POLITECNICO DI TORINO

Master's Degree in Mathematical Engineering

Master's Degree Thesis

Recovering Beam Search for the 0-1 Knapsack Problem with Forfeits

Supervisor **Candidate**

Prof. Federico DELLA CROCE **Ghassane BEN EL AATTAR**

DI DOJOLA

October 2024

Summary

The 0-1 Knapsack Problem with Forfeits is a variant of the Knapsack Problem where, given a set of items, we want to choose a subset of them such that the total sum of their values minus the sum of forfeits induced by choosing specific pairs of items is maximized. As in the regular Knapsack Problem, the sum of weights of the subset of items selected must be less or equal to a specified value (the budget).

In this thesis, we present a heuristic approach for this problem, which is based on the Recovering Beam Search algorithm and Dynamic Programming.

Our goal was to develop an algorithm that could attain better results, given a time limit, than a well-known commercial solver for mathematical optimization problems, for specified instances and a large number of items.

Specifically, given a time limit, we wanted our heuristic to return a value of the objective function closer to the optimal value than the one given by the solver used as a reference.

Contents

1 Introduction **7**

1.1 Combinatorial Optimization 7

1.2 Computational Complexity 8

1.3 Linear Programming 13

2 The Knapsack Problem **15**

2.1 Linear Programming Model 15

2.2 Decision Version 16

2.3 Branch and Bound 18

2.3.1 Branch and Bound Algorithm 18

2.3.2 Branching 19

2.3.3 Binary Branching 19

2.3.4 N-ary Branching 20

2.3.5 Bounding 20

2.3.6 Branch and Bound for the 0-1 Knapsack Problem . . . 22

2.4 Dynamic Programming 25

2.4.1 Stages 25

2.4.2 States 26

2.4.3 Recursive Optimization 26

2.4.4 Dynamic Programming for the 0-1 Knapsack Problem . 27

3 0-1 Knapsack Problem with Forfeits **33**

3.1 Mathematical Formulation 33

3.2 Literature 34

3.2.1 GreedyForfeits 35

3.2.2 CarouselForfeits 38

4 Recovering Beam Search 41

4.1 Description of the Procedure 42

5 Recovering Beam Search for the 0-1 Knapsack Problem with Forfeits 47

5.1 Data Representation 47

5.2 Order of the Items 48

5.3 Upper Bound Computation 49

5.4 Lower Bound Computation 50

5.5 Recovering Beam Search 57

5.5.1 Branching and Solutions Generation 57

5.5.2 Recovering Step 57

5.5.3 Filtering Solutions 61

5.5.4 Inserting new Partial Solutions 64

5.5.5 Overall Time and Memory Complexity 64

6 Computational Results 66

6.1 Description of Used Instances and Environment 66

6.2 General Results 67

6.3 Choice of Beam Size 68

6.4 Results for Bigger Time Limits 69

7 Conclusions 70

7.1 Our Solution for the 0-1 Knapsack Problem with Forfeits . . . 70

7.2 Improving the Upper Bound for each Partial Solution 70

List of Algorithms

1 KNAPSACKDPTOPSORT 29
2 KNAPSACK 30
3 GreedyForfeits - Part I 37
4 GreedyForfeits - Part II 38
5 CarouselForfeits 39
6 RBS method-Part I (beam width = w, search tree depth = u) 45
7 RBS method-Part II (continued from the previous page) . . . 46
8 Building Dynamic Programming Table for 0-1 Knapsack . . . 51
9 GetOrder Function 51
10 Update Active Variables Function 53
11 Get Lower Bound Function (Part I) 54
12 Get Lower Bound Function (Part II) 55
13 Get Lower Bound Function (Part III) 56
14 Recovering Beam Search - Part I 59
15 Recovering Beam Search - Part II 60
16 Solutions Filtering and Evaluation Function Computation - Part I 62
17 Solutions Filtering and Evaluation Function Computation - Part II 63
18 Updating Solutions 65

List of Figures

1 Computational Complexity Classes [**?**] 12

2 Types of Branching 19

3 Example of a 0-1 Knapsack Problem 24

4 Example of a search tree in a Branch and Bound algorithm for the 0-1 Knapsack problem described before 24

5 Example of State Tree for the Knapsack Problem [**?**] 28

6 A compact representation of an instance of the Knapsack problem. 31

7 An example of a dynamic programming table/array for a 0-1 Knapsack Problem. 32

List of Tables

1 Results Comparison for n = 5000 67

2 Results Comparison for n = 10000 68

3 Beam Tests 68

4 Results for Larger Time Limits - I - Instance n_{10000_6} 69

5 Results for Larger Time Limits - II - Instance n_{10000_7} 69

1 Introduction

The Knapsack Problem is one of the well-known problems in the field of combinatorial optimization [1] [2], studied for over a century. It is a problem with a very intuitive construction, but it has numerous applications in various fields such as, for instance, cryptography [3].

There are several variants of this problem, such as the 0-1 Knapsack Problem, the Multidimensional Knapsack Problem, and others.

In this thesis, our goal is to develop a solution for the 0-1 Knapsack Problem with Forfeits (or Penalties) [4]. This problem is a particular case of the 0-1 Knapsack Problem in which there is a reduction in profit when certain pairs of items are taken into the knapsack.

This problem is difficult to solve, as it belongs to the class of NP-Hard problems.

A heuristic solution based on Recovering Beam Search will be presented, with an approach to estimate an upper bound at each explored node based on Dynamic Programming.

Initially, a description of the relevant theory to the algorithm will be provided, followed by a detailed description of the problem, and finally, our solution will be presented, including computational results.

1.1 Combinatorial Optimization

Combinatorial optimization is the process of searching for maximums (or minimums) of an objective function F whose domain is a discrete but extensive configuration space (unlike an N-dimensional continuous space). Some

simple examples of typical combinatorial optimization problems are:

- **The Traveling Salesman Problem**: Given the positions (x, y) of N different cities, find the shortest possible path that visits each city exactly once.

- **Bin-Packing**: Given a set of N objects, each with a specified size s_i, pack them into the fewest number of containers (each of size B).

- **Integer Linear Programming**: Maximize a specified linear combination of a set of integers $X_1, \ldots, X_N$ subject to a set of linear constraints, each of the forms $a_1 X_1 + \ldots + a_N X_N \leq c$.

- **Job-shop Scheduling**: Given a set of jobs to be performed, and a limited set of tools with which these jobs can be done, find a schedule for which jobs should be done when and with which tools that minimizes the total time until all jobs are completed.

- **Boolean Satisfiability**: Assign values to a set of Boolean variables in order to satisfy a given Boolean expression.

1.2 Computational Complexity

Computational Complexity Theory [5] is a branch of computer science and mathematics that focuses on classifying computational problems according to their intrinsic difficulty and on the relationship between these classes. A computational problem is understood as a task solved by a computer. The formulation of a specific problem in general terms specifies the desired output

for a given input.

Definition 1.1 (Computational Problem):

A computational problem can be defined as a tuple $P = (I, O, s, m)$, where:

- I is the set of all possible inputs,
- O is the set of all possible outputs,
- $s : I \to O$ is the problem specification, which maps each input to an output,
- $m : I \to N$ is the measure function, which assigns a size to each input.

Definition 1.2 (Decision Problem):

A Decision Problem is a problem that can be posed as a yes-no question regarding input values.

The various classes defined in computational complexity theory refer to decision problems. Every optimization problem can be expressed as a decision problem by adding an appropriate bound.

Definition 1.3 (Computational Complexity):

We define the computational complexity of an algorithm as a function that maps each instance to the execution time of the algorithm required to solve that specific instance of the problem. We denote it as $f(n)$, where n is the size of the problem instance.

Computational complexity is generally indicated using the Big O asymptotic notation. We say that:

$$f(n) \in O(g(n)) \text{ if there exist } n_0, c > 0 \text{ such that } f(n) \leq c \cdot g(n), \forall n \geq n_0.$$

In particular, if $g(n)$ is a polynomial, we say that the algorithm has polynomial complexity. Otherwise, if the algorithm is $O(2^n)$ or $O(n!)$, we say that the algorithm has non-polynomial complexity. This is an important distinction used to divide problems into different classes. The computational complexity of a problem is defined by the complexity of the best algorithm capable of solving it.

Definition 1.4 (Class P):

The class P (Polynomial Time) includes problems that can be solved by a deterministic Turing machine in polynomial time. Formally, a problem P is in P if there exists an algorithm A and a polynomial p such that for every input $x \in I$, A solves P in at most $p(|x|)$ steps, where $|x|$ denotes the size of the input.

Definition 1.5 (Class NP):

The class NP (Nondeterministic Polynomial Time) includes problems for which a proposed solution can be verified in polynomial time by a deterministic Turing machine. Formally, a problem P is in NP if there exists a verifier algorithm V and a polynomial q such that for every input x and proposed solution y, V verifies whether y is a correct solution for x in at most $q(|x|)$ steps.

The NP class includes problems solvable in polynomial time by a nondeterministic Turing machine.

Definition 1.6 (NP-Completeness):

A problem P is NP-complete if:

1. P is in NP,
2. Every problem in NP is polynomial-time reducible to P.

The concept of NP-completeness is used to demonstrate the difficulty of problems. If an NP-complete problem has a polynomial-time solution, then every problem in NP also has a polynomial-time solution, which effectively means that P = NP.

The P vs. NP problem is one of the most significant open problems in computational complexity theory. It asks whether every problem whose solution can be quickly verified (in NP) can also be quickly solved (in P). This question remains central to understanding the limits of what can be efficiently computed.

Definition 1.7 (NP-Hard):

A problem X is classified as NP-hard if there exists a NP-complete problem that reduces to it, but X has not been proven to be in NP. Hence, an NP-hard problem does not necessarily have to belong to NP, so it may not be verifiable in polynomial time.

Typical example of an NP-hard problem is the optimization version of an NP-complete decision problem.

Definition 1.8 (Problem Reduction):

In complexity theory, problem reduction is a method for demonstrating the difficulty of a problem by showing that another problem known as the "starting problem" can be efficiently solved through a transformation into a "target problem." In other words, if we can solve the target problem, we can also efficiently solve the starting problem. This concept is fundamental for demonstrating the complexity of various problems and for classifying them within different complexity classes, such as NP, NP-complete, or NP-hard.

By Cook's Theorem, every problem in NP can be polynomial-time reduced to a particular problem called the Boolean Satisfiability Problem (SAT).

The SAT problem consists of determining whether a Boolean formula can be satisfied by assigning true/false Boolean values to the variables.

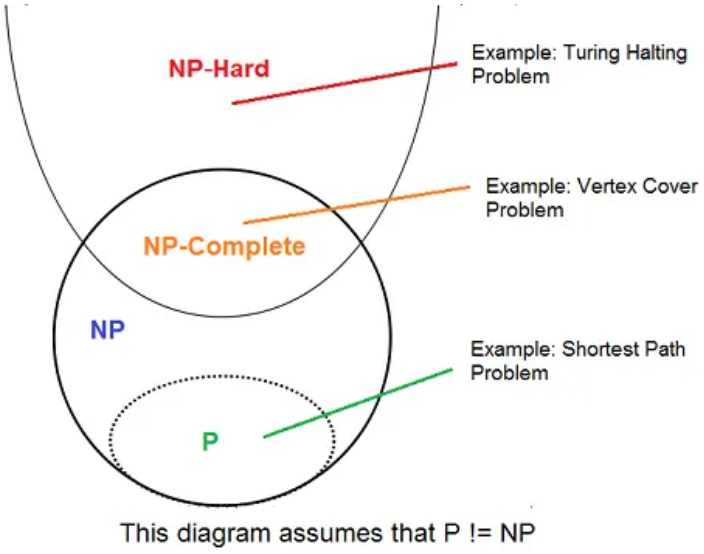

Figure 1: Computational Complexity Classes [**?**]

1.3 Linear Programming

Linear programming is a mathematical method used to solve optimization problems in which we seek to maximize or minimize a linear function subject to linear constraints. The standard form of a linear programming problem is given by:

$$\text{Maximize} \quad c^T x$$

subject to:

$$Ax \leq b$$

$$x \geq 0$$

where c is the coefficient vector of the objective function, x is the vector of decision variables, A is the coefficient matrix of the constraints, and b is the vector of constraint constants. The decision variables are often non-negative ($x \geq 0$), but they can also be free. The constraints are expressed as linear inequalities.

Variants of linear programming include integer linear programming (ILP), where decision variables are constrained to be integers, mixed-integer linear programming (MILP), where some variables are constrained to be integers while others are not, and binary linear programming (BLP), where decision variables are binary, meaning they take on only values 0 or 1.

Integer linear programming is particularly useful for modeling problems where decision variables represent discrete units, such as the number of prod-

ucts to manufacture or items to select.

Binary linear programming, on the other hand, is commonly used to represent yes/no decisions or selection among exclusive alternatives.

2 The Knapsack Problem

The Knapsack Problem is a well-known combinatorial optimization problem, which involves finding the maximum profit obtainable by choosing a subset of items, subject to the constraint on the sum of their weights being less than a predetermined value, the budget.

This problem has several variants and different formulations, depending on the context.

Some of these are:

- 0-1 Knapsack Problem: In this variant, which is the one of our interest in this thesis work, each item can be selected only once. The problem is thus to determine which items to choose to maximize profit, subject to the sum of weights constraint.
- Unbounded Knapsack Problem: It is possible to select the same item multiple times.
- Multi-dimensional Knapsack Problem: In this case, we have multiple knapsacks available.

2.1 Linear Programming Model

The 0-1 Knapsack Problem can be formulated as a linear programming model as follows. Let x_i be a binary decision variable indicating whether item i is selected ($x_i = 1$) or not ($x_i = 0$). The objective function aims to maximize the total value of selected items while respecting the knapsack capacity constraint:

$$\text{Maximize} \quad \sum_{i=1}^{n} v_i x_i$$

subject to the constraint:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$x_i \in \{0, 1\}, \quad i = 1, 2, \ldots, n$$

where n is the number of items, v_i is the value of item i, w_i is the weight of item i, and W is the knapsack capacity.

2.2 Decision Version

The decision version of the 0-1 knapsack problem: given n items with weights $w_1, w_2, \ldots, w_n$, value $v_1, v_2, \ldots, v_n$, capacity W, and value V, is there a subset $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq V$? [6]. We recall here the NP-completeness proof of the decision version of the knapsack problem [7].

Theorem 1 *The knapsack problem is NP-complete.*

Proof 1 *Firstly, the knapsack problem is in NP. The proof involves considering the set S of selected items and the verification process, which computes $\sum_{i \in S} s_i$ and $\sum_{i \in S} v_i$, which requires polynomial time with respect to the input size.*

Secondly, we show that there exists a polynomial reduction from the Partition problem to the Knapsack problem. It suffices to demonstrate the existence of a reduction $Q(\cdot)$ in polynomial time such that $Q(X)$ is a "Yes" instance for the Knapsack problem if and only if X is a "Yes" instance for the Partition problem.

Suppose we have $a_1, a_2, \ldots, a_n$ for the Partition problem, consider the following Knapsack problem: $s_i = a_i$, $v_i = a_i$ for $i = 1, \ldots, n$, $B = V = \frac{1}{2}\sum_{i=1}^{n} a_i$. $Q(\cdot)$ here is the process that converts the Partition problem into the Knapsack problem. It is evident that this process is polynomial in the input size.

If X is a "Yes" instance for the Partition problem, there exists S and T such that $\sum_{i\in S} a_i = \sum_{i\in T} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i$. Suppose our knapsack contains the items in S, then $\sum_{i\in S} s_i = \sum_{i\in S} a_i = B$ and $\sum_{i\in S} v_i = \sum_{i\in S} a_i = V$. Therefore, $Q(X)$ is a "Yes" instance for the Knapsack problem.

Conversely, if $Q(X)$ is a "Yes" instance for the Knapsack problem, with the selected set S, consider $T = \{1, 2, \ldots, n\} - S$. We have $\sum_{i\in S} s_i = \sum_{i\in S} a_i \leq B = \frac{1}{2}\sum_{i=1}^{n} a_i$ and $\sum_{i\in S} v_i = \sum_{i\in S} a_i \geq V = \frac{1}{2}\sum_{i=1}^{n} a_i$. This implies that $\sum_{i\in S} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i$ and $\sum_{i\in T} a_i = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i$. Therefore, $\{S, T\}$ is the desired partition and X is a "Yes" instance for the Partition problem.

This confirms the NP-completeness of the Knapsack problem.

2.3 Branch and Bound

Discrete optimization problems in general are very difficult to solve because the number of solutions grows exponentially with the number of variables and tools from differential calculus, such as derivatives (useful for characterizing optimal points), are not available.

Due to the combinatorial explosion of the number of solutions, explicit enumeration is not feasible.

However, there are techniques for implicit enumeration such as:

- Branch and Bound [8]
- Dynamic Programming [9]

2.3.1 Branch and Bound Algorithm

In a branch-and-bound algorithm, a difficult problem P is recursively decomposed into multiple easier sub-problems $F_1, F_2, \ldots, F_n$. The decomposition (branching) must satisfy the following condition to ensure the correctness of the algorithm:

$$\chi(P) = \bigcup_{i=1}^{n} \chi(F_i)$$

The optimal solution of P is determined by comparing the optimal solutions of the sub-problems originated from it. In case of minimization:

$$z^*(P) = \min_{i=1,\ldots,n} \{z^*(F_i)\}$$

The recursive decomposition of problems into sub-problems generates a tree (also called a decision tree or search tree), in which the root corresponds to the original problem P and every other node corresponds to a sub-problem.

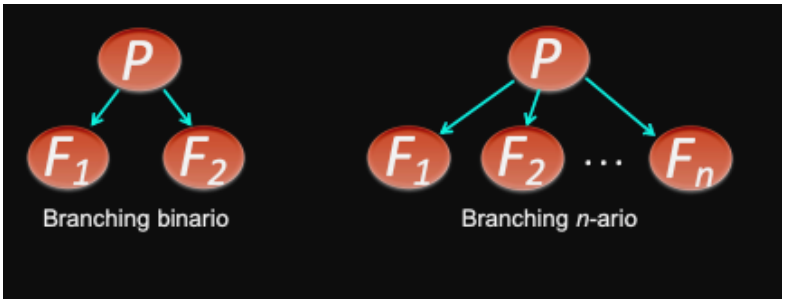

Figure 2: Types of Branching

2.3.2 Branching

For efficiency purposes, the decomposition usually involves partitioning $\chi(P)$ into disjoint subsets so that no solution needs to be (implicitly) considered more than once:

$$\chi(F_i) \cap \chi(F_j) = \emptyset \quad \forall i \neq j = 1, \ldots, n$$

There are two main ways of branching:

- Variable fixing;
- Constraint insertion.

Each sub-problem is a restriction of its predecessor and a relaxation of its successors.

2.3.3 Binary Branching

In this case, common branching rules are as follows.

- Branching on a binary variable: A binary variable x is selected. Two sub-problems are generated by setting $x = 0$ in one and $x = 1$ in the other.

- Branching on an integer constraint: A vector of integer variables $(x_1, x_2, \ldots, x_n)$, an appropriate vector of integer coefficients $(a_1, a_2, \ldots, a_n)$, and an appropriate integer constant term k are chosen. Two sub-problems are generated by inserting the constraints $ax \leq k$ in one and $ax \geq k+1$ in the other.

2.3.4 N-ary Branching

Rules for n-ary branching are as follows.

- Branching on an integer variable: An integer variable $x \in [1, \ldots, n]$ is selected. n sub-problems are generated by fixing $x = 1, x = 2, \ldots, x = n$.

- Branching on n binary variables: A vector of n binary variables $(x_1, x_2, \ldots, x_n)$ is chosen. $n + 1$ sub-problems are generated by fixing some variables as follows (one row for each sub-problem):

$$x_1 = 1$$
$$x_1 = 0, x_2 = 1$$
$$x_1 = x_2 = 0, x_3 = 1$$
$$\ldots$$
$$x_1 = x_2 = \ldots = x_{n-1} = 0, x_n = 1$$
$$x_1 = x_2 = \ldots = x_n = 0$$

2.3.5 Bounding

Given a problem P:

$$\text{minimize } z_P(x) \quad \text{s.t. } x \in \chi_P$$

a problem R:

$$\text{minimize } z_R(x) \quad \text{s.t. } x \in \chi_R$$

is a relaxation of P if and only if the following two conditions hold:

- $\chi_P \subseteq \chi_R$
- $z_R(x) \leq z_P(x) \quad \forall x \in \chi_P$.

The optimal value of the relaxation is never worse than the optimal value of the original problem:

$$z_R^* \leq z_P^*$$

As a consequence of the relaxation definition, the following corollaries hold.

Corollary 1 *If R is infeasible, then P is also infeasible.*

Corollary 2 *If x^* is optimal for R and is feasible for P and $z_R(x^*) = z_P(x^*)$, then x^* is also optimal for P.*

Corollary 3 *If $z_R^* \geq \bar{z}$, then $z_P^* \geq \bar{z}$.*

Bounding involves associating a lower bound with each sub-problem F.

Since $z_R^* \leq z_P^*$, the optimal value of $R(F)$ (a relaxation of F) provides a lower bound for each sub-problem F:

$$z_R^*(F) \leq z_F^*$$

The lower bound is compared with an upper bound corresponding to the value $z_P(\bar{x})$ of a feasible solution $\bar{x} \in X(P)$. If the lower bound of F is not

better than the best available feasible solution (the minimum found upper bound), then F can be discarded.

The correctness of bounding is given by the concatenation of two inequalities.

- The first ensures that no solution can exist in $\chi(F)$ with a better value than $z_R^*(F)$, since $z_F^* \geq z_R^*(F)$.
- The second is $z_R^*(F) \geq z_P(\bar{x})$.

Concatenating them concludes that

$$z_F^* \geq z_R^*(F) \geq z_P(\bar{x})$$

which means that solving problem F optimally is futile because it cannot provide any solution better than the one already known, $\bar{x}$. Discarding sub-problems in a branch-and-bound algorithm is crucial for saving time and memory.

For maximization problems, the optimal value of $R(F)$ provides an upper bound for each sub-problem F and every feasible solution constitutes a lower bound.

2.3.6 Branch and Bound for the 0-1 Knapsack Problem

A branch and bound algorithm for solving the 0-1 knapsack problem proceeds as follows. Consider the problem instance represented by a set of n items, each characterized by a value v_i and a weight w_i, and a maximum weight limit W that the knapsack can hold. The objective is to maximize the total value of the items placed in the knapsack while respecting the maximum weight allowed.

Initially, create a root node representing the space of initial feasible solutions. Subsequently, perform branching on each decision variable x_i, corresponding to the decision of including or not including item i in the knapsack. At each branching, two sub-problems are generated: one where item i is included in the knapsack and one where it is not.

For each generated node, calculate an upper bound based on a relaxation function of the problem. In the case of the 0-1 knapsack, a common bound is given by the optimal solution of the continuous relaxation of the problem, where fractional items are considered.

Then, proceed to explore the search tree of sub-problems in depth, maintaining an ordered list of open nodes based on the computed bound. During exploration, cuts are made on nodes that cannot lead to an optimal solution, for example when the upper bound of the node does not exceed the best available feasible solution (largest lower bound) value.

The algorithm continues to explore the tree until there are no more open nodes. The optimal solution is then given by the best available feasible solution.

This approach guarantees the optimality of the found solution, as it exhaustively explores the entire space of feasible solutions, cutting off branches of the tree that do not lead to solutions better than those already found. The procedure is therefore guaranteed to return the optimal solution to the 0-1 knapsack problem.

$$\begin{aligned} \max\, & 16x_1 + 22x_2 + 12x_3 + 8x_4 \\ & 5x_1 + 7x_2 + 4x_3 + 3x_4 \quad \leq 14 \\ & x_1, x_2, x_3, x_4 \qquad\quad \in \{0,1\}. \end{aligned}$$

Figure 3: Example of a 0-1 Knapsack Problem

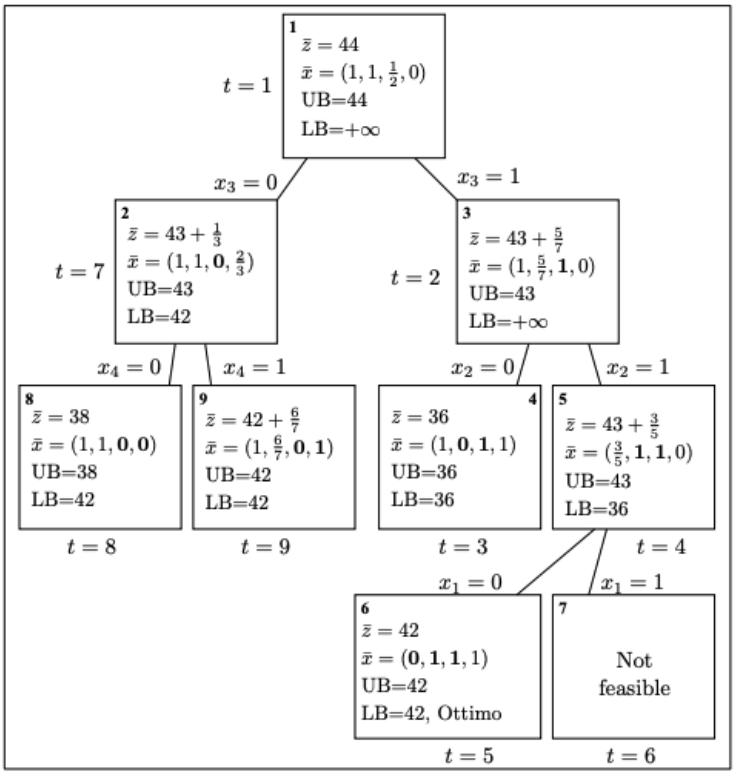

Figure 4: Example of a search tree in a Branch and Bound algorithm for the 0-1 Knapsack problem described before

2.4 Dynamic Programming

Dynamic Programming is a technique used to solve optimization problems.

- Reduces the problem's complexity, typically from $O(2^n)$ to polynomial (or pseudo-polynomial, as for the 0-1 Knapsack Problem).
- An optimization problem is broken down into subproblems, defined by variables called "states". Solutions for every subproblem is recursively calculated from previously calcualted ones, starting from the base case.

2.4.1 Stages

The essential characteristic of the dynamic programming approach is the structuring of optimization problems into multiple stages [9], which are solved sequentially one stage at a time. Although each problem of a single stage is solved like a normal optimization problem, its solution helps define the characteristics of the next single-stage problem in the sequence.

Often, stages represent different time periods in the planning horizon of the problem. For example, the problem of determining the inventory level of a single commodity can be formulated as a dynamic program. The decision variable is the amount to order at the beginning of each month; the goal is to minimize the total ordering and inventory holding costs; the basic constraint requires that the demand for the product be met. If we can only order at the beginning of each month and want an optimal ordering policy for the next year, we could decompose the problem into 12 stages, each representing the ordering decision at the beginning of the corresponding month.

2.4.2 States

Associated with each stage of the optimization problem are the states of the process. States reflect the information necessary to fully assess the consequences that the current decision has on future actions.

In the inventory problem mentioned before, each stage has only one variable describing the state: the available inventory level of the single commodity.

Specifying the states of the system is perhaps the most critical design parameter of the dynamic programming model. There are no fixed rules for doing this. In fact, for the most part, this is an art that often requires creativity and subtle intuition about the problem at hand. The essential properties that should motivate the selection of states are: i) States should convey sufficient information to make future decisions regardless of how the process has reached the current state; and ii) The number of state variables should be small, as the computational effort associated with the dynamic programming approach is prohibitively high when more than two, or possibly three, state variables are involved in formulating the model.

The latter characteristic significantly limits the applicability of dynamic programming in practice.

2.4.3 Recursive Optimization

The last general characteristic of the dynamic programming approach is the development of a recursive optimization procedure, which constructs a solution to the overall N-stage problem by first solving a single-stage problem and sequentially including one stage at a time and solving single-stage prob-

lems until the overall optimum is found. This procedure can be based on a process of backward induction, where the initial stage to be analyzed is the final stage of the problem, and problems are solved by moving back one stage at a time until all stages are included. Alternatively, the recursive procedure can be based on a process of forward induction, where the initial stage to be solved is the initial stage of the problem, and problems are solved by advancing one stage at a time until all stages are included. In certain problem contexts, only one of these induction processes may be applied (e.g., only backward induction is allowed in most problems involving uncertainties).

The basis of the recursive optimization procedure is the so-called decomposition principle of optimality, which has already been stated: an optimal policy has the property that, whatever the current state and decision, the remaining decisions must constitute an optimal policy with respect to the state resulting from the current decision.

2.4.4 Dynamic Programming for the 0-1 Knapsack Problem

Let us consider a brute force approach for this problem, where we explore the entire state tree. The edges will consist of adding or not adding a variable to the knapsack, therefore the number of nodes in the tree will be of the order of 2^n.

Let us now see how we can use dynamic programming to develop a faster solution.

A solution to an instance of the Knapsack problem will indicate which items should be added to the knapsack [10]. The solution can be broken into n true/false decisions $d_0, \ldots, d_{n-1}$. For $0 \leq i \leq n-1$, d_i indicates whether

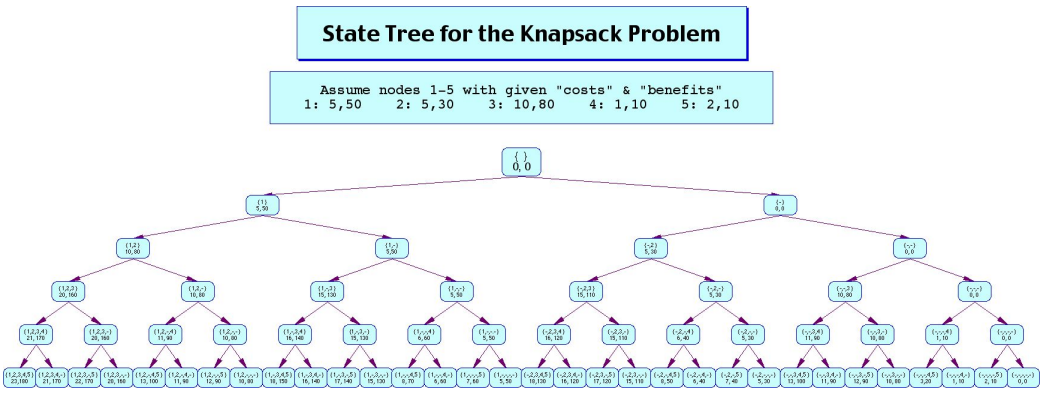

Figure 5: Example of State Tree for the Knapsack Problem [**?**]

item i will be taken into the knapsack.

In order to decide whether to add an item to the knapsack or not, we need to know if we have enough capacity left over. So the "current state" when making a decision must include the available capacity or, equivalently, the weight of the items that have already been added to the knapsack.

Let $dp[i][j]$ be the maximum value that can be obtained by using a subset of the items $i, \ldots, n-1$ (last $n-i$ items) which weighs at most j pounds. When computing $dp[i][j]$, we need to consider all the possible values of d_i (the decision at step i):

1. Add item i to the knapsack. In this case, we need to choose a subset of the items $i+1, \ldots, n-1$ that weighs at most $j - s_i$ pounds. Assuming we do that optimally, we will obtain $dp[i+1][j-s_i]$ value out of items $i+1, \ldots, n-1$, so the total value will be $v_i + dp[i+1][j-s_i]$.

2. Don't add item i to the knapsack, so we will re-use the optimal solution

for items $i+1, \ldots, n-1$ that weighs at most j pounds. That answer is in $dp[i+1][j]$.

We want to maximize our profits, so we will choose the best possible outcome.

$$dp[i][j] = \max\left(dp[i+1][j], dp[i+1][j-s_i] + v_i\right) \text{ if } j \geq s_i$$

To wrap up the loose ends, we notice that $dp[n][j] = 0, \forall 0 \leq j \leq S$ is a good base case, as the interval $n \ldots n-1$ contains no items, so there's nothing to add to the knapsack, which means the total sale value will be 0. The answer to our original problem can be found in $dp[0][S]$. The value in $dp[i][j]$ depends on values of $dp[i+1][k]$ where $k < j$, so a good topological sort would be:

$$dp[n][0]\ dp[n][1]\ \ldots\ dp[n][S]\ dp[n-1][0]\ dp[n-1][1]\ \ldots\ dp[n-1][S] \ldots dp[0][0],$$

$$, dp[0][1]\ \ldots\ dp[0][S]$$

This topological sort can be produced by the pseudo-code below.

Algorithm 1 KNAPSACKDPTOPSORT

```
1: for i in {n, n − 1, . . . , 0} do
2:     for j in {0, 1, . . . , S} do
3:         print (i, j)
4:     end for
5: end for
```

The full pseudo-code is straightforward to write, as it closely follows the topological sort and the Dynamic Programming recurrence.

Algorithm 2 KNAPSACK

```
1: for i in {n, n − 1, ..., 0} do
2:     for j in {0, 1, ..., S} do
3:         if i == n then
4:             dp[i][j] = 0 // initial condition
5:         else
6:             choices = []
7:             APPEND(choices, dp[i + 1][j])
8:             if j ≥ s_i then
9:                 APPEND(choices, dp[i + 1][j − s_i] + v_i)
10:            end if
11:            dp[i][j] = MAX(choices)
12:        end if
13:    end for
14: end for
15: return dp[0][S]
```

The dynamic programming solution to the Knapsack problem requires solving $O(nS)$ sub-problems. The solution of one sub-problem depends on two other sub-problems, so it can be computed in $O(1)$ time. Therefore, the solution's total running time is $O(nS)$.

The solution running time is not polynomial in the input size. The next paragraph explains the subtle difference between polynomial running times and pseudo-polynomial running times, and why it matters.

Polynomial Time vs Pseudo-Polynomial Time

The input for an instance of the Knapsack problem can be represented in a reasonably compact form as follows (see Figure below):

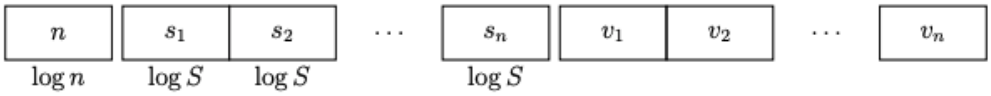

Figure 6: A compact representation of an instance of the Knapsack problem.

- The number of items n, which can be represented using $O(\log n)$ bits.
- n item weights. We notice that item weights should be between $0 \ldots S$ because we can ignore any items whose weight exceeds the knapsack capacity. This means that each weight can be represented using $O(\log S)$ bits, and all the weights will take up $O(n \log S)$ bits.
- n item values. Let V be the maximum value, so we can represent each value using $O(\log V)$ bits, and all the values will take up $O(n \log V)$ bits.

The total input size is $O(\log(n) + n(\log S + \log V)) = O(n(\log S + \log V))$. Let $b = \log S$, $v = \log V$, so the input size is $O(n(v + b))$. The running time for the Dynamic Programming solution is $O(nS) = O(n \cdot 2^b)$.

So, how does our Knapsack solution runtime change if we double the input size? We can double the input size by doubling the number of items, so $n_0 = 2n$. The running time is $O(nS)$, so we can expect that the running time will double.

However, we can also double the input size by doubling v and b, the number of bits required to represent the item weights and values. v doesn't show up in the running time, so let's study the impact of doubling the input size by doubling b. If we set $b_0 = 2b$, the $O(n \cdot 2^b)$ result of our algorithm analysis suggests that the running time will increase quadratically.

The Dynamic Programming solution to the Knapsack problem is a pseudo-polynomial algorithm, because the running time will not always scale linearly if the input size is doubled.

d	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$z_1(d)$	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1
$z_2(d)$	0	0	1	1	1	1	1	5	5	6	6	6	6	6	6
$z_3(d)$	0	0	1	1	1	1	1	5	5	6	9	9	10	10	10
$z_4(d)$	0	0	1	4	4	5	5	5	5	6	9	9	10	13	13
$z_5(d)$	0	0	6	6	7	10	10	11	11	11	11	12	15	15	16

Figure 7: An example of a dynamic programming table/array for a 0-1 Knapsack Problem.

3 0-1 Knapsack Problem with Forfeits

In this thesis we study the 0-1 Knapsack Problem with Forfeits, which is a 0-1 Knapsack Problem that considers soft conflict constraints, or forfeits. In more detail [4], we introduce a forfeit cost to be paid each time that both objects in a so-called forfeit pair are chosen to be part of the solution. This variant can be of use in scenarios in which strict conflicts may lead to infeasible solutions, or the drawback caused by avoiding all conflicts may impact the result more than allowing some of them. We can think of several applications of the problem, including:

- Each object is a machine that needs a worker to be operated. Forfeit pairs represent machines that can only be operated by a worker that we are currently paying, and hence the activation of two such machines requires hiring a new worker, i.e. another salary;
- The chosen items represent the work shift assigned to an employee, and forfeit pairs represent tasks that would involve extras on the salary if assigned together;
- In deciding a series of investments, a cost could derive from making two investment decisions at the same time.

3.1 Mathematical Formulation

Let n be the number of objects, composing the set X. Each object $i \in X$ has an associated profit $p_i > 0$ in the set P and positive weight $w_i > 0$ in the set W, $i = 1, \ldots, n$. Let F be a set of l distinct forfeit pairs $F = \{F_k\}_{k=1,\ldots,l}$,

$F_k \subseteq X$, $|F_k| = 2$ for all $F_k \in F$, and let $d_k > 0$, in the set D, be the forfeit cost associated with F_k, $k = 1, \ldots, l$. Finally, let $b > 0$ be the available budget, that is, the upper bound on the maximum weight of the items chosen to be part of the solution. The problem can be formulated as follows:

$$\max \sum_{i=1}^{n} p_i x_i - \sum_{k=1}^{l} d_k v_k \tag{1}$$

$$\text{s.t.} \tag{2}$$

$$\sum_{i=1}^{n} w_i x_i \leq b \tag{3}$$

$$x_i + x_j - v_k \leq 1 \quad \forall F_k = \{i, j\}, \quad k = 1, \ldots, l \tag{4}$$

$$x_i, v_k \in \{0, 1\} \quad \forall i = 1, \ldots, n, \quad \forall k = 1, \ldots, l \tag{5}$$

where:

- Variable x_i is equal to 1 if object i is selected, and 0 otherwise;
- Variable v_k assumes value 1 if the forfeit cost d_k is to be paid according to the chosen objects, and 0 otherwise.

The problem includes the classical 0-1 Knapsack Problem as special case and is therefore NP-Hard.

3.2 Literature

We consider now GreedyForfeits Algorithm, a basic constructive heuristic for this problem, and an improved heuristic approach , the CarouselForfeits Algorithm, which improves upon the GreedyForfeits Algorithm [4].

3.2.1 GreedyForfeits

The GreedyForfeits algorithm (see the relevant pseudocode depicted below in Algorithms 3 and 4) takes as input the items set X, the profit and weight sets P and W, the budget value b, the forfeits set F, and forfeit costs set D. The set $S \subseteq X$ initialized in line 1 of Algorithm 3 will contain the items chosen to be included in the solution, while the b_{res} value, introduced in line 2, corresponds in any phase of the algorithm to the residual budget, that is, $b_{\text{res}} = b - \sum_{i \in S} w_i$.

The main loop of the algorithm is contained in lines 3–28. In each iteration, we first build the set X_{iter} (lines 4–9), containing the items that can still be added to S. That is, X_{iter} contains any item $i \in X$ which does not currently belong to S, and such that its weight w_i is not greater than b_{res}.

If X_{iter} is empty, clearly no more items can be added, and the algorithm stops returning S (lines 10–12). Otherwise, we evaluate the most promising element of X_{iter} to be added to S. The main idea is to evaluate each item $i \in X_{\text{iter}}$ according to the ratio between profit and weight, p_i.

However, for any forfeit pair $F_k = \{i, j\}$ containing i and such that w_i the other item j is already in S, we subtract from p_i the related cost d_k. The updated profit value, indicated as p'_i, reflects the forfeit costs that would have to be paid if i is added to S. For each $i \in X_{\text{iter}}$, the computation of p'_i is described in lines 14–19, while the computation of the ratio value $ratio_i$ is reported in line 20.

Then, the element $i^* \in X_{\text{iter}}$ corresponding to the maximum ratio value is identified (line 1 of ALgorithm 4). We note that it is possible for p_{i^*} (and therefore for $ratio_{i^*}$) to be a negative value. If this is true, it means that

it is not convenient to add any other item to S, and the set is returned (lines 2–4 of Algorithm 4). Otherwise, both S and b_{res} are updated to reflect the addition of i to the solution (lines 5–6 of Algorithm 4), and the current iteration ends.

Finally, if the main loop ends without encountering any of the two mentioned stopping conditions (meaning that trivially all elements of X could be added to S), the set S is returned.

As in many constructive greedy algorithms, a limit of Greedy-Forfeits is that the contribution of each item composing the solution is evaluated at the moment it is added to it. An item appearing attractive in the first iterations could actually lead to many forfeit costs to be added later on.

Algorithm 3 GreedyForfeits - Part I

Require: (X, W, P, b, F, D)

$S \leftarrow \emptyset$

$b_{\text{res}} \leftarrow b$

While $X \setminus S \neq \emptyset$ **do**

$X_{\text{iter}} \leftarrow \emptyset$

for $i \in X$ **do**

    if $w_i \leq b_{\text{res}}$ **and** $i \notin S$ **then**

        $X_{\text{iter}} \leftarrow X_{\text{iter}} \cup \{i\}$

    end if

end for

if $X_{\text{iter}} = \emptyset$ **then**

    return S

end if

for $i \in X_{\text{iter}}$ **do**

    $p'_i \leftarrow p_i$

    for $F_k = \{i, j\} \in F$ **do**

        if $j \in S$ **then**

            $p'_i \leftarrow p'_i - d_k$

        end if

    end for

    $ratio_i \leftarrow \frac{p'_i}{w_i}$

end for

Algorithm 4 GreedyForfeits - Part II

$i^* \leftarrow \text{argmax}[ratio]$
if $ratio_{i^*} < 0$ **then**
    return S
end if
$S \leftarrow S \cup \{i^*\}$
$b_{\text{res}} \leftarrow b_{\text{res}} - w_{i^*}$
end while
return S

3.2.2 CarouselForfeits

The Carousel Greedy (CG) paradigm, provides a generalized framework to improve the constructive greedy algorithms, posing itself as a trade-off (in terms of computational time and solution quality) among such greedy procedures and meta-heuristics.

The main intuition is that, generally, the choices taken according to the greedy criteria in the first steps of the algorithm could be not very effective due to the lack of knowledge about the subsequent structure of the solution. Therefore, such early choices could end up compromising the quality of the final solution.

In order to overcome this phenomenon, earlier choices are iteratively reconsidered and eventually replaced with new ones. Given a basic constructive heuristic, a CG is composed of three main steps:

1. Using the greedy algorithm, a solution is first built, and then some of its latest choices are discarded, obtaining a partial solution.

2. For a predefined number of iterations, the oldest choice is discarded, and a new one is taken according to the greedy criteria of the basic algorithm.

3. Finally, the solution is completed by applying the greedy algorithm, starting from the partial solution obtained at point 2.

Algorithm 5 CarouselForfeits

Require: $X, W, P, b, F, D, \alpha, \beta$

1: $S \leftarrow \text{GreedyForfeits}(X, W, P, b, F, D)$
2: $S \leftarrow \text{RemoveLastChoices}(S, \beta)$
3: $size \leftarrow |S'|$
4: **for** $i \leftarrow 1$ to $\alpha \times size$ **do**
5: $\quad S \leftarrow \text{RemoveOldestChoice}(S')$
6: $\quad i^* \leftarrow \text{GreedyForfeitsSingle}(X, W, P, b, F, D, S')$
7: $\quad S' \leftarrow S' \cup \{i^*\}$
8: **end for**
9: $S'' \leftarrow \text{GreedyForfeitsInit}(X, W, P, b, F, D, S')$
10: **return** S''

The algorithm (here denoted as Algorithm 5) takes the same input of GreedyForfeits, plus two parameters, α and β, such that $0 \leq \beta \leq 1$ and $\alpha \geq 1$.

Lines 1–2 correspond to the first CG step. We first use our greedy to obtain a feasible solution $S \subseteq X$. We then obtain a partial solution S' by dropping some of the last choices; more precisely, the last $\beta|S|$ added items

are dropped. Let size be $|S'|$ at this point; the second CG step (lines 4–8) is iterated $\alpha \times$ size times.

In each iteration, we first drop from S the oldest choice. We then execute a variant of GreedyForfeits called GreedyForfeitsSingle. It initializes the solution with S instead of the empty set, executes a single iteration of the main loop, identifying the best element to be added i according to our greedy criterion, and returns it. S is then updated to include i.

Finally, in the third and last CG step (line 9), we complete S by executing a second variant of our greedy, GreedyForfeitsInit, which again initializes the solution with S', and completes the solution iterating the main loop until no more items can be added. The resulting solution S'' is returned (line 10).

4 Recovering Beam Search

Our proposed solution for the 0-1 Knapsack Problem with Forfeits is based on a Recovering Beam Search approach. In this section, this algorithm will be described in detail

Recovering Beam Search (RBS) [11] is a hybrid heuristic method for combinatorial optimization problems.

This method is an enhancement of the beam search approach, which in turn is a well-established heuristic approach originally invented in the AI community.

BS consists of a truncated branch and bound with a breadth-first search strategy where only the most promising w nodes at each level of the search tree are selected as nodes to branch from; w is the so-called beam width. Obviously, the larger the beam width, the slower the algorithm.

The nodes evaluation process at each level is the main issue of any BS procedure: typically, a two-stage approach is applied. First, a crude evaluation (filtering phase) is applied to select a reduced number of nodes for the accurate evaluation. This crude evaluation is a one-shot evaluation and is applied to reduce the computational burden of the procedure. Then, the selected nodes are accurately evaluated, and the best w nodes, w being the beam width, are retained for branching. Note that the crude evaluation is actually an optional component of the BS approach.

The accurate evaluation is typically performed by means of bounding procedures for the given problem. The more time-consuming these procedures are, the more time-consuming the overall procedure will be.

An error in the nodes evaluation of any BS procedure that induces the

pruning of a good node (namely a node leading to an optimal or nearly optimal solution) can never be recovered. This is the major drawback of the BS approach: whenever all the best nodes are pruned, the best feasible solution reached may be significantly far from the optimum.

To avoid this, the only means available for a BS procedure is to use a sufficiently large beam width, sometimes dramatically slowing down the procedure's efficiency.

The Recovering Beam Search method overcomes this issue by introducing a recovering step that searches for improved partial solutions with respect to those selected by the beam.

In order to evaluate a limited number of nodes in the search tree, the recovering step searches only for partial solutions situated at the same level of the search tree with respect to those selected by the beam. This step, which allows partial recovery from wrong decisions, is applied in such a way so as to increase only slightly the CPU time required by the procedure.

4.1 Description of the Procedure

Classic Beam Search procedures cannot recover from wrong decisions: if a branch leading to the optimal solution in the search tree is pruned in the nodes evaluation process, there is no way to reach afterwards that solution. The RBS method seeks to overcome this issue by means of a recovering step that searches for improved partial solutions dominating those selected by the beam.

Consider a combinatorial optimization problem where the objective function must be minimized. Assume that a branching scheme and correspond-

ingly a search tree has been devised for that problem either by considering available branch and bound procedures or by having devised an ad hoc exact search tree algorithm.

The node evaluation process is guided here both by lower and upper bound procedures. Each node is evaluated by means of a convex combination of lower (LB) and upper (UB) bounds.

The simplest way to do this is to consider a linear combination, namely the weighted sum $V = (1-\alpha)LB + \alpha UB$, where V is the evaluation function and $0 \leq \alpha \leq 1$ is a parameter generally defined by experimental testing. The more accurate the evaluation function at each node is, the smaller the deviation of the final solution value from the optimal solution value will be. A correct tuning of parameter α allows to obtain high quality results also for problems where either the LB procedure or the UB procedure (but not both) are not too precise.

In the recovering beam search method, the filtering phase works as follows. Problem dependent dominance conditions, denoted as valid dominance conditions, when available, are applied together with so-called pseudo dominance conditions, holding in a heuristic context only. Whenever a valid dominance condition or a pseudo dominance condition applies for a given node, that node is pruned.

In the Recovering Beam Search method (like in the classic Beam Search approach), the beam width is constant and is kept generally fairly low (≤ 10) in order to minimize the overall procedure CPU time. The main feature of the proposed method is the so-called Recovering Phase that is applied at each search tree level.

Let $S = \{\sigma_k, k = 1, ..., l \leq w\}$ be the vector of current partial solutions at a given level. These solutions are considered one at a time. The recovering phase checks, typically by means of interchange operators applied to the current partial solution x, whether solution x is dominated by another partial solution y sharing the same search tree level.

If so, x is discarded. Further, if y does not belong to set S, then it becomes a new current partial solution. If y already belongs to S, then there is room for another partial solution to be examined by means of the recovering step and then retained so as to maintain, when possible, exactly w nodes.

Indeed, this step often allows one to recover from previous wrong decisions in the procedure.

Note that, in the recovering step, a partial solution may be only substituted by another partial solution sharing the same search tree level: this guarantees that the total number of explored nodes is polynomial provided that the search tree depth is polynomial. Note also that the dominance of a partial solution vs another partial solution may be also considered in a heuristic fashion by means of pseudo dominance conditions.

Consider a minimization problem with search tree depth equal to u. The main steps of the Recovering Beam Search method are as follows (see Algorithms 6 and 7).

Algorithm 6 RBS method-Part I (beam width = w, search tree depth = u)

1: **Initialization:**
2: l = search tree level = 0;
3: σ_1 = best current partial solution = root node (typically no variable has been fixed, namely $\sigma_1 = \{\}$);
4: S = vector of current partial solutions = $\{\sigma_1\}$;
5: x = incumbent best solution value = $+\infty$.
6: **for** $k = 1, k \leq \min\{|S|, w\}, k++$ **do**
7: Branch σ_k generating the corresponding children.
8: **Filtering phase:** prune all child nodes that are dominated by means of valid or pseudo-dominance conditions.
9: **end for**
10: Empty set S: $S = \{\}$.
11: **for** each remaining child node **do**
12: Compute LB and UB. **IF** UB $< x$, **THEN** x = UB.
13: Compute the evaluation function $V = (1 - \alpha)\text{LB} + \alpha\text{UB}$ with $0 \leq \alpha \leq 1$.
14: **end for**
15: *(Algorithm continued on next page...)*

Algorithm 7 RBS method-Part II (continued from the previous page)

1: Sort the set T of remaining children nodes in non-decreasing order of their evaluation function: let σ_k be the k-th best node.

2: Set $k = 1$.

3: **while** $(|S| < w)$ **AND** $(k \leq |T|)$ **do**

4: **Recovering step:** search for a partial solution $\bar{\sigma}_k$ that dominates σ_k (and shares with σ_k the same search tree level) by means of interchange operators. **IF** $\bar{\sigma}_k$ is found, **THEN** set $\sigma_k = \bar{\sigma}_k$. **IF** $\sigma_k \notin S$, **THEN** $S = S \cup \{\sigma_k\}$, **ELSE** prune σ_k.

5: $k = k + 1$.

6: **end while**

7: $l = l + 1$. **IF** $l < u$, **GOTO** 2, **ELSE STOP**: x is the final solution value.

5 Recovering Beam Search for the 0-1 Knapsack Problem with Forfeits

In this section we will detail our approach for the 0-1 Knapsack Problem with Forfeits, which is based on the Recovering Beam Search method. Then, in the next section, computational results will be shown, compared to the commercial solver CPLEX.

5.1 Data Representation

As shown in section 4, the parameters we need to consider are:

- The weights, which represent the weight of adding an item to the knapsack
- The profits, which represent the profit of adding an item to the knapsack
- Forfeit pairs, the pairs i, j for which a forfeit value exist
- Forfeit values, which represent the penalty to the objective function of including a certain pair of items i, j in the knapsack

The weights, profits and forfeit values will be stored as integer arrays, while the forfeit pairs will be stored as arrays of pairs of integers (a pair is considered as an array of size 2).

As we will show later, for each item i it would be ideal to have relatively quick access to other items j such that i, j is a forfeit pair and the corresponding forfeit value. To achieve this result, we can represent forfeit pairs

as edges in an undirected weighted graph and store it as an adjacency list. In this way, for each item i accessing all the items j such that i, j is a forfeit pair, we will have a complexity $O(|E_i|)$, where $|E_i|$ is the number of edges connected to the item i in the graph.

5.2 Order of the Items

In a binary tree-like procedure such as the Branch and Bound algorithm, at the i-th level of the tree the branching will be done on the i-th variable in a specific order. This order could be relevant in achieving better performances.

We will now examine different ways to permute the variables and how to efficiently access the correct variable at each level of the tree.

Two different ways to sort/permute the items have been compared in our work:

- Sorting by the ratio $\frac{values_i}{weights_i}$. This will sort based on the total value per single unit of weight, which is a common idea in solutions for the regular 0-1 Knapsack Problem.

- Sorting by the ratio $\frac{values_i}{forfeitCount_i}$. This will be sorted based on the total value per single unit of forfeit induced by the item i. This sorting method would put first items that have a higher value compared to the total forfeit value of pairs where they are included.

To access efficiently the correct item at any level of the tree, we can use a permutation array such that at the i-th position $perm_i$ will be the i-th item in the decided order, where $perm$ is a permutation of the values $0, ..., n-1$.

Regarding the sorting method, we used the default C++ sorting function provided by the Standard Template Library (STL) std::sort(). This function is implemented using a variant of the introsort algorithm, which combines quicksort, heapsort, and insertion sort. The time complexity in this case is $O(nlog(n))$, where n is the number of items.

5.3 Upper Bound Computation

To estimate the upper bound at any particular node of the tree, we used a dynamic programming approach.

Lemma 1 *The solution of a 0-1 Knapsack Problem without forfeits with the same parameters (weights, profits) is an upper bound to the original problem.*

Proof 2 *If the forfeits are* 0 *for each pair of items* (i, j) *such that* $0 \leq i, j \leq n-1$*, then the problem is exactly the same as the 0-1 Knapsack Problem.*

By contradiction, if a better upper bound exists such that the contribution of the forfeit penalty to the objective function is positive, then it would mean that the contribution of the sum of profits in the objective function is higher than the one for the relaxed problem. Let us define this hypotethical subset of items that contradict our claim as S_p*.*

Let S_r *be the subset of items that yield the optimal feasible solution for the relaxed problem. If our claim is untrue, then it would mean that a certain subset* S_p *exists such that the corresponding solution is feasible and:*

$$\sum_{i \in S_r} x_i v_i < \sum_{j \in S_p} x_j v_j \tag{6}$$

If such a set S_p *would exist, then that solution would also be the optimal solution to the relaxed version of the problem, thing which concludes the proof.*

This is obvious considering that this relaxed solution will be better than any other possible solution. This upper bound will be the solution of the original problem with 0 forfeits applied to the objective function.

Now, let us see how to compute this upper bound given a partial solution in the tree. Let $x_k, x_{k+1}, ..., x_{n-1}$ be a suffix of variables which are already set to 0 or 1 in a partial solution. We can compute the corresponding upper bound by finding the total weight sum for this partial solution W_k.

The upper bound will then be $dp[k-1][W-W_k]$, where $dp[i][j]$ is the solution to the 0-1 Knapsack Problem with the first i elements and j weight, minus the forfeits induced by the variables in the partial solution.

The dynamic programming table will be precomputed before running the recovering beam search and solutions to the related subproblems can be computed in $O(1)$. The time and memory complexity for the construction of the dynamic programming table is $O(n * W)$.

The array xorder is used to access the $i-th$ item in the sorted order as described before.

5.4 Lower Bound Computation

Given the upper bound computed through dynamic programming as described before, the next step is determining how to derive a lower bound, therefore a feasible solution, for a certain node of the tree.

In a specific node of the tree, the variables $x_k, x_{k+1}, ..., x_{n-1}$ are already fixed to either 0 or 1, so what is remaining is to find the values of the variables $x_0, ..., x_{k-1}$ that yield the given upper bound value.

This can be computed in $O(n)$ by backtracking through the dynamic

Algorithm 8 Building Dynamic Programming Table for 0-1 Knapsack

```
1: procedure BuildDP(dp, n, w, weights, values, x_order)
2:     for j ← 0 to w do
3:         if weights[x_order[0]] ≤ j then
4:             dp[0][j] ← values[x_order[0]]
5:         end if
6:     end for
7:     for i ← 1 to n − 1 do
8:         for j ← 0 to w do
9:             dp[i][j] ← dp[i − 1][j]
10:            if j − weights[x_order[i]] ≥ 0 then
11:                dp[i][j] ← max(dp[i][j], dp[i − 1][j − weights[x_order[i]]] +
    values[x_order[i]])
12:            end if
13:        end for
14:    end for
15: end procedure
```

Algorithm 9 GetOrder Function

```
1: function GetOrder(x_order, weights, values, forf_count)
2:     Sort x_order by values[a]/weights[a] > values[b]/weights[b]    ▷ Sort in decreasing order of
    value-to-weight ratio
3:     return x_order
4: end function
```

programming array.

This procedure (Algorithm 9) checks whether a valid transition between a subproblem with an item included and another one with that item not included (but everything else unaltered) exists. If this is the case, it means that the item was included in the knapsack in the optimal solution of the subproblem we were starting with, the one which does not include the items already fixed in the tree.

Assuming that the solution made of these items is feasible, its value may possibly be very far from the corresponding upper bound. Logically, if the number of forfeit pairs and their value is big enough, their contribution to the optimal solution will be non-negligible, and thus the solution to the relaxed version of the problem could be very far from the one of the original problem.

To address this potential issue, we can add an additional local search to this starting solution given by the procedure described before. In $O(n + P)$, where P is the number of forfeit pairs, we can try to individually flip the value of each variable x_i corresponding to the items and check whether this change improves upon the current solution value.

The algorithm to get the lower bound is shown in Algorithms 10, 11, 12 and 13.

Algorithm 10 Update Active Variables Function

```
1: procedure UpdateActiveVariables(n, dp, vis, currentweight, pos, weights, values, x_order)
2:     while pos > 0 do
3:         newcurrentweight ← currentweight − weights[x_order[pos]]
4:         if newcurrentweight < 0 then
5:             pos ← pos − 1
6:             continue
7:         end if
8:         if dp[pos − 1][newcurrentweight] = dp[pos][currentweight] − values[x_order[pos]] then
9:             currentweight ← newcurrentweight
10:            vis[x_order[pos]] ← true
11:        end if
12:        pos ← pos − 1
13:    end while
14:    if pos = 0 and currentweight ≥ weights[x_order[0]] then
15:        vis[x_order[0]] ← true
16:    end if
17: end procedure
```

Algorithm 11 Get Lower Bound Function (Part I)

```
1: function GetLowerBound(vis, adj, pairs, forfeits, values, weights, w)
2:     n ← size of vis
3:     p ← size of pairs
4:     tot_value ← 0
5:     tot_weight ← 0
6:     for i ← 0 to n − 1 do
7:         if vis[i] then
8:             tot_value ← tot_value + values[i]
9:             tot_weight ← tot_weight + weights[i]
10:        end if
11:    end for
12:    for i ← 0 to p − 1 do
13:        if vis[pairs[i].first] and vis[pairs[i].second] then
14:            tot_value ← tot_value − forfeits[i]
15:        end if
16:    end for
17: end function
```

Algorithm 12 Get Lower Bound Function (Part II)

```
1: for i ← 0 to n − 1 do
2:     if not vis[i] then
3:         continue
4:     end if
5:     curr ← tot_value − values[i]
6:     for each neighbor in adj[i] do
7:         if vis[neighbor.first] then
8:             curr ← curr + neighbor.second
9:         end if
10:    end for
11:    if curr > tot_value then
12:        tot_value ← curr
13:        tot_weight ← tot_weight − weights[i]
14:        vis[i] ← false
15:    end if
16: end for
17: return tot_value
18:
```

Algorithm 13 Get Lower Bound Function (Part III)

```
1: for i ← 0 to n − 1 do
2:     if not vis[i] then
3:         continue
4:     end if
5:     curr ← tot_value + values[i]
6:     neww ← tot_weight + weights[i]
7:     if neww > w then
8:         continue
9:     end if
10:    for each it in adj[i] do
11:        if vis[it.first] then
12:            curr ← curr − it.second
13:        end if
14:    end for
15:    if curr > tot_value then
16:        tot_value ← curr
17:        tot_weight ← neww
18:        vis[i] ← true
19:    end if
20: end for
21: return tot_value
22:
```

5.5 Recovering Beam Search

The core of our solution consists on the Recovering Beam Search function, which will be the focus of this subsection.

5.5.1 Branching and Solutions Generation

At each level of the search tree, we will have a subset of solutions, whose size depends on the chosen *beam* parameter. A solution is represented as an array of values 0 and 1, but also -1 for variables which are not already assigned to a value in the procedure.

At each level i of the tree, with $0 \leq i \leq n-1$, the branch will be on the $n-i$-th element of the array xvalues. Reminder that the array xvalues determines the permutation of items and corresponding order of visit. Starting from the back will make it possible to quickly compute the upper bound as described before.

New solutions are thus generated at each level of tree by performing the described branch operation on the solution set generated at the previous step (level $i-1$).

5.5.2 Recovering Step

The recovering step is a crucial aspect of a Recovering Beam Search-based solution, as it would allow us to potentially recover better partial solutions discarded during the tree exploration process.

In our approach, this step is performed before the computation of the lower bound, upper bound, and evaluation function for a specific partial

solution. We will use a property of the problem to find a partial solution to another one that dominates it.

Lemma 2 *Let S be a partial solution and V its corresponding value of the objective function. Let $S_k \subseteq S$ such that $S_k = S \setminus i$, where i is the i-th item, and V_k be the value of the objective function corresponding to S_k.*

If $V_k \geq V$, then S_k dominates S.

Clearly if an item induces equal or more total forfeit than its profit, then removing that item from the knapsack will yield a solution which is strictly better than the older one, as its removal would also free up space from the knapsack and increase the remaining budget.

Thus, we can recover a better partial solution by checking for each item i such that $pos \leq i \leq n-1$, where pos is a variable that tracks the correct position of the array $xvalues$, calculated as $pos = n - 1 - level$, where $0 \leq level \leq n-1$.

It is also important to consider the efficiency of this recovery process. We implemented a $O(n+P)$ approach to verify for each item i whether it's optimal to remove it or not, according to the definition expressed before, by iterating through items and their forfeit edges. Having an adjacency list representation of the forfeit pairs allows us to perform this operation in $O(n+P)$ instead of $O(n*P)$, as we don't need to iterate through the entire array of forfeit pairs for each item.

Note also that an $O(n+P)$ procedure is relatively inexpensive compared to the algorithm as a whole. The overall time and memory complexity of the algorithm will be described later in this section.

Algorithm 14 Recovering Beam Search - Part I

1: Let $newsols$ be a vector of vectors of integers
2: Let $evaluation\_functions$ be a vector of integers
3: Let $temporary\_solutions$ be a vector of integers
4: Let $sols\_set$ be an unordered set of vectors of integers
5: **for** $i \leftarrow 0$ **to** $\min(\text{beam}, |sols|) - 1$ **do**
6: $\quad sols[i][pos] \leftarrow 0$
7: $\quad newsols$.push_back($sols[i]$)
8: $\quad sols[i][pos] \leftarrow 1$
9: $\quad newsols$.push_back($sols[i]$)
10: **end for**
11: $evaluation\_functions$.resize($|newsols|$)
12: $sols$.clear()

Algorithm 15 Recovering Beam Search - Part II

```
1: for i ← 0 to (newsols.size() − 1) do
2:     Let vis be a vector of booleans of size n, initialized to false
3:     for j ← n − 1 downto pos do
4:         if newsols[i][j] = 1 then
5:             vis[x_order[j]] ← true
6:         end if
7:     end for
8:     for j ← n − 1 downto pos do
9:         if newsols[i][j] = 0 then
10:            continue
11:        end if
12:        profit ← values[x_order[j]]
13:        penalty ← 0
14:        for each neighbor in adj[x_order[j]] do
15:            if vis[neighbor.first] then
16:                penalty ← penalty + neighbor.second
17:            end if
18:        end for
19:        if profit ≤ penalty then
20:            vis[x_order[j]] ← false
21:            newsols[i][j] ← 0
22:        end if
23:    end for
24: end for
```

5.5.3 Filtering Solutions

The next step of the algorithm is filtering solutions that are non-feasible or have a lower upper bound than the current best found feasible solution.

We can find if a solution is feasible in $O(n)$ by just adding the corresponding weights. The computation of the lower and upper bound for a partial solution has been descibed in sections 5.3 and 5.4.

Let S_l be the new partial solutions generated at a specific level of the tree, then the complexity of this filtering step is $O(|S_l| * (n + P))$.

For each unfiltered solution, we then compute its evaluation function, which, as described in Section 6, is a weighted sum of upper and lower bound for a solution. Based on this value, new partial solutions will be sorted in non-increasing order and inserted into the new partial solutions set to be processed in the next level of the tree search.

Sorting solutions by their evaluation function values have a time complexity of $O(|FS_l| * log(|FS_l|)$, where FS_l is the partial solutions set S_l after filtering.

Algorithm 16 Solutions Filtering and Evaluation Function Computation - Part I

1: **for** $i \leftarrow 0$ **to** (**newsols.size()** $- 1$) **do**
2: Let $currentWeight \leftarrow w$
3: Let $currentvalue \leftarrow 0$
4: Let vis be a vector of booleans of size n
5: **for** $j \leftarrow n - 1$ **downto** pos **do**
6: **if** $newsols[i][j] = 1$ **then**
7: $currentWeight \leftarrow currentWeight - weights[x\_order[j]]$
8: $currentvalue \leftarrow currentvalue + values[x\_order[j]]$
9: $vis[x\_order[j]] \leftarrow$ true
10: **end if**
11: **end for**
12: **if** $currentWeight < 0$ **then**
13: **continue**
14: **end if**
15: $upperbound \leftarrow 0$
16: **if** $pos = 0$ **then**
17: $upperbound \leftarrow currentvalue$
18: **else**
19: $upperbound \leftarrow currentvalue + dp[pos - 1][currentWeight]$
20: **end if**

Algorithm 17 Solutions Filtering and Evaluation Function Computation - Part II

1: **for** $j \leftarrow 0$ **to** $(p-1)$ **do**
2: **if** $vis[pairs[j].first]$ **and** $vis[pairs[j].second]$ **then**
3: $upperbound \leftarrow upperbound - forfeits[j]$
4: **end if**
5: **end for**
6: **if** $upperbound < lowerbound$ **then**
7: **continue**
8: **end if**
9: update_active_variables$(n, dp, vis, currentWeight, pos - 1, weights, values, x_order)$
10: Let $currlb \leftarrow$ get_lower_bound$(vis, adj, pairs, forfeits, values, weights, w)$
11: **if** $currlb > lowerbound$ **then**
12: $optimal\_x \leftarrow vis$
13: **end if**
14: $lowerbound \leftarrow \max(lowerbound, currlb)$
15: **if** $currlb > lowerbound$ **then**
16: $optimal\_x \leftarrow vis$
17: **end if**
18: $lowerbound \leftarrow \max(lowerbound, currlb)$
19: $evaluation\_functions[i] \leftarrow$ compute_v$(\alpha, currlb, upperbound)$
20: $temporary\_solutions$.push_back(i)
21: **end for**

5.5.4 Inserting new Partial Solutions

The final step of the Recovering Beam Search function is to insert new partial solutions into the set which will be processed into the next level of the tree. The size of this set cannot exceedthe beam size.

To store solutions, we used a hash set of arrays, thing which allowed us to verify whether a partial solution was already inserted there. An hash set of arrays can allow insertion and retrieval of an array in $O(|A|)$, where A is the array in question.

5.5.5 Overall Time and Memory Complexity

Given all the operations described in this section, the time complexity of the algorithm is: $O(n * W + n * b * (n + P))$. Note that b is the beam size and is constant. The memory complexity is: $O(n * W + p)$.

Algorithm 18 Updating Solutions

```
1: for i ← 0 to (temporary_solutions.size() − 1) and (sols.size() < beam) do
2:     Let vis be a vector of booleans of size n
3:     Let sol_position ← temporary_solutions[i]
4:     for j ← 0 to (newsols[sol_position].size() − 1) do
5:         if newsols[sol_position][j] = 1 then
6:             x_value_position ← x_order[j]
7:             vis[x_value_position] ← true
8:         end if
9:     end for
10:    for j ← 0 to (n − 1) do
11:        x_value_position ← x_order[j]
12:        if vis[x_value_position] then
13:            newsols[sol_position][j] ← 1
14:        else if newsols[i][j] ≠ −1 then
15:            newsols[sol_position][j] ← 0
16:        end if
17:    end for
18:    if ¬sols_set.count(newsols[sol_position]) then
19:        sols.push_back(newsols[sol_position])
20:        sols_set.insert(newsols[sol_position])
21:    end if
22: end for
23: pos ← pos − 1
```

6 Computational Results

In this section will show how our solution performs computationally, compared to "IBM ILOG CPLEX Optimization Studio", a known mathematical programming solver. The comparisons will be made for fixed values of n (number of items) and 5 different instances for each chosen value of n.

The parameters *alpha* and *beam* of the recovering beam search used are: $alpha = 0.95$ and $b = 10$.

6.1 Description of Used Instances and Environment

The instances of the problem used for our experiments were generated randomly with the following structure:

- Number of items n.
- Weights from 3 to 20.
- Profits from 5 to 25.
- Number of forfeit pairs $6n$. These pairs are always unique and there are no pairs consisting of an item with itself.
- Forfeit values from 2 to 15.
- Budget $3n$.

Our solution has been written and executed in C++, while CPLEX has been used in Python from the *docplex* library. The instances have also been generated in Python.

The machine used for all our tests is a Mac Mini with M2 Apple Silicon processor, 8-Core CPU 10-Core GPU and 8 GB of RAM.

To have fair comparison between solutions, only 1 CPU thread was used for both approaches.

6.2 General Results

We will now compare results of our heuristic solution with the CPLEX solver. Comparisons will be made by comparing the value of the best feasible solution obtained by our solution to the one found by CPLEX, for a fixed time limit of 60 seconds. The value of n, which determines the size of the instances, will be $n = 5000$ and then $n = 10000$.

Table 1: Results Comparison for n = 5000

Instance	CPLEX	Heuristic	Difference %
n_{5000_1}	20478.0	21133	3.19%
n_{5000_2}	18915.0	20755	9.73%
n_{5000_3}	19715.0	20924	6.14%
n_{5000_4}	19239.0	21338	10.91%
n_{5000_5}	20089.0	20702	3.05%

Let us compute the average difference percentage between CPLEX and the heuristic solution as:

$$\text{Average Difference Percentage} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\text{Heuristic}_i - \text{CPLEX}_i}{\text{CPLEX}_i} \times 100\right)$$

For $n = 5000$ and the considered instances, this value is equal to 6.606%.

Table 2: Results Comparison for n = 10000

Instance	CPLEX	Heuristic	Difference %
n_{10000_1}	38289.0	41315	7.91%
n_{10000_2}	38682.0	41159	6.41%
n_{10000_3}	39291.0	41130	4.67%
n_{10000_4}	39217.0	41617	6.12%
n_{10000_5}	39303.0	42043	6.98%

For $n = 10000$, the average difference percentage is 6.62%, which shows a similar behavior even after doubling the number of items.

These results show the effectiveness of our solution for relatively large values of n, when a well known commercial solver like CPLEX cannot produce an optimal solution for this problem in 60 seconds.

6.3 Choice of Beam Size

We will now show computational experiments that highlight the results by changing the value of the beam size. The instance used is named as n_{10000_8}, with $n = 10000$. The time limit has been set again as 60 seconds.

Table 3: Beam Tests

Beam Size	Objective
10	42020
20	42020
200	41972

We can see how increasing the beam size to 200 leads to worse results, as the processing time for each level of the tree becomes 20 times slower. It is important to notice that a larger beam size would lead to bigger memory usage.

6.4 Results for Bigger Time Limits

In this subsection, we will evaluate results for bigger time limits, 5 minutes and 10 minutes, and $n = 10000$. We would like to show the time required by CPLEX to outperform our heuristic solution.

Table 4: Results for Larger Time Limits - I - Instance n_{10000_6}

Time	**CPLEX**	**Heuristic**
5 minutes	39144	41472
10 minutes	43080	41476

Table 5: Results for Larger Time Limits - II - Instance n_{10000_7}

Time	**CPLEX**	**Heuristic**
5 minutes	39867	41696
10 minutes	44990	41727

We can notice that at the 5 minutes mark, our heuristic still outperforms CPLEX, by returning a larger value of the objective function. At the 10 minutes mark, CPLEX outperforms our heuristic solution. The heuristic solution improves more slowly, as the complexity for traversing each level of the search tree in breadh-first order is not negligible.

7 Conclusions

In this chapter, we recap our accomplishments in this thesis and show potential future developments for the proposed algorithm.

7.1 Our Solution for the 0-1 Knapsack Problem with Forfeits

In this work, we proposed a heuristic approach for the 0-1 Knapsack Problem with Forfeits. Specifically, our aim was to develop an heuristic approach that, for the specified instances and large number of items, would outperform a highly optimized and well known commercial solver for mathematical optimization problems.

Our heuristic combined multiple techniques such as dynamic programming, branch and bound, and beam search to achieve those results.

7.2 Improving the Upper Bound for each Partial Solution

Finding an efficient and reliable way to compute an upper bound for a partial solution in this problem is a non-trivial and very difficult task.

We believe that our algorithm has room for future improvements, both in execution time and getting a value of the objective function even closer to the optimal one with more consistency.

Our upper bound strategy, although relatively efficient, does not take into account the contribution of forfeits in its calculation. This can result for

specific instances where the contribution of forfeits to the objective function is substantial in generating upper bounds that quite far from the optimal solution.

By finding a similarly efficient way to compute an upper bound for a partial solution in the process that considers both the positive and negative contributions to the objective function, we could potentially improve the upper bound computation even further. This improvement could allow to quickly remove certain partial solutions and converge more quickly to the final solution of the algorithm, while potentially resulting in a better solution value (closer to the optimal one for a greater variety of instances).

References

[1] V. Cacchiani M. Iori A. Locatelli S. Martello. Knapsack problems — an overview of recent advances. part i: Single knapsack problems. *Computers and Operations Research, 143*, 2022, 105692.

[2] V. Cacchiani M. Iori A. Locatelli S. Martello. Knapsack problems — an overview of recent advances. part ii: Multiple, multidimensional, and quadratic knapsack problems. *Computers and Operations Research, 143*, 2022, 105692.

[3] W. Zhang B. Wang Y. Hu. A new knapsack public-key cryptosystem. *2009 Fifth International Conference on Information Assurance and Security, Xi'an, China, 2009, pp. 53-56.*

[4] R. Cerulli C. D'Ambrosio A. Raiconi G. Vitale. The knapsack problem with forfeits. *In: M. Baiou, B. Gendron, O. Gunluk, A.R. Mahjoub (eds) Combinatorial Optimization. ISCO 2020. LNCS 12176. Springer, Cham, 2020.*

[5] R. Tadei F. Della Croce. Elementi di ricerca operativa. *ESCULAPIO Bologna, 2010.*

[6] D. Williamson W. Qian. Lecture 25. *ORIE 6300 Mathematical Programming I, 2014,* `https://people.orie.cornell.edu/dpw/orie6300/Lectures/lec25.pdf`.

[7] M. Garey D. Johnson. Computers and intractability: A guide to the theory of np-completeness. *W. H. Freeman Co., New York, NY*, 1979.

[8] G. Righini. Branch-and-bound. *Operations Research Complements, 2022,* https://homes.di.unimi.it/righini/Didattica/RicercaOperativa/Materiale/08%20-%20Branch-and-bound.pdf.

[9] T. Magnanti J. Orlin. Dynamic programming. *15.053: Optimization Methods in Business Analytics, 2022,* https://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

[10] E. Demaine S. Devadas. The knapsack problem. *6.006 Introduction to Algorithms Recitation 19, 2011,* https://courses.csail.mit.edu/6.006/fall11/rec/rec21_knapsack.pdf.

[11] R. Tadei F. Della Croce M. Ghirardi. Recovering beam search: Enhancing the beam search approach for combinatorial optimization problems. *Journal of Heuristics, 10, 89-104, Kluwer, 2004.*