2. Maximum number of iterations $N_i = 1$. Being purely greedy, there is no randomness, so there is also no need to run it more than once;

3.5 Tabu

The Tabu Search implemented is presented in [10]. It is characterized by diversification by restart and presenting strategic oscillation.

Algorithm 4 Tabu

Require: N_i, α

1: $S^* \leftarrow \varnothing$
2: **for** $i = 1, \ldots, N_i$ **do**
3: $\quad S_0 \leftarrow$ GRASP-Construction(α)
4: $\quad S_+ \leftarrow$ Tabu-Local-Search(S)
5: $\quad$ **if** $\|S_+\| > \|S^*\|$ **then**
6: $\quad\quad S^* \leftarrow S_+$
7: **return** S^*

You may notice that the algorithm is very similar to Algorithm 1, GRASP. Usually, the Tabu Search algorithm does not include the iterations (loop in line 2). However, that is the modification we propose for this project: to use a diversification by restart approach. The reason for adopting that is because the problem seems to require different areas to be explored. Given an intial solution, it may not be easy to explore areas out of its vinicity. A diversification by restart approach aims to solve that exactly problem.

In fact, the difference between the Algorithm 1 and Algorithm 4 is the local search method: Tabu-Search.

3.5.1 Tabu-Search

The Algorithm 5 is the Tabu-Search procedure. It is a variation of what is know as "strategic oscillation". Its mechanism is described below.

First, vertices are added to the solution allowing it to temporarily exceed, by a factor W_r^+ (called Capacity Expansion Ratio), the knapsack capacity W. Second, it performs substitutions of vertices. So far, it seems very similar to the Algorithm 3. But Tabu-Search has one more step, required to make the solution feasible again: remove vertices till the knapsack capacity is satisfied once again and the solution becomes feasible. For the removal, at each step, it selects the vertex which maximizes the Greedy Criteria, in other words, the heaviest one.

There is yet another very important difference between Tabu-Search and GRASP-Local-Search: the former does not allow vertices in the tabu list to be changed (each time it changes the solution, it records what has been done in the tabu list). whereas the latter doesn't really requires that since it tends to always increase the weight of the solution. For the Tabu-Search, not recording teh moves might mean it would go back to a prevoius visited solution.

Finally, the process runs till a certain number of iterations has passed without improvements in the current best solution known S^*.

Algorithm 5 Tabu-Search

Require: S, t_r, W_r^+, N_{wi}

1: $S^* \leftarrow S$
2: $T \leftarrow$ a vector of $t_r \cdot |V|$ entries filled with -1[2]
3: **while** number of iterations without improvement $< N_{wi}$ **do**
4: add elements to S while while $\sum_{v \in S} w_v \leqslant^* (1 + W_r^+) \cdot W$
5: substitute elements of S while possible
6: remove elements of S while $\sum_{v \in S} w_v >^* W$
7: **if** $\|S\| > \|S^*\|$ **then**
8: $S^* \leftarrow S$ ▷ Improvement found
9: **return** S

Obs: $x \leqslant^* y$ is true when ALL components of the vector x are smaller than the ones of y. Obs 2: $x >^* y$ is true when that ANY component of the vector x is bigger than the ones of y.

For combinatorial optimization problems, near-optimal solutions are mostly at the border of the feasibility space, and so there are many infeasible solutions around it.

The Tabu-Search procedure aims to optimize the search by taking shortcuts throught the space of the infeasible solutions. It hopefully jumps from one local optimal to the other, without going back to previously visited solutions (thanks to the tabu list).

3.5.2 Parameters Selection

For the implementation of Tabu of this project, we decided to use:

1. the Greedy Criteria g presented in the Subsubsection 3.2;
2. Greedy Parameter $\alpha = 2$ (construction phase);
3. Tenure Ratio $t_r = 0.4$;
4. Capacity Expansion Ratio $W_r^+ = 20\%$
5. Number of iterations without improvement $N_{wi} = 10$

4 Instance Generation

Since no instance for the problem proposed here was found in the literature, it becomes necessary to create the instances in this project. This section proposes a problem instances generation method. It is divided in three parts: graph, weight of the vertices, knapsack capacity.

4.1 Graph Generation

As stated earlier, the precedence contraint is defined by a Directed Acyclic Graph (DAG). We are actually going to prove a very interesting results:

Theorem 1. Given an problem instance I defined by a Directed Graph, it can be reduced to an instance I' defined by a Directed Acyclic Graph Transitively Reduced [11].

Proof. It follows directly from Lemma 1 and Lemma 2. □

4.1.1 Removing cycles: Directed Graph to Directed Acyclic Graph

Lemma 1. Given an problem instance I defined by a Directed Graph, it can be reduced to an instance I' defined by a Directed Acyclic Graph.

Proof. Suppose that I contains at most one cycle. Let:

1. a cycle $C = \{u_1, \ldots, u_m\} \subseteq V$ defined by its vertices;
2. $E_{in} = \{\langle u, v\rangle : v \in C\}$ the edges that point to the vertices of the cycle C;
3. $E_{out} = \{\langle u, v\rangle : u \in C\}$ the edges that point from the vertices of the cycle C;

Create a new graph replacing:

1. the cycle C by a vertex U with weight $\sum_{u \in C} w_u$;
2. the edges E_{in} by edges that point from the same vertices as originally to the vertex U;
3. the edges E_{out} by edges that point from U to the same vertices as originally;

Notice that if both Precedence-constraint and Capacity-constraint are satisfied in I, then they are also satisfied in I'. Therefore, both instances are equivalent. For the general case in which I has more than one cycle, one has to simply run the procedure described above for each cycle. □

4.1.2 Transitive Reduction: Directed Acyclic Graph to Directed Acyclic Transitively Reduced Graph

Definition 3. The transitive reduction of a graph G is the graph G' which has as few edges as possible but the same transitive closure (reachability) of G [11].

Obs: given a graph G and its transitive reduction G', I am refering to G' as G "Transitively Reduced".

Lemma 2. Given an problem instance I defined by a Directed Acyclic Graph, it can be reduced to an instance I' defined by a Directed Acyclic Graph Transitively Reduced.

Proof. It follows directly from the fact that the transitive reduction preserves thetransitive closure (reachability). □

4.1.3 How to generate a Directed Acyclic Transitively Reduced Graph

Put simply:

1. Generate a Directed Acyclic Graph Transitively Reduced by generating a Directed Acyclic Graph and computing one of its transitive reduction;
2. Generate a Directed Acyclic Graph by generating a random upper triangular connectivity matrix (with only ones and zeros) with the main diagonal null (zero);

There are several computational tools that implement the functionalities above. For this project, we used [12] for generating the random matrix and [13] for the graph manipulation.

4.1.4 Graph Generation Parameters

The following parameters are used to control the creation of the graph:

1. number of nodes: integer positive number
2. edge probability: a number between 0 and 1 (inclusive). It is the probability of each edge to exist. In some way, it controls the number of edges of the instance;

4.2 Vertices Weight and Knapsack Capacity

Both the weight of a vertex and the Knapsack Capacity are a multidimensiona vectors of positive integer entries. To generate them, it is as simple as generating some random numbers in a specific range of values and organizing it so that one gets a vector. For this, [12] was used.

We want some sort of relation between the knapsack capacity and the weight generated for all vertices. What we do is:

1. generate the weight of all vertices;
2. compute the sum of the weight of all vertices and set the knapsack capacity as a fraction of such value;

In that way, there is some sort of (statictical) guarantee that some but not all vertices are going to fit into the knapsack.

4.2.1 Vertices Weight Generation Parameters

1. weight size: the size or dimension of the weight vector;
2. weight minimum value and weight maximum value: they define the interval in which the values must be;
3. percentage of nodes to fit: a number between 0 (zero) and 1 (one) exclusive, it is the fraction used to multiply sum of the weight of all vertices in order to set the knapsack capacity;

5 Results

5.1 Computational Environment

The GRASP, Tabu, and Greedy algorithms were implemented in the Java programming language (Java version 17 and gradle version 7). The Integer Linear Programming was solved using Gurobi 9.5 and Python.

All the experiments were executed in a ideapad S145 81S90005BR: Lenovo IdeaPad S145 Notebook Intel Core i5-8265U (6MB Cache, 1.6GHz), 8GB DDR4-SDRAM, 460 GB SSD, Intel UHD Graphics 620.

The operating system was a Fedora 35 executando o Java 17 e Gradle 7. O código desenvolvido pode ser encontrado em [14].

5.2 Tables

name	problem_info			ilp		greedy		grasp		tabu	
	capacity	edges	nodes	cost	time[s]	cost	time[s]	cost	time[s]	cost	time[s]
problem											
N100_E0_W51666	51666	0	100	72	0.001373	69	0.008	70	0.031	70	0.025
N100_E0_W52398	52398	0	100	72	0.002125	69	0.039	69	0.126	69	0.102
N100_E111_W52365	52365	111	100	71	0.002847	70	0.009	70	0.025	70	0.044
N100_E118_W52247	52247	118	100	71	0.003379	70	0.017	71	0.043	70	0.078
N100_E213_W52199	52199	213	100	70	0.006474	70	0.006	70	0.014	69	0.056
N100_E222_W53297	53297	222	100	70	0.011230	69	0.002	69	0.014	69	0.036

Table 1: Cost and running time of all metaheuristics for problem instances with 100 nodes.

name	problem_info			ilp		greedy		grasp		tabu	
	capacity	edges	nodes	cost	time[s]	cost	time[s]	cost	time[s]	cost	time[s]
problem											
N500_E570_W262641	262641	570	500	361	0.010798	355	0.129	355	1.729	353	1.086
N500_E594_W262931	262931	594	500	360	0.017267	353	0.127	356	1.524	353	1.040
N500_E598_W263031	263031	598	500	359	0.024541	353	0.245	354	1.985	353	1.712
N500_E1157_W262412	262412	1157	500	357	0.065347	351	0.072	353	0.595	352	1.046
N500_E1169_W262155	262155	1169	500	356	0.061799	350	0.036	353	0.583	351	1.016
N500_E1189_W260774	260774	1189	500	357	0.056502	353	0.058	353	0.621	351	0.990

Table 2: Cost and running time of all metaheuristics for problem instances with 500 nodes.

name	problem_info			greedy		grasp		tabu	
	capacity	edges	nodes	cost	time[s]	cost	time[s]	cost	time[s]
problem									
N1000_E954_W523944	523944	954	1000	707	1.311	710	27.659	707	8.613
N1000_E975_W523980	523980	975	1000	708	0.702	712	22.837	712	7.913
N1000_E1017_W523215	523215	1017	1000	708	0.751	711	22.985	708	8.659
N1000_E1925_W523221	523221	1925	1000	706	0.489	707	10.759	703	7.565
N1000_E1956_W527027	527027	1956	1000	705	0.549	708	10.030	705	7.726
N1000_E2032_W526397	526397	2032	1000	706	0.235	708	7.158	701	6.929

Table 3: Cost and running time of all metaheuristics for problem instances with 1000 nodes.

5.3 Performance Profiles

For those, a tolerance of 1% is admitted in the solution. In other words, if an algorithms found a solution as good as the best one found (within a 1% tolerance), then it is considered that it solved the problem in that run.

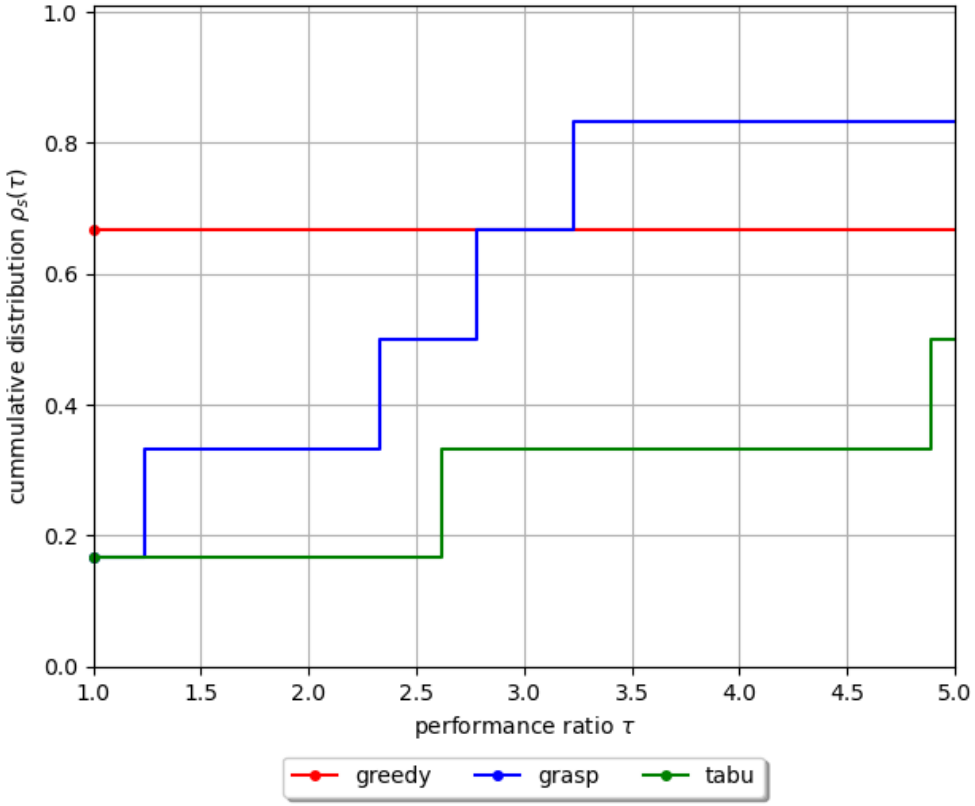

Figure 2: Performance Profiles for the three algorithms for the instances of 100 vertices.

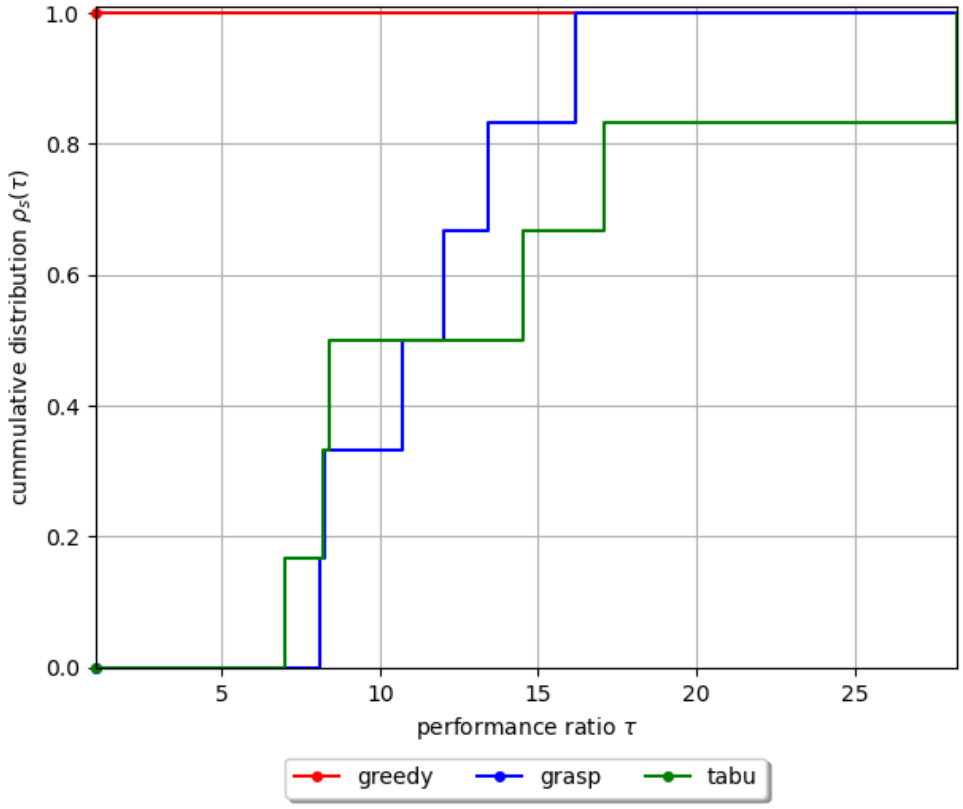

Figure 3: Performance Profiles for the three algorithms for the instances of 500 vertices.

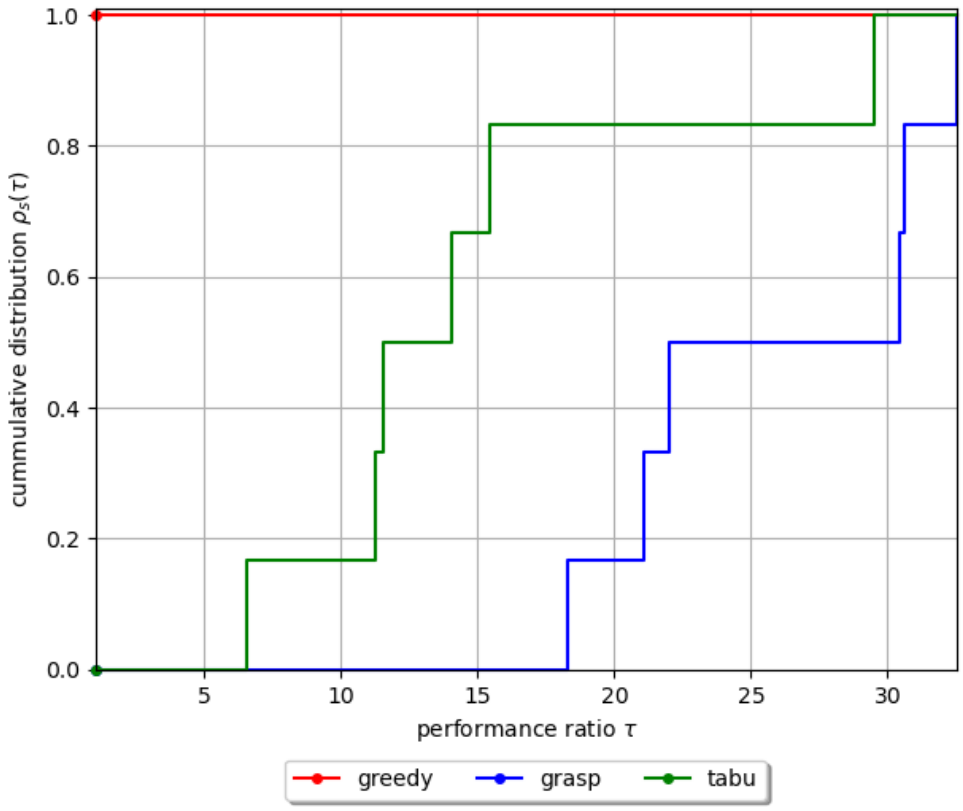

Figure 4: Performance Profiles for the three algorithms for the instances of 1000 vertices.

6 Analysis of the Results

6.1 Analysis of the Tables 1, 2, 3

Looking them, it is clear that the ILP was the dominant algorithm. It is guarantee to find the optimal solution, what would make one wonders that it would be slower. Interestingly, quite the opposite happened: it outperformed all the other algorithms.

Notice that ILP did not solve the instances with 1000 vertices. That is not because it can't, but because the Gurobi license available doesn't allow one to run models that large.

From the tables, it is possible to notice GRASP always gave a better result, in terms of cost, than Greedy. That is expected: GRASP explores more the solution domain and so it is able to find better ones. That comes with a price: higher running time. In all cases, GRASP ran more than 4 times slower than Greedy. One can notice that the difference get bigger as one increases the domain size. That is because the way GRASP was implemented: to search more in bigger problems.

It is interesting to notice that Tabu started slower than GRASP in the instances with 100 vertices, then they were more or less equivalent in the instances with 500 vertices, and for 1000, Tabu is considerably faster. That is actuall quite conta-intuitive. As stated in Section 3.5, the only difference between Tabu and GRASP is the local search method. One would expect Tabu to search more,a and so to take longer. That is not what we observe though.

We expected the algorithms to perform worse for instances without edges, as it is the case of the first two cases in the Table 1. That is because, without the precedence constraints, there are just too many ways of combining the items, and so it should be difficult to find the best combinations. That is not what we observe though. Except ILP, all the other algorithms had a slightly worse performance (it is really small the difference).

We notice that Tabu got results worse than GRASP in all cases. We expected the oscilation plus diversification strategies to have performed better than GRASP, to take the sortcuts and find better solutions. For the cases with 100 and 500 vertices, it is clear that it had the possibility since GRASP didn't find the optimal solution.

Finally, notice that, balacing performance and quality, Greedy performed better than GRASP and Tabu: its results are as good as GRASP and its running time is much faster.

6.2 Analysis of the Performance Profiles 2, 3, 4

The first thing we notice is that in both Figures 3 and 4, Greedy finds the near-optimal solution at least 5 times faste than GRASP and Tabu. For the instances of the figure 2, it is the fastest approximately 65% of the times.

Notice that the choice of the 1% tolerance is quite arbitrary. In practice, considering that it is related to an allocation problem, that would be a fairly good value.

In all those figures, ILP was not included because it would dominate: it finds the optimal solution most of the times faster than the metaheuristics implemented in this project.

Comparing GRASP and Tabu, one can see the same behavior pointed out in the last section: GRASP performs better in small instances (100 vertices), they are almost equivalent for 500 vertices, and Tabu performs better in the instances with 1000 vertices.

References

[1] Byungjun You and Takeo Yamada. A pegging approach to the precedence-constrained knapsack problem. *European journal of operational research*, 183(2):618–632, 2007.

[2] Cezar A Mortari. *Introdução à lógica*. Unesp, 2001.

[3] Zhen Yang, Guoqing Wang, and Feng Chu. An effective grasp and tabu search for the 0–1 quadratic knapsack problem. *Computers & Operations Research*, 40(5):1176–1185, 2013.

[4] Said Hanafi and Arnaud Freville. An efficient tabu search approach for the 0–1 multidimensional knapsack problem. *European Journal of Operational Research*, 106(2-3):659–675, 1998.

[5] A Freville and G Plateau. *Méthodes heuristiques performantes pour les problèmes en variables 0-1 à plusieurs contraintes en inégalité*. Université de Lille I, UER d'IEEA Informatique, 1982.

[6] Arnaud Freville and Gérard Plateau. An efficient preprocessing procedure for the multidimensional 0–1 knapsack problem. *Discrete applied mathematics*, 49(1-3):189–212, 1994.

[7] Fred Glover and Gary A Kochenberger. Critical event tabu search for multidimensional knapsack problems. In *Meta-heuristics*, pages 407–427. Springer, 1996.

[8] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[9] Mauricio GC Resende and Celso C Ribeiro. Greedy randomized adaptive search procedures: advances and extensions. In *Handbook of metaheuristics*, pages 169–220. Springer, 2019.

[10] Michel Gendreau and Jean-Yves Potvin. Tabu search. In *Search methodologies*, pages 165–186. Springer, 2005.

[11] Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[12] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[13] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[14] Lucas Guesser Targino da Silva. Mo824a-combinatorial-optimization. `https://github.com/lucasguesserts/MO824A-combinatorial-optimization/tree/activity-5/activity_5`.

A Examples of Instances - Drawings

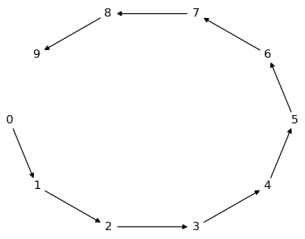

Figure 5: Example of an instance too easy to solve, the vertices are too tight to one another.

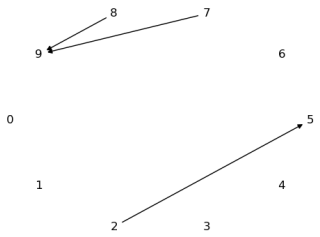

Figure 6: Example of an instance too difficult to solve, the vertices are too loose from one another.

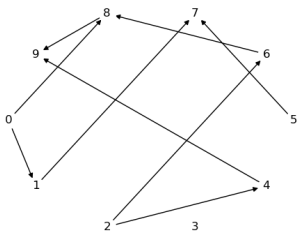

Figure 7: Example of a good instance, not too hard, not too easy. The vertices are somewhat tight to one another, but not too much.