

The Path-Precedence Constrained Knapsack Problem with Unit Weights: Exact Algorithms, Linear Formulations and Instances

Cristiano Martins Monteiro

Shipping Optimization Team (SOT) - Mercado Libre
3003 Nações Unidas Ave., Osasco, SP 06233-903, Brazil
cristiano.monteiro@mercadolivre.com

Tatiane Fernandes Figueiredo

Shipping Optimization Team (SOT) - Mercado Libre
3003 Nações Unidas Ave., Osasco, SP 06233-903, Brazil

Joaquín del Priore

Shipping Optimization Team (SOT) - Mercado Libre
4789 Posta St., Buenos Aires, C1430EKH, Argentina

Eduardo Candido Xavier

Shipping Optimization Team (SOT) - Mercado Libre
3003 Nações Unidas Ave., Osasco, SP 06233-903, Brazil

Daniel Negrotto

Shipping Optimization Team (SOT) - Mercado Libre
4789 Posta St., Buenos Aires, C1430EKH, Argentina

ABSTRACT

This work formalizes a special case of the strongly NP-Hard problem Precedence Constrained Knapsack Problem (PCKP). In this special case, the precedence constraints are restricted to paths within the same class of items, and all items have weight equal to one. This special case is named as Path-Precedence Constrained Knapsack Problem with Unit Weights (PPCKP-UW) and it has real-world applications, including a demand in logistics for solving it in real-time. This work proves that PPCKP-UW is solvable in polynomial-time, compares the computational performance of solving it using seven different approaches, including integer linear formulations, algorithms based on dynamic programming and shortest path, and applying pre/post-processing tasks to the instances. Ten new instances were generated and shared with the source codes, allowing the reproduction of all experiments made. Experimental results showed that the proposed approaches run faster than the conventional formulation based on PCKP.

KEYWORDS. Knapsack Problem. Exact algorithms. Integer Programming.

Logistics and Transportation

1. Introduction

In this work, we study a special case of the Precedence Constrained Knapsack Problem (PCKP). Considering a knapsack with a capacity limit and items of unit weight, these items are grouped into classes and subjected to precedence restrictions, which can be viewed as a dependency path for each class of items. We call this version the Path-Precedence Constrained Knapsack Problem with Unit Weights (PPCKP-UW). Note that the precedence relationships between items in each class can be represented by means of a directed acyclic graph, where nodes correspond to items in a one-to-one way.

The Figure 1 illustrates the items of an instance for PPCKP-UW. Each row indicates a class of items. The arrows indicate the dependency between items of the same class, forming a path. As examples, it is only allowed to select the item number 5 if there is available capacity in the knapsack and the items before it but in this same class (items number 4 and 3) are selected; however, items number 1, 3 and 8 may be selected regardless of other items if there is available capacity in the knapsack.

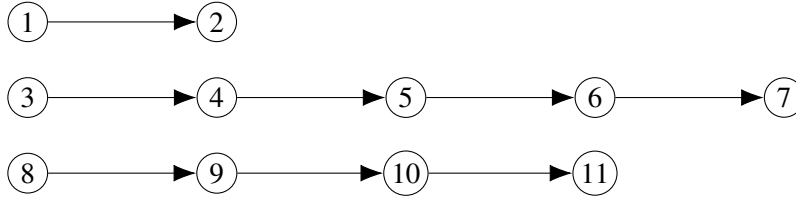


Figure 1: Example of an instance for PPCKP-UW with 3 classes.

More formally, to define the input of the PPCKP-UW, let us consider the input tuple $\vartheta = (\mathbb{C}, W, p, w)$ where, $\mathbb{C} = \{\mathbb{G}_1(\mathbb{V}_1, \mathbb{E}_1), \dots, \mathbb{G}_n(\mathbb{V}_n, \mathbb{E}_n)\}$ is the set of n directed acyclic graph that represents precedence restrictions of each of the n classes, W is the maximum capacity of the knapsack and p, w are functions that return the profits and unit weights of each item. Thus, the 0-1 programming problem formulation, which was presented by [Nemhauser and Wolsey, 1993] to solve the PCKP, can be easily modified to solve PPCKP-UW. It is only necessary to add a set of new indices that represent which class each item is in. The PPCKP-UW standard formulation is defined by the equations from 1 to 5. Let $x_{c,i}$ be a variable such that

$$x_{c,i} = \begin{cases} 1, & \text{if item } i \text{ of class } c \text{ is put in knapsack,} \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

$$\max \sum_{c \in \mathbb{C}} \sum_{i \in \mathbb{V}_c} p_{c,i} x_{c,i} \quad (2)$$

$$\text{s.a: } \sum_{c \in \mathbb{C}} \sum_{i \in \mathbb{V}_c} w_{c,i} x_{c,i} \leq W \quad (3)$$

$$x_{c,i} \leq x_{c,j}, \forall c \in \mathbb{C}, \forall (i, j) \in \mathbb{E}_c \quad (4)$$

$$x_{c,i} \in \{0, 1\}, \forall c \in \mathbb{C}, \forall i \in \mathbb{V}_c \quad (5)$$

Constraints (3) Limits the choice of items according to available capacity and constraints(4) ensures the precedence rules for each item in each class. Finally, constraints (5) determine that the variables are binary and the objective function (2) maximizes the total profit.

Knapsack problems have applications in several areas. Grouping items into classes and applying specific precedence relationships to each class arise naturally as a consequence of logistics/physical requirements among items in industry. For example, in open-pit mining problems, we can remove a block only when all the blocks lying immediately above have been removed [Moreno et al., 2010]. For the transportation programming problem, choosing items from different classes could include a transit and a non-transit alternative to the same congestion problem [Zhong and Young, 2010].

As mentioned, the PPCKP-UW can be seen as a special case of PCKP when the number of classes is equal to one. Many techniques based on integer linear programming have been proposed for solving the PCKP (see Boyd [1993]; Park and Sungsoo [1997]; Van de Leensel et al. [1999]). Samphaiboon and Yamada [2000] presented dynamic programming algorithms that, combined with a method to reduce the size of the problem, are able to solve PCKPs with up to 2,000 items spending less than a few minutes of CPU time.

An important subclass of PCKP is the Tree-Knapsack Problem (TKP), where the instance's graph is a directed tree rooted. In the TKP, all the items are subjected to a partial ordering represented the instance's graph. If a node is selected to be packed into the knapsack, then all the items on the path from the selected node to the root must also be packed. Shaw and Cho [1998] presented an upper bound for the TKP that can be obtained from the lagrangian dual generated from the integer linear programming model defined in its work. Based on this fact, the authors also presented a branch-and-bound procedure. Some dynamic programming algorithms to solve this problem are presented by Johnson and Niemi [1983]; Cho and Shaw [1997]. Note that, the PPCKP-UW can also be seen as a TKP, considering a forest as input, where each tree represents a class of items.

The PPCKP-UW can also be considered a special case of the Multiple-Choice Knapsack Problem (MCKP) where a set of items is partitioned into classes, being possible only selection of exactly one item out of each class. Like other Knapsack Problem variants, MCKP can be solved using two approaches: the use of branch-bound methods as presented by Dyer et al. [1984]; Kozanidis and Melachrinoudis [2004] or dynamic programming algorithms [see Pisinger [1995]; Dyer et al. [1995]]. It is important to mention that the dynamic programming algorithms presented in literature have an integer requirement (either the project cost or project profit must be an integer). The same assumption is used in our work.

To solve the PPCKP-UW as a special case of MCKP, we can easily modify the standard formulation presented previously, considering the same indices, constants and variables with the following changes:

$$\max \sum_{c \in \mathbb{C}} \sum_{i \in \mathbb{V}_c} p_{c,i} x_{c,i} \quad (6)$$

$$\text{s.a: } \sum_{c \in \mathbb{C}} \sum_{i \in \mathbb{V}_c} i x_{c,i} \leq W \quad (7)$$

$$\sum_{i \in \mathbb{V}_c} x_{c,i} \leq 1, \forall c \in \mathbb{C}, \forall c \in \mathbb{C} \quad (8)$$

$$x_{c,i} \in \{0, 1\}, \forall c \in \mathbb{C}, \forall i \in \mathbb{V}_c \quad (9)$$

Constraints (7) limits the choice of items to the available quantity because in this version the quantity of items is accumulated and constraints (8) limits selection to only one number of items per class (Multiple-Choice Knapsack). Finally, constraints (9) determine that the variables are binary and the objective function (6) maximizes the total profit.

Both of these formulations, based on the PCKP and the MCKP, do not have a matrix of constraints that fulfills the conditions to be Totally Unimodular [Wolsey, 1998]. Therefore, it is not guaranteed that solving the PPCKP-UW using such formulations will be performed in polynomial-time. This may make the formulation impractical to be used for large instances, not allowing its use in large-scale simulations as those made in [Monteiro and Davis, 2023] for transportation problems.

To the best of our knowledge, the PPCKP-UW has not been formally presented in the literature, but it has impactful real-world applications. In the company where the authors work, that belongs to the logistic sector, this problem is solved frequently and currently has a run time limit of only 200 milliseconds. Therefore, this work aims to present and discuss this problem in the operational research scientific community, sharing our results and inviting other researchers to advance knowledge in this topic. By doing so, the whole community may benefit from modeling the PPCKP-UW into different applications and by finding more effective methods to solve it. This will allow us to approach this problem into scenarios of even more restricted run time limit, computational resources, or with larger instances.

In this work, we presented integer linear formulations and exact algorithms to solve PPCKP-UW, including an algorithm based on dynamic programming that is solvable in polynomial-time. Thus, although the PCKP is strongly NP-Hard, we showed that the PPCKP-UW is in $\mathbb{P}$. We proposed a set of new instances that simulate real logistical situations and run computational experiments using pre/post-processing tasks. This work is organized as follows: Section 2 discusses the proposed methods to solve the PPCKP-UW; Section 3 presents the simulated instances; Section 4 shows the results obtained, considerations about the time/space complexity, and the run time spent by all proposed methods; and Section 5 presents general concluding remarks and future works.

2. Proposed Methods

This section presents three proposed methods, besides the two linear formulations discussed in Section 1 for solving the PPCKP-UW. All source codes are available for download [Monteiro et al., 2024b]. The following subsection describes the dynamic programming approach.

2.1. Dynamic Programming

This approach resembles the traditional dynamic programming algorithm for solving the binary knapsack problem [Cormen et al., 2009]. The main difference is that an additional loop must be added to also check the possibilities yielded by the precedences inside the same class. The relation of recurrence for defining the array of the dynamic programming is defined by Equation 10, where $T_{i,c}$ holds the optimal value for a knapsack with up to i capacity and considering up to c classes; $P_{k,c}$ is the sum of profits up to the k item from the precedences of class c ; and W is the knapsack capacity.

$$T_{i,c} = \begin{cases} 0, & \text{if } i = 0 \text{ or } c = 0, \\ \max(T_{i-k,c-1} + P_{k,c}, T_{i,c-1}), & \forall k \mid 1 \leq k \leq \min(W, i) \text{ otherwise.} \end{cases} \quad (10)$$

The time complexity for this approach is $\Theta(\sum_{c \in \mathbb{C}} \sum_{i=1}^{\min(W, |\mathbb{V}_c|)} i)$. The Θ was used because that complexity is the same for the worst and best scenarios [Cormen et al., 2009]. Although the W is a constant, what would indicate that this algorithm runs in pseudo-polynomial time [Cormen et al., 2009], in the PPCKP-UW it is guaranteed that the capacity W will always be smaller than the total number of items. Otherwise, there would be more capacity in the knapsack than items to be selected, and thus, there would be no problem to be solved. So, the worst case in time complexity $O(\sum_{c \in \mathbb{C}} \sum_{i=1}^{\min(W, |\mathbb{V}_c|)} i)$ is smaller than or equal to $O(\sum_{c \in \mathbb{C}} \sum_{i=1}^{|\mathbb{V}_c|} i)$. Therefore, the PPCKP-UW is solved in polynomial time, differently from the PCKP that is strongly NP-Hard [Boyd, 1993].

If only the optimal objective value is needed (without needing to backtrack the optimal solution) the space complexity can be reduced to $O(W)$, since the recurrence relation only refers to the current and earlier class. However, if the backtracking of the solution is required, the space complexity increases to $O(W|C|)$ to store the number of selected items for each class. The following subsection presents the approach based on Dijkstra's Shortest Path.

2.2. Dijkstra

This algorithm solves the PPCKP-UW by representing each possible solution as vertices on a graph, and by finding the shortest path from the initial solution to an optimal one. Figure 2 illustrates an instance of two classes and allows a maximum of three items to be selected. The number of items selected from each class is represented by the elements in the sets indicated by each vertex. Therefore, the first vertex labeled as $\{0,0\}$ indicates that both classes start with zero items. After selecting an item to a class, the algorithm takes into account a weight regarding this added item. This weight is represented by $w\{1^+, 0\}$ when the selected class is the first one. The 1^+ goes in the second element when the item is selected to the second class: $w\{0, 1^+\}$.

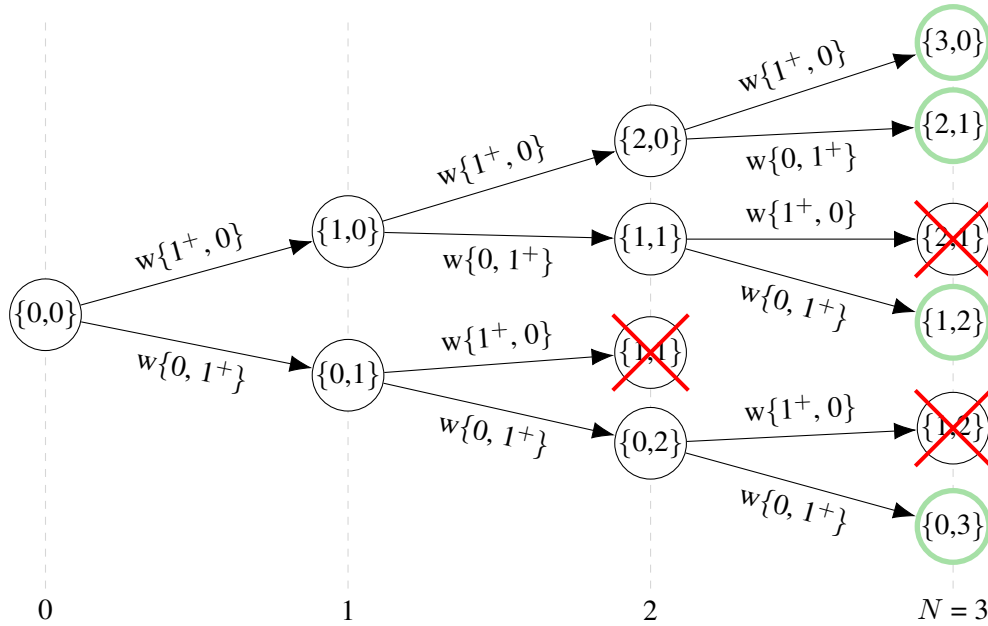


Figure 2: Example of instance for the Dijkstra method. It considers two classes and $N = 3$.

Although the PPCKP-UW has only unit weights, this algorithm considers the items' profits as weights. By doing so, it is possible to properly differ the benefits of each item without loss of generality. However, the objective function must be changed since the shortest path problem aims to minimize the sum of edges' weights in the path, instead of maximizing the sum of profits as the PPCKP-UW requires. In order to deal with the PPCKP-UW in a minimization sense, this algorithm considers the precedence of items in its reverted order. By doing so, the last profit to be achieved in the conventional PPCKP-UW (got by selecting the last possible item in the precedence of a class) is represented in this algorithm as the first item's weight of the class. Therefore, the shortest path found by this algorithm is exactly the least profitable sequence of items among the classes.

Besides reverting the precedence of items, it is also required to replace the capacity W by an "inverted capacity" N . This inverted capacity is the total of items that will not be selected because of the capacity constraint. It can be calculated by subtracting the capacity W from the total

quantity of items among all classes. Equation 11 formalizes the N value. Therefore, each step made while calculating the shortest path is actually a decision of which items should be ignored from the knapsack. And the algorithm finishes when the worse N items are selected.

$$N = \sum_{c \in \mathbb{C}} |\mathbb{V}_c| - W \quad (11)$$

As illustrated in Figure 2, it is possible that vertices indicating the same combination appear from different paths. Those repeated vertices will not provide any useful information, since they would yield symmetric branches. Those cases are pruned by the algorithm, as shown by the vertices crossed in red. The algorithm finishes when any vertex of the last layer (depicted in green) is reached. Since there will be no cycle of negative weights, it is assured that the path performed builds an optimal solution that solves the PPCKP-UW [Cormen et al., 2009; Mrena et al., 2019].

The shortest path in this algorithm is equal to Dijkstra's Algorithm for finding the Shortest Paths to all nodes of a graph, starting from a source node. In this work, it was used a Binary Heap to maintain the priority queue. Therefore, the time complexity for this implementation is $O(n \log(n) + m \log(n)) = O(m \log(n))$, and the space complexity is $O(m + n)$, where n is the number of vertices and m is the number of edges in the graph [Cormen et al., 2009; Mrena et al., 2019]. Although there are priority queues with operations having lower time complexity, such implementations are only faster than Binary Heap for instances with more than 1 million nodes [Mrena et al., 2019]. Thus, for the instances of this work, a Binary Heap is more suitable.

Although the algorithm for finding the shortest path is effective, solving the PPCKP-UW using this approach often requires walking through a large graph, even after pruning the symmetric paths. In the worst case scenario, when all possible vertices are generated, it will be required $\frac{(|\mathbb{C}|+N)!}{|\mathbb{C}|!N!}$ vertices and edges, where $|\mathbb{C}|$ is the number of classes and N is the inverted capacity of the knapsack. That will not only be time expensive to calculate, but also will be cumbersome to hold all this graph in primary memory. However, the best case scenario happens when the path performed has gone deeply in the same branch, similarly to a Depth-First Search. In this case, an optimal solution is found after only visiting $|\mathbb{C}| * N$ vertices. The following section presents the pre/post-processing task proposed.

2.3. Pre and Post Processing

Algorithm 1 abstracts the idea of the proposed pre/post processing. The rationale is to “invert” the instance when the capacity W is greater than half of the total of items. This method is similar to the approach based on Dijkstra's Shortest Path, which benefits from instances with high capacity W . By doing so, the algorithm passed as parameter will focus on finding which of the items should be ignored from the knapsack instead of finding which items should be put in the knapsack. The algorithm in the parameter can be any of the proposed ones, including the linear formulations, but except the Dijkstra (because it already inverts the instance as default).

In the best case scenario, the time complexity of this pre/post-processing is $O(|\mathbb{C}|)$ when the total of items is smaller than or equal to half of the capacity W . In this case, no further pre or post-processing is required since it will be better to run the algorithm in the parameter as it is. In the worst case scenario, the time complexity is $O(\sum_{c \in \mathbb{C}} |\mathbb{V}_c|)$ since it is required to invert the order of the precedences, as made in the Dijkstra approach, and after solving the algorithm, reverting back the instance to get the actual optimal solution. The following section presents the instances and computational resources used in the experiments.

3. Instances and Resources

Ten instances were generated to simulate real logistical situations using the following data as input parameters: *No. Class* is the number of classes generated; *Class Length* is the number

Algorithm 1 Check Reduce Solve

```

1: function CHECKREDUCESOLVE(algorithm, instance)
2:    $\mathbb{C} \leftarrow \text{instance.classes}$ 
3:    $\text{sum\_lengths} \leftarrow \sum_{c \in \mathbb{C}} |\mathbb{V}_c|$ 
4:   if  $\text{instance.W} / \text{sum\_lengths} \leq 0.5$  then
5:     return algorithm(instance)
6:   else
7:      $N \leftarrow \text{sum\_lengths} - \text{instance.W}$ 
8:      $\text{inverted\_instance} \leftarrow \text{new Instance}(\text{instance.name}, N)$ 
9:      $\text{inverted\_instance.classes} \leftarrow \text{INVERTCLASSES}(\mathbb{C}, N)$ 
10:     $\text{inverted\_solution} \leftarrow \text{algorithm}(\text{inverted\_instance})$ 
11:     $\text{reverted\_solution} \leftarrow \text{REVERTSOLUTION}(\text{inverted\_solution}, \mathbb{C})$ 
12:    return reverted_solution
13: end if

```

of items in each class; *Max. Profit* is the maximum random profit a item was allowed to have; and *Capacity (%)* that considers the knapsack capacity as a percentage of the total sum of items generated. The parameters used are described in Table 1. The first four instances had their *Class Length* and *Max. Profit* different for each class. Therefore, these instances have a class with a pattern evidently different from the others.

Table 1: Parameter values for each instance.

<i>Instance</i>	<i>Name</i>	<i>No. Class</i>	<i>Class Length</i>	<i>Max. Profit</i>	<i>Capacity (%)</i>
1	EvidentWorstClass10CapPct90	10	[200,600]	8,738*	90
2	EvidentBestClass10CapPct90	10	[200,600]	10**	90
3	EvidentWorstClass10CapPct10	10	[200,600]	8,738*	10
4	EvidentBestClass10CapPct10	10	[200,600]	10**	10
5	LowEarnClass10Len5CapPct50	10	5	10	50
6	HighEarnClass10Len5CapPct50	10	5	1,000	50
7	LowEarnClass100Len50CapPct50	100	50	10	50
8	HighEarnClass100Len50CapPct50	100	50	1,000	50
9	HighEarnClass10Len500CapPct50	10	500	1,000	50
10	HighEarnClass50Len1KCapPct50	50	1,000	1,000	50

Notes. * For class “B” the gains were set to 10. ** for class “B” the gains were set to 8,738.

However, the other six instances had the same parameter fixed for all classes. By doing so, all classes (although with different random values) follow the same patterns of maximum profit generated and quantity of items. As an example, since the last instance has 50 classes, each with 1,000 items, and its percentage capacity is 50%, the capacity W for this instance was set as $W = 50 \times 1,000 \times 0.50 = 25,000$. The rationale behind these parameter settings is to generate a diverse set of instances that may be easier or cumbersome to different algorithms. These instances and the source codes used in the experimental results are available for download [Monteiro et al., 2024b] through the link <https://data.mendeley.com/datasets/tyfkfhh8vx/2>.

The final part of the link “/2” indicates that the dataset is currently in its second version. Any update made even to the description of this dataset will imply in generating a different version, and thus, having a different ending of the link. If any problem emerge while accessing this dataset, please try another (earlier or later) version.

The experiments were run in a MacBook Pro with processor Apple M1, main memory of 16 GB and operational system macOS Sonoma 14.4.1. The evaluated approaches were implemented in Python and run using Python 3.12.3, the library for modeling the linear formulation and calling the solver was PuLP 2.8.0, and the solver used was the CBC MILP Solver 2.10.3. All runs of the solver did not print the log messages, to avoid making the run time higher than the actual solving time. The following section presents and discusses the experiments. The discussion of experiments also considered statistical tests, namely Shapiro-Wilk test for normality, ANOVA and Mann Whitney U. All these tests were performed using the SciPy library version 1.13.1.

4. Experiments and Analysis

The experiments consisted in running 50 times seven approaches for solving the PPCKP-UW. These seven approaches and their names referenced in the results were:

- *PCK*: representing the formulation based on PCKP and described from Equation 3 to 5;
- *MCK*: the formulation based on MCKP and described from Equation 7 to 9;
- *DP*: the algorithm that used dynamic programming, described in Subsection 2.1;
- each of the above approaches but with a “-*P*” in the end, namely *PCK-P*, *MCK-P* and *DP-P*, indicating the pre/post-processing tasks described in Subsection 2.3;
- *Dijkstra*: the algorithm based on Dijkstra’s Shortest Path, described in Subsection 2.2;

Experiments that took more than 10 minutes for solving an instance were canceled. That happened to experiments running the approach based on Dijkstra for all instances except for the 1, 5 and 6. This longer time required was expected given the high time complexity in the worst case of the Dijkstra approach. Table 2 presents the mean of the time required to run each experiment, and Table 3 presents the coefficient of variation for the time runs. The best running time for each experiment is written in bold and algorithms with more than 10 minutes running appear as N/A.

<i>Instance</i>	<i>PCK</i>	<i>P-PCK</i>	<i>MCK</i>	<i>P-MCK</i>	<i>DP</i>	<i>P-DP</i>	<i>Dijkstra</i>
1	0.741	0.245	0.313	0.172	7.118	0.195	0.033
2	4.166	3.224	0.440	0.392	7.487	0.200	N/A
3	2.693	2.695	0.303	0.311	0.199	0.200	N/A
4	0.383	0.382	0.202	0.200	0.195	0.193	N/A
5	0.045	0.045	0.046	0.047	0.001	0.001	92.262
6	0.052	0.052	0.052	0.055	0.001	0.001	143.686
7	3.028	3.039	0.450	0.451	5.211	5.233	N/A
8	3.081	3.090	0.502	0.508	5.204	5.251	N/A
9	3.323	3.332	0.477	0.483	5.484	5.544	N/A
10	163.379	163.302	4.055	4.147	526.472	528.395	N/A

Table 2: Mean of the run time (in seconds) for each algorithm and instance. The fastest algorithm for each instance is written in bold.

The coefficient of variation was chosen to be presented in Table 3 instead of the standard deviation due to its properties of neither depending on an unit of measurement nor scale [Santos and Dias, 2021]. Therefore, even when comparing random variables of the same unit of measurement, the coefficient of variation show how the experiments varied compared to its mean, not being biased by different scales such as the Dijkstra run times of instances 5 and 6 that spent seconds of time run instead of milliseconds as spent by the other algorithms.

<i>Instance</i>	<i>PCK</i>	<i>P-PCK</i>	<i>MCK</i>	<i>P-MCK</i>	<i>DP</i>	<i>P-DP</i>	<i>Dijkstra</i>
1	2.810	7.127	4.501	10.871	0.506	1.024	33.867
2	0.743	0.458	3.174	14.164	0.148	0.178	N/A
3	0.686	0.914	4.363	22.718	0.147	1.611	N/A
4	4.090	2.798	5.575	10.240	0.468	0.665	N/A
5	5.737	9.929	7.259	6.258	0.656	3.513	0.295
6	1.138	1.561	0.904	12.651	1.571	4.701	0.513
7	0.783	0.681	3.874	4.397	0.078	0.236	N/A
8	0.616	0.541	3.340	3.877	0.065	1.057	N/A
9	0.672	0.552	3.328	3.680	0.065	1.258	N/A
10	0.370	0.299	0.550	1.256	0.234	0.193	N/A

Table 3: Coefficient of variation of the run time for each algorithm and instance. The lowest variation for each instance is written in bold.

In general, the algorithms run time had a low dispersion along their mean, as can be seen in the Table 3 and Figure 3. The algorithms' performance is highly influenced by which instance they are solving. Among the evaluated scenarios, the most extreme cases of performance variation through instances are the *Dijkstra* with be best run time for instance one, but the worst run time for every other instance; and the *P-DP* with good run times from instance one to six, but bad performances for the other instances. The good results from *Dijkstra* on instance 1 is due to the low *Max. Profit* for class "B" and high *Capacity (%)*. Therefore, the algorithm performed similarly to a Depth-First Search, as expected as its best case scenario of time complexity mentioned in Subsection 2.2. Regarding the *P-DP*, the instances from 7 to 10 have their *Capacity (%)* set to 50. Thus, the pre/post-processing tasks described in Subsection 2.3 could not reduce the instances' size.

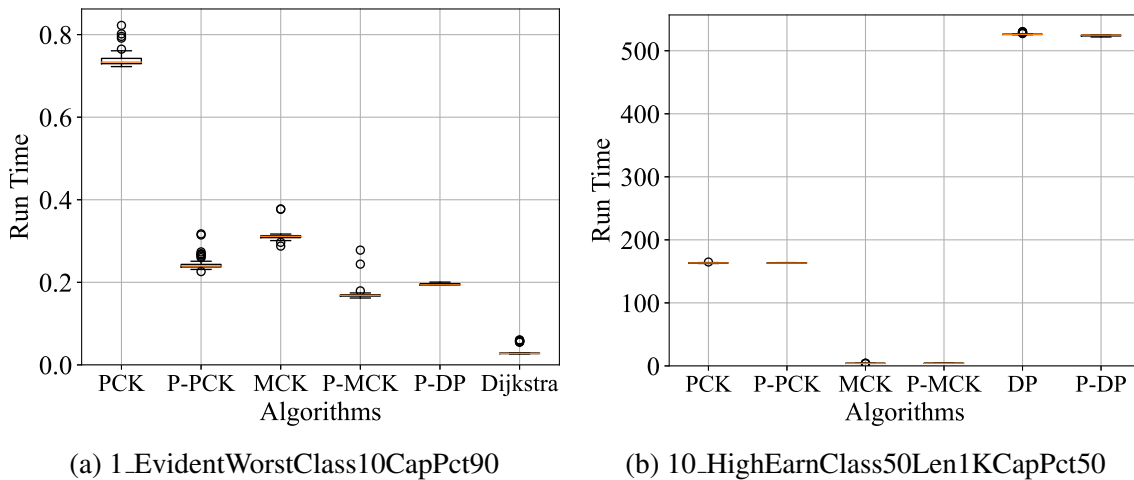


Figure 3: Run time (in seconds) of the first and last instances defined.

Although it is possible to affirm that there is significant difference between the distribution of two boxplots with boxes that does not overlap and their data follows a normal distribution [Krzywinski and Altman, 2014], not all sets of experiments followed a normal distribution according with the Shapiro-Wilk. Therefore, the analysis of significant differences made followed the rules defined in García et al. [2010] and presented by Rodríguez-Fdez et al. [2015]. According to these rules,

analysis such as from this work should perform the ANOVA test if the data fulfills the ANOVA's conditions (including normality), and perform the Mann Whitney U test otherwise.

For every instance, those statistical tests were applied to all pairs of run time sets. Most of the pairs got a p-value lower than 0.05, indicating that there is significant difference between the sets of data. The pairs that do not have such low p-value are presented in the Table 4. All scenarios with $p\text{-value} \geq 0.05$ were calculated by the non-parametric test Mann-Whitney U. Most of these tests happened for instances 3 and 4, which have the lowest *Capacity (%)* among all instances. In other words, those instances have a low capacity W compared with the available number of items.

<i>Instance</i>	<i>Approach 1</i>	<i>Approach 2</i>	<i>p-value</i>
3	<i>PCK</i>	<i>P-PCK</i>	0.608
4	<i>PCK</i>	<i>P-PCK</i>	0.844
4	<i>P-MCK</i>	<i>DP</i>	0.746
4	<i>P-MCK</i>	<i>P-DP</i>	0.089
6	<i>P-PCK</i>	<i>MCK</i>	0.381
7	<i>MCK</i>	<i>P-MCK</i>	0.780

Table 4: Experiments not significantly different, having $p\text{-value} \geq 0.05$.

Those low capacities influenced such results in two ways:

1. it simplified the instances, making their run times similar;
2. since the capacity proportion is lower than or equal to 50%, the pre/post-processing is waived because it will be better not to convert the instance and solve the problem as it is.

None of the scenarios with $p\text{-value} \geq 0.05$ happened to instances having *Capacity (%)* lower than 50%, so that would be worthy of running the pre/post-processing. Besides, three scenarios were about comparing approaches with and without the pre/post-processing task, namely the scenario from instance 3, the first scenario from instance 4 and the scenario from instance 7. Therefore, the pre/post-processing task influences the run time when this task is useful, and the pre/post-processing may not influence the run time when it is not useful. This can be seen on Table 2, since the approaches with pre/post-processing compared to the same approach but without the pre/post-processing had more impact by reducing the run times when *Capacity (%)* was lower than 50% (instances 1 and 2), than by changing the run times when *Capacity (%)* was greater than or equal to 50% (all other instances).

Regarding the largest instance, namely 10, the fastest approach for solving it was the *MCK*. This shows the impacts of applying a formulation different from the conventional from the PCKP, here namely as *PCK*, that could benefit from the reduction in the number of constraints. In this case, the reduction in the number of constraints was more impactful than having a matrix of constraints with coefficients higher than one, as defined by Equation 7 of the *MCK*, which could have hampered the performance of the solver by placing the linear relaxation more distant than an optimal integer solution [Monteiro et al., 2024a]. The following section concludes this work.

5. Conclusion

This paper presented the PPCKP-UW, shared ten instances and source codes for solving this problem, proposed and analyzed the time and space complexity of seven approaches to solve it, compared the time spent by solving the ten instances with the proposed approaches, proved the effectiveness of the proposed methods in finding the optimal solution, and performed statistical analysis over the experiments.

The approach that would be a straightforward approach for solving the PPCKP-UW, namely *PCK*, did not get the fastest mean run time for any of the instances. Regarding the largest instance, namely 10, the best approach was the linear formulation based on a different version of the knapsack problem, namely MCKP. This formulation got good results even without the pre/post-processing tasks proposed. However, this pre/post-processing task was essential to make the dynamic programming algorithm one of the fastest approaches for instances with high capacity.

Results evidenced the effectiveness of the proposed approaches for instances with different characteristics. An example of that is the algorithm based on *Dijkstra*, which although was the worst approach for nine of the proposed instances, for the first instance the mean of its run times was at least five times faster than any other approach. This indicates the variety of future works that may be done by reusing the codes and data set shared by this work. As future works we suggest:

- Reduce the space complexity for backtracking the optimal solution of the dynamic programming approach;
- Propose other exact algorithms with lower time and space complexity;
- Propose a linear formulation for the PPCKP-UW that is Totally Unimodular;
- Evaluate the performance of different programming languages and compiler parameters;
- Propose other pre/post processing tasks, mainly for the algorithm based on *Dijkstra*;
- Evaluate the benefits of preprocessing tasks made by the solver.

References

- Boyd, E. A. (1993). Polyhedral results for the precedence-constrained knapsack problem. *Discrete Applied Mathematics*, 41(3):185–201.
- Cho, G. and Shaw, D. X. (1997). A depth-first dynamic programming algorithm for the tree knapsack problem. *INFORMS Journal on Computing*, 9(4):431–438.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press, 3 edition.
- Dyer, M., Kayal, N., and Walker, J. (1984). A branch and bound algorithm for solving the multiple-choice knapsack problem. *Journal of computational and applied mathematics*, 11(2):231–249.
- Dyer, M., Riha, W., and Walker, J. (1995). A hybrid dynamic programming/branch-and-bound algorithm for the multiple-choice knapsack problem. *Journal of Computational and Applied Mathematics*, 58(1):43–54.
- García, S., Fernández, A., Luengo, J., and Herrera, F. (2010). Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: Experimental analysis of power. *Information sciences*, 180(10):2044–2064.
- Johnson, D. S. and Niemi, K. (1983). On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8(1):1–14.
- Kozanidis, G. and Melachrinoudis, E. (2004). A branch & bound algorithm for the 0-1 mixed integer knapsack problem with linear multiple choice constraints. *Computers & Operations Research*, 31(5):695–711.

- Krzywinski, M. and Altman, N. (2014). Points of Significance: Visualizing samples with box plots. *Nature Methods*, 11(2):119–120.
- Monteiro, C. M. and Davis, C. A. (2023). Polynomial-time carsharing optimization: Linear formulation and large-scale simulations. *IEEE Transactions on Intelligent Transportation Systems*, 24(4):4428–4437.
- Monteiro, C. M., dos Santos, V. F., and Davis, C. A. (2024a). Computational complexity and linear formulations for optimizing the location of shared mobility stations. *International Transactions in Operational Research*, 31(5):2903–2929.
- Monteiro, C. M., Figueiredo, T. F., del Priore, J., Xavier, E. C., Negrotto, D., and Shipping Operational Research Team, S. (2024b). Instances and Code of the Paper 'The Path-Precedence Constrained Knapsack Problem with Unit Weights: Exact Algorithms, Linear Formulations and Instances'. <https://data.mendeley.com/datasets/tyfkfh8vx/2>.
- Moreno, E., Espinoza, D., and Goycoolea, M. (2010). Large-scale multi-period precedence constrained knapsack problem: a mining application. *Electronic notes in discrete mathematics*, 36: 407–414.
- Mrena, M., Sedlacek, P., and Kvassay, M. (2019). Practical applicability of advanced implementations of priority queues in finding shortest paths. In *2019 international conference on information and digital technologies (IDT)*, p. 335–344. IEEE.
- Nemhauser, D. B. G. and Wolsey, L. (1993). Integer programming and combinatorial optimization.
- Park, K. and Sungsoo, P. (1997). Lifting cover inequalities for the precedence-constrained knapsack problem. *Discrete Applied Mathematics*, 72(3):219–241.
- Pisinger, D. (1995). A minimal algorithm for the multiple-choice knapsack problem. *European Journal of Operational Research*, 83(2):394–410.
- Rodríguez-Fdez, I., Canosa, A., Mucientes, M., and Bugarín, A. (2015). STAC: a web platform for the comparison of algorithms using statistical tests. In *2015 IEEE international conference on fuzzy systems (FUZZ-IEEE)*, p. 1–8. IEEE.
- Samphaiboon, N. and Yamada, Y. (2000). Heuristic and exact algorithms for the precedence-constrained knapsack problem. *Journal of optimization theory and applications*, 105:659–676.
- Santos, C. and Dias, C. (2021). Note on the coefficient of variation properties. *Brazilian Electronic Journal of Mathematics*, 2(4):101–11.
- Shaw, D. X. and Cho, G. (1998). The critical-item, upper bounds, and a branch-and-bound algorithm for the tree knapsack problem. *Networks: An International Journal*, 31(4):205–216.
- Van de Leensel, R. L., Van Hoesel, C., and Van de Klundert, J. (1999). Lifting valid inequalities for the precedence constrained knapsack problem. *Mathematical programming*, 86:161–185.
- Wolsey, L. A. (1998). *Integer Programming*. John Wiley & Sons, 1 edition.
- Zhong, T. and Young, R. (2010). Multiple choice knapsack problem: Example of planning choice in transportation. *Evaluation and program planning*, 33(2):128–137.