Time-Bomb Knapsack Problems

Reus, Richard J.

Award date:
2022

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

Time-Bomb Knapsack Problems

Bachelor final project

Richard Reus

Supervisors:
prof. dr. Frits Spieksma
Andrés Lopéz Martínez, MSc

Final version

Eindhoven, July 2022

Abstract

We consider the 0-1 Time-Bomb Knapsack Problem (01-TB-KP), which is a stochastic version of the 0-1 Knapsack Problem in which, in addition to a profit and weight, each item has an additional parameter which represents the probability of exploding. Should such an item explode the entire contents of the knapsack is destroyed, modelling real world scenarios in logistics and cloud computing scheduling. We study and expand upon the literature presented in "Monaci" et al.[10]. We then consider a variant of the 01-TB-KP in which we add an additional constraint bounds the probability that the knapsack explodes. We present an algorithm based on Lagrangian Relaxation that yields approximations of optimal solutions for this variant and evaluate it's performance on a set of benchmark instances. The computational experiments show that for a relatively low proportion of time-bomb items this algorithm performs reasonably well, but the accuracy deteriorates the higher the proportion of time-bomb items get. In addition, for now the algorithm is quite slow and can thus only be used instances with a small number of items.

Contents

Chapter 1

Introduction

Knapsack problems have been studied for more than a century, the name stemming from trying to pack the most valuable items without overloading the knapsack. The formal definition of a knapsack problem is given as follows: We are given an item set N, consisting of n items j with profits p_j, weights w_j and capacity c. The objective is find a subset of N such that the total weight does not exceed the capacity and the profit is maximized [6]. When each item can only be included once this is known as a 0-1 Knapsack Problem. It is known that 0-1 Knapsack problems are $\mathcal{NP}$-hard , although algorithms exists that can solve them in pseudo-polynomial time. [6]

Uncertainty is an inherent property of knapsack problems in real world situations and these types of problems are called "Stochastic Knapsack Problems" or "SKP's". The most common SKP's are variants of the regular 0-1 Knapsack Problem, where some part of the input is no longer deterministic. For example, the weights of the items could be unknown, while following a known distribution. The true weights are only revealed once all items are selected. A possible objective is then to maximize the expected profit given that you have to remove an item if the capacity is exceeded, as explored in Merzifonluoğlu et al. [9]

Instead of unknown weights it could also happen that the weights are deterministic, but profits of the items are unknown, follow a known distribution and only become known once all items are selected. This type of problem arises for example in the financial world when an investor one wants to allocate funds among several projects. In this case a possible objective could be to maximize the probability that the total profit exceeds a certain value as described Henig [5].

This Thesis focuses on a different variant of the 0-1 Knapsack Problem, namely the 0-1 Time-Bomb Knapsack Problem (01-TB-KP) as discussed in the following paper [10]. In this problem a subset of the items are so-called time-bomb items, in other words they can explode with a given probability. If a packed item does indeed explode this would destroy the entire content of the knapsack and thus yield no profit. This type of problem arises, for example, in the field of transportation when some of the goods to be transported are potentially hazardous such as lithium-ion batteries [8]. Another scenario where the 01-TB-KP arises is in the management of data centers. In this case, allocating a server or application to a container or machines to a server corresponds to packing items into a knapsack. If some of the servers or applications are vulnerable there is a chance for an attacker to exploit them and take over the container.

The aim of this thesis is to study and expand upon the literature discussed in "Monaci" et al. [10] and develop an algorithm to solve or approximate optimal solutions of a variant of the 01-TB-KP. In this variant an additional constraint has been added to the problem which gives a bound on the probability that the knapsack explodes and its content is lost.

Chapter 2

Preliminaries

In the 0-1 Knapsack Problem we are given a knapsack with capacity $c \in \mathbb{N}$ and $n \in \mathbb{N}$ items. All items have a profit $p_j \in \mathbb{N}$ and a weight $w_j \in \mathbb{N}$. The 01-TB-KP has, in addition to all the parameters above, an additional parameter $q_j \in [0, 1]$ which represents the probability of an item exploding. In contrast to the original 01- Knapsack Problem it would not make much sense to solely maximize the profit of this type of problem since each item has a probability of exploding and whenever a selected item explodes the entire content of the knapsack is lost. Therefore, instead of maximizing the profit the objective in the 01-TB-KP is to maximize the total expected profit. This is of course done by multiplying the total profit of the selected items by the probability that the knapsack will explode.

For each item j the probability that it does not explode is denoted by $\Pi_j = 1 - q_j$ and $T = \{j \in \{1, ..., n\} : \Pi_j < 1\}$. In addition the variables $\alpha_j = 1 - q_j x_j$ are introduced for notation convenience. This new variable has the favourable property that if $x_j = 1$ then $\alpha_j = 1$ and otherwise if $x_j = 0$ then $\alpha_j = 0$.
The 01-TB-KP can then be modelled as follows:

$$max \quad (\sum_{j=1}^{n} p_j x_j)(\prod_{j=1}^{n} \alpha_j) \tag{2.1}$$

$$s.t \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{2.2}$$

$$\alpha_j = 1 - q_j x_j, \qquad j \in \{1, ..., n\} \tag{2.3}$$

$$\underline{x} \in \{0, 1\} \tag{2.4}$$

Here the objective function 2.1 maximizes the total expected profit, equations 2.2 and 2.4 represent the constraints of the system, namely the weight capacity constraint of the knapsack and the 0-1 constraint respectively, and equation 2.3 defines the variables α_j.

In chapter 3 we will discuss a more general form of the 01-TB-KP described above. Instead of a single capacity constraint we now introduce a system of linear constraints which could for example represent, in addition to the weight constraint, additional constraints on the dimensions of the selected items. We will refer to this problem as the 01-TB-LIN

A model for the 01-TB-LIN is given as follows:

$$max \quad (\sum_{j=1}^{n} p_j x_j)(\prod_{j \in T} \alpha_j) \tag{2.5}$$

$$s.t \quad A\underline{x} \leq \underline{b} \tag{2.6}$$

$$\alpha_j = 1 - q_j x_j, \qquad j \in \{1, ..., n\} \tag{2.7}$$

$$\underline{x} \in \{0, 1\} \tag{2.8}$$

Chapter 3

Upper and lower bounds for the 01-TB-KPG

"Monaci" et al, [10] introduce two upper and lower bounds for the 01-TB-KP given by 2.1 - 2.4. The bounds are then used and tested in a branch and bound algorithm. It turned out that the combinatorial upper and lower bound were easier to compute, but were usually outperformed by bounds 3.7 and 3.14. In this chapter we prove that these bounds also hold if there is more than one linear constraint. In other words this chapter aims to prove that the upper and lower bounds can also be used in a branch and bound algorithm for model 2.5 - 2.8.

Upper bounds

From "Monaci" et al. [10] it is known that the optimal objective value of the following deterministic 01-KP provides an upper bound for the 01-TB-KP.

$$max \quad \sum_{j=1}^{n} p_j \pi_j x_j \tag{3.1}$$

$$s.t \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{3.2}$$

$$\underline{x} \in \{0,1\} \tag{3.3}$$

The following claim extends this result for the 01-TB-LIN:
Claim 3.1: *The optimal value $\overline{Z_1}$ of the following deterministic 01-KP*

$$\overline{z_1} = max \quad \sum_{j=1}^{n} p_j \pi_j x_j \tag{3.4}$$

$$s.t \quad A\underline{x} \leq \underline{b} \tag{3.5}$$

$$\underline{x} \in \{0,1\} \tag{3.6}$$

is an upper bound for the 01-TB-LIN

proof. We show that for any $\underline{x}$ the objective function (5) is always larger than the objective function (1):

$$(\sum_{j=1}^{n} p_j x_j)(\prod_{j\in T}) = \sum_{j=1}^{n}(p_j x_j \prod_{i\in T} \alpha_i) \leq \sum_{j=1}^{n} p_j x_j \alpha_j =$$

$$= \sum_{j=1}^{n} p_j x_j (1 - q_j x_j) = \sum_{j=1}^{n} p_j (1 - q_j) x_j^2 = \sum_{j=1}^{n} p_j (1 - q_j) x_j = \sum_{j=1}^{n} p_j \pi_j x_j$$

Furthermore, any feasible solution $\underline{x}$ of model (1)-(4) is also a feasible solution for model (5)-(7). The above shows that for each of these solutions, function (5) bounds (1) from above. So for the optimal solution $\underline{x^*}$ of model (5)-(7) with objective value $\overline{z_1}$ it holds that $\overline{z_1}$ overestimates the optimal objective value of model (1)-(4). For if there existed a feasible solution x^{**} for model (1)-(4) with objective value $z_2 \geq \overline{z_1}$ then x^{**} would be a feasible solution for model (5)-(7) with objective value larger than z_2 contradicting the optimality of x^*. □

Following "Monaci" et al's approach, we obtain a second upper bound for the extended model by dropping the integrality requirement 2.8 from model 2.5 - 2.8 which yields the following model:

$$\overline{z}_2 = max \quad (\sum_{j=1}^{n} p_j x_j)(\prod_{j\in T} \alpha_j) \tag{3.7}$$

$$s.t \quad A\underline{x} \leq \underline{b} \tag{3.8}$$

$$\alpha_j = 1 - q_j x_j, \qquad j \in \{1, ..., n\} \tag{3.9}$$

$$\underline{x} \in [0,1]^n \tag{3.10}$$

Clearly the set of feasible solutions of the model 2.5 - 2.8 is a subset of the set of feasible solutions of the continuous relaxation. Thus the optimal solution of this relaxation $\overline{z}_2$ is larger or equal to the optimal solution of the original problem and hence $\overline{z}_2$ is an upper bound for the original problem.
"Monaci" et al. also discusses a method of solving the continuous relaxation. The core of the method relies on the fact that the objective function $f(x) = (\sum_{j=1}^{n} p_j x_j)(\prod_{j\in T} \alpha_j)$ is a concave function (second derivative is negative) being maximized on a convex set $P = \{x \in [0,1]^n : \sum_{j=1}^{n} w_j x_j \leq c\}$. In the more general case with more constraints the objective function does not change and is therefore still concave. Furthermore the region that is being maximized over is now given by $P = \{x \in [0,1]^n : A\underline{x} \leq \underline{b}\}$ which is convex. Therefore, the same methods to find optimal solutions to the continuous relaxation can be used.

Lower bounds

The combinatorial upper bound discussed above has the same feasible set as the original problem. Therefore, any feasible solution to the relaxation is also feasible for the original problem. In particular, given an optimal solution to the relaxation we can evaluate the expected profit of the associated set of items and derive a lower bound $\underline{z}_1$.

"Monaci et al." introduces a second lower bound for the 01-TB-KP given by the optimal objective value of the following binary quadratic problem.

$$\underline{z}_2 = max \quad (\sum_{j=1}^{n} p_j x_j)(1 - \sum_{j=1}^{n} q_j x_j) \tag{3.11}$$

$$s.t \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{3.12}$$

$$\underline{x} \in \{0,1\} \tag{3.13}$$

The following claim extends this result for the 01-TB-LIN:

Claim 3.2. *The optimal objective value $\underline{Z_2}$ of the following binary quadratic problem*

$$\underline{z}_2 = max \quad (\sum_{j=1}^{n} p_j x_j)(1 - \sum_{j \in T} q_j x_j) \tag{3.14}$$

$$s.t \quad A\underline{x} \leq \underline{b} \tag{3.15}$$

$$\underline{x} \in \{0, 1\} \tag{3.16}$$

is a lower bound for 2.5 - 2.8.

proof. Bound 3.14 bounds 2.5 from below since:

$$\prod_{j=1}^{n} \alpha_j = 1 - \mathbf{P}[\text{any item j explodes}] = 1 - \mathbf{P}[\bigcup_{j=1}^{n} \text{packed item j explodes}]$$

$$\geq 1 - \sum_{j=1}^{n} \mathbf{P}[\text{packed item j explodes}] = 1 - \sum_{j=1}^{n} q_j x_j$$

where the inequality follows from Boole's inequality. Then after linearizing model 3.14 - 3.16 the lower bound can be found relatively easily.

Chapter 4

Bounding the explosion probability

Maximizing the total expected profit might still lead to solutions where the probability that the knapsack explodes is relatively high, or higher than tolerable. Therefore, in this chapter we introduce an extension to the original 01-TB-KP. We add an additional constraint to the system that says that the total probability of exploding can not exceed a certain threshold $\beta \in [0, 1]$. Of course the probability that the knapsack does not explode is equal to the probability that none of the selected items explode. A model for the 01-TB-KP with probability constraint is thus:

$$max \quad \sum_{j=1}^{n} p_j x_j \tag{4.1}$$

$$s.t \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{4.2}$$

$$\prod_{j=1}^{n} \alpha_j \geq \beta \tag{4.3}$$

$$\alpha_j = 1 - q_j x_j, \qquad j \in \{1, ..., n\} \tag{4.4}$$

$$x_j \in \{0, 1\}, \qquad j \in \{1, ..., n\} \tag{4.5}$$

The objective function is once again simply the total profit of the selected items, so we no longer try to maximize the expected profit. However, constraint **??** has been added and requires the total explosion probability of the selected items to be larger than β since the product of α_j for $j \in \{1, ..., n\}$ is equal to the probability that none of the selected items will explode. This new constraint is quite difficult to deal with (for example its non-linear) so we will use Lagrangian Relaxation to find the optimal solutions or at least high quality approximations of optimal solutions.

4.1 Lagrangian Relaxation in general

Before we apply Lagrangian relaxation on our extended model, we first give a short overview of Lagrangian relaxation as a method to approximate a difficult optimization problem by a simpler problem. Given an optimization problem of the form

$$max_x \quad f(x) \tag{4.6}$$

$$s.t \quad g_i(x) \geq 0 \tag{4.7}$$

$$h_i(x) \geq 0 \tag{4.8}$$

$$x_i \in \{0, 1\} \tag{4.9}$$

where $g(x)$ represent the "easy" constraints, in most cases this means linear, and $h(x)$ represent the "hard" constraints (non-linear).

We move the "hard" constraints to the objective function to obtain the relaxed problem:

$$Z_L(\lambda) = max_x \quad f(x) + \lambda h(x) \tag{4.10}$$
$$s.t \quad g(x) \geq 0 \tag{4.11}$$
$$x_i \in \{0, 1\} \tag{4.12}$$

where $\lambda \geq 0$.

The optimal solution to the relaxed problem we denote by x^*_λ with corresponding optimal objective value $Z^*_R(\lambda)$. The idea is that for given $\lambda \geq 0$ we get penalized in the objective function if constraint 4.3 is violated, and rewarded if we satisfy the constraint. Furthermore, the relaxation is easy to solve since all the "hard" constraints have been removed.

A useful property of the Lagrangian relaxation of a problem is that solving it provides us with upper bounds to the original problem. Let x^* be the optimal solution to the original problem then for all $\lambda \geq 0$:

$$Z^*_R(\lambda) \geq f(\underline{x}) + \lambda g(\underline{x}) \geq Z^*$$

since x^* is a solution to the original problem we have $g(\underline{x}) \geq 0$ and $Z^* = f(x^*)$. Given that for each $\lambda \geq 0$ the Lagrangian relaxation provides an upper bound a natural question is which value of λ provides the best upper bound. If we define $P(\lambda)$ to represent 4.10 - 4.12 then the mathematical formulation of the above is as follows:

$$min_{\lambda \geq 0} \quad P(\lambda) \tag{4.13}$$

and is generally referred to as the Lagrangian dual (D). Thus a Lagrangian relaxation algorithm consists of exploring all possible values of λ and seeking the one that minimizes $P(\lambda)$. An important observation is that equation 4.10 is always convex no matter what the original problem is since it the supremum of a collection of linear functions in λ. Moreover, except for the points where the Lagrangian problem has multiple solutions it is differentiable everywhere.[3] Thus we make use of specific optimization techniques such as the subgradient method.

4.1.1 Subgradient method

Given $\lambda^{(k)}$ at iteration k the subgradient method gives us the means to calculate the new iterate $\lambda^{(k+1)}$ as follows:

$$\lambda^{(k+1)} = max\{0, \lambda^{(k)} - \psi_k((\prod_{j \in T} \alpha_j^{(k)}) - \beta)\} \tag{4.14}$$

where $\alpha_j^{(k)} = 1 - q_j x_j^{(k)}$ and $x_j^{(k)}$ are the timebomb items that correspond to the optimal solution $x^{(k)}$ of $Z_L(\lambda(k))$. And ψ_k is a properly selected step length.
A well-known formula that works in practice is given by:

$$\psi_k = \frac{\gamma_k(Z_L(\lambda_k) - Z^*)}{\|(\prod_{j \in T} \alpha_j^{(k)} - \beta)\|^2}, \tag{4.15}$$

where Z^* is the best known feasible solution to the original problem and $\gamma_k \in (0, 2]$ is usually set to 2 and then halved when $Z_L(\lambda_k)$ stays the same for several iterations. [2]

4.2 Lagrangian Relaxation applied

Now we are ready to apply Lagrangian relaxation to the proposed extension. For fixed $\lambda \geq 0$ we consider the relaxed version of 4.1 - 4.5:

$$Z_L(\lambda) = max \quad \sum_{j=1}^{n} p_j x_j + \lambda(\prod_{j=1}^{n} \alpha_j - \beta) \tag{4.16}$$

$$s.t \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{4.17}$$

$$\alpha_j = 1 - q_j x_j, \qquad j \in \{1, ..., n\} \tag{4.18}$$

$$x_j \in \{0, 1\}, \qquad j \in \{1, ..., n\} \tag{4.19}$$

and its dual:

$$\min_{\lambda} \quad \max_{x} \sum_{j=1}^{n} p_j x_j + \lambda(\prod_{j=1}^{n} \alpha_j - \beta) \tag{4.20}$$

$$s.t \quad \lambda \geq 0 \tag{4.21}$$

What is required at this point is an algorithm that implements the subgradient method to iteratively approaches the optimal value for λ and thus to either the optimal solution or to an high quality approximation and high quality bounds. Before the describing the algorithm lets define some variables to simplify the notation:

- $g_k = \prod_{j=1}^{n}(1 - q_j x_j) - \beta$: This is the subgradient associated to optimal solution x_{λ_k} at time-step k.

- $P(\lambda_k)$: Represents the model 4.16 - 4.19. So the statement solve $P(\lambda_k)$ implies solving the model 4.16 - 4.19 with $\lambda = \lambda_k$.

- $Z_L(\lambda_k)$: Represents the optimal value of the objective function 4.16 at time-step k.

- gap = $\frac{Zub - Zlb}{Zlb}$: The gap is defined by the best known upper bound minus the best known lower bound divided by the best known lower bound.

Algorithm 1 provides the pseudo-code for the algorithm that implements the subgradient method to solve the Lagrangian dual. In short, we start with an initial value for λ and solve the relaxed problem for this value. Based on the solution to the relaxed problem with this particular λ we can update the best known bounds and solutions. Furthermore, using the same solution we can find how we must update λ in order to get closer to the value for λ that is optimal for the dual problem.

Algorithm 1 Subgradient method applied

Input:
N: ▷ Maximum number of iterations
err: ▷ Target gap between upper and lower bound
Initialisation:
Gamma = 2: ▷ Constant in the time step 4.15
x0 = 0: ▷ Any feasible solution
Zlb = 0: ▷ Initial lower bound
Zub = ∞: ▷ Initial upper bound
current_best_sol = x0: ▷ Tracks best known solution
current_best_obj_val = Zlb: ▷ Tracks best known objective value
iterr = 0: ▷ Tracks amount of times the upper bound has not changed
gap = 1000: ▷ Tracks the gap between the best upper and lower bound
for k $\leq$N **do**
  if gap $\leq$err **then**
    return gap, best_current_sol ▷ Optimal gap achieved so the algorithm stops
  end if
Solve $P(\lambda_k)$: ▷ Solve the relaxed version of the problem at iteration k
Obtain $Z_L(\lambda_k)$
Obtain x_{λ_k}
Compute g_k
$\text{Zub}_{k+1} = min(Zub_k, Z_L(\lambda_k))$
  if $g_k \geq 0$ **then**
    $\text{Zlb}_{k+1} = max(Zlb_k, Z_L(\lambda_k) - (\lambda_k g_k))$
    if $Z_L(\lambda_k) - (\lambda_k g_k) \geq$ current_best_obj_val **then**
      current_best_sol = x_{λ_k}
      current_best_obj_val = $Z_L(\lambda_k) - (\lambda_k g_k)$
    end if
  else
    $\text{Zlb}_{k+1} = \text{Zlb}_k$
  end if

  if $\text{Zub}_{k+1} = \text{Zub}_k$ **then**
    iterr = iterr +1
  end if
  if Iterr = 4 **then**
    Gamma = 0.5 $\cdot$ Gamma

  end if
Compute Step: ▷ According to 4.15
$\lambda = \max(0, \lambda - step \cdot g_k)$
Compute gap_{k+1}
end for
Return gap, current_best_sol

This algorithm with the chosen heuristic 4.15 works on most instances, even if there are multiple constraints that need to be relaxed. However, since we are only relaxing one constraint there are methods that are less involved. For example we can use a dichotomy approach. In this approach the core of the algorithm stays the same, but the way we update λ is different. First we define:

- λ_min_feas: The smallest λ_k such that the optmial solution for the relaxation is feasible, i.e $g_k \geq 0$. Initially this is set to ∞

- λ_max_unfeas: The largest λ_k such that the optimal solution for the relaxation is infeasible, i.e $g_k < 0$. Initially this is set to 0.

Then algorithm 1 is used, but now we update λ_k as follows:

- If the optimal solution for the relaxation with λ_k is feasible then $\lambda\_min\_feas_{k+1} = \lambda_k$.
- If the optimal solution for the relaxation with λ_k is infeasible then $\lambda\_max\_unfeas_{k+1} = \lambda_k$.

Furthermore, λ_{k+1} is chosen as follows:

$$\lambda_{k+1} = \frac{1}{2}(\lambda\_min\_feas_{k+1} + \lambda\_max\_unfeas_{k+1})$$

The idea is that whenever λ_k provides a feasible solution then $\lambda_k \geq \lambda^*$, where λ^* is the optimal value to the dual problem. If λ_k provides a solution that is infeasible for the original problem then $\lambda_k \leq \lambda^*$. [4] This way the interval in which the optimal solution to the dual problem lies keeps shrinking so with each iteration we get closer to the optimal solution.

Chapter 5

Computational experiments

The algorithms described in Chapter 4 where implemented in python using the Pyomo environment [1]. In both algorithms at each iteration the Lagrangian relaxation must be solved for a specific value of λ. Since the objective function contains a non-linear term we must use a non-linear solver. For this the open-source solver "Couenne" is used [1]. Both algorithms are iterative and are thus sensitive to a starting point. It turned out that a combination of both algorithms worked best: Algorithm 1 was used until we had found a value $\lambda = y$ such that the optimal solution corresponding to the relaxed problem $P(y)$ is feasible. Then this point is used as the starting point for the dichotomy approach. Unless explicitly specified, all the experiments were executed on a Hp Zbook studio G4 with an Intel Core i7-7700HQ processor running at 2.7GHz. Sadly the fan system inside the computer was defect so overheating would happen very quickly which did not have improve the performance. All relevant codes will be in the appendix

5.1 Instance generation

To see how the algorithms perform we generate instances with different characteristics. First the weights, profits and capacity for a regular 01-Knapsack Problem are generated based on the *hard instances* introduced by Pisinger [**?**]. From these instances we consider the following with $R = 1000$:

1. *Uncorrelated instances*: The profits p_j and w_j are both randomly chosen in $[1, R]$ so there is no correlation between the profit and the weight of an item.

2. *Strongly correlated instances*: Weights w_j are chosen randomly in $[1, R]$ and the profits $p_j = w_j + R/10$.

Then given the proportion of time-bomb items $B \in [0, 1]$ we first determine the $\lceil nB \rceil$ items with the highest profit to be the set of time-bomb items. Next for each instance we consider two cases to generate the explosion probabilities for each item. For these two cases we use class 1 and 3 as described in "Monaci" et al. [10]. Unless otherwise specified the threshold for the explosion probability is always $\beta = 0.9$. In addition, unless otherwise specified the capacity is set to half of the total weights of all the items. The exact way the classes define the explosion probabilities can be seen in "Monaci", but below the general idea of the two classes is given:

- Class 1 generates the probabilities such that each item has a relatively low probability of exploding ($q_j \leq 0.1$), which would represent the most realistic scenarios. Additionally, the probablities are proportional to the profits so that the higher the profit the higher the probability of an explosion.

- Class 2 generates the probabilities according to a beta distribution. This ensures that the probabilities are relatively low, but are in this case not correlated with the profits nor the weight of an item.

[1] See http://www.pyomo.org/ for more information.

5.2 Results

		$B = 0.2$			$B = 0.4$			$B = 0.6$			$B = 0.8$		
Class:	Size:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:
Class 1	25	0.0000	0.2837	1	0.0000	0.4662	0.9430	0.0332	2.0490	0.9103	0.0194	7.4728	0.9040
	50	0.0000	1.6049	1	0.0000	2.0866	0.9033	0.0097	12.9382	0.9068	0.0498	21.8239	0.9124
	75	0.0000	0.5947	1	0.0106	8.0483	0.9151	0.0095	36.8290	0.9041			
	100	0.0000	2.9856	1	0.0274	11.020	0.9229	0.0094	50.9348	0.9056			
Class 2	25	0.0000	1.1449	1	0.0075	2.7606	0.9711	0.0459	9.0039	0.9122	0.0357	3.5697	0.9107
	50	0.0000	1.7882	1	0.0029	3.7190	0.9118	0.0162	14.2448	0.9078	0.0138	8.1688	0.9036
	75	0.0000	6.7489	1	0.0183	11.8099	0.9112	0.0507	25.0648	0.9258			
	100	0.0000	14.7538	1	0.0084	19.5473	0.9180	0.0283	91.7629	0.9209			

Table 5.1: Strongly correlated instances

Table 2: Uncorrelated instances													
		$B = 0.2$			$B = 0.4$			$B = 0.6$			$B = 0.8$		
Class:	Size:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:
Class 1	25	0.0313	1.9641	0.9237	0.0562	2.2062	0.9245	0.0812	3.8810	0.9198	0.3385	8.3268	0.9417
	50	0.0143	3.0317	0.9118	0.0297	5.4945	0.9187	0.02539	13.9139	0.9095	0.1949	50.5860	0.9378
	75	0.0102	4.5382	0.9111	0.0451	15.8779	0.9351	0.0235	59.1244	0.9081	0.7902	768.632	0.9693
	100	0.0106	9.0623	0.9166	0.0126	35.3889	0.9142	0.0015	42.5616	0.9008			
Class 2	25	0.0315	1.397	0.9240	0.0069	0.4823	0.9054	0.0471	1.8332	0.9199	0.0873	1.5015	0.9182
	50	0.0088	1.0787	0.9082	0.0518	2.6474	0.9225	0.0252	2.4930	0.9067	0.0957	6.3301	0.9191
	75	0.0275	3.9734	0.9392	0.0033	5.1310	0.9024	0.0138	4.1470	0.9038	0.0150	123.013	0.9072
	100	0.0265	9.1156	0.9385	0.0095	7.6656	0.9096	0.0593	28.3152	0.9171			

Table 5.2: Uncorrelated instances

		$B = 0.2$			$B = 0.4$			$B = 0.6$			$B = 0.8$		
Class:	Size:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:
Class 1	50	0.0000	3.1787	0.9906	0.0171	7.1359	0.9169	0.0385	19.9435	0.9147	0.0638	158.430	0.9147
Class 2	50	0.0000	1.1295	0.9999	0.0125	1.7577	0.9327	0.0401	6.1928	0.9169	0.0456	9.6696	0.9116

Table 5.3: Strongly correlated instances averaged

		$B = 0.2$			$B = 0.4$			$B = 0.6$			$B = 0.8$		
Class:	Size:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:	Gap:	Time:	NEP:
Class 1	50	0.0182	2.2387	0.9293	0.0259	3.4875	0.9162	0.0440	16.6090	0.9202	0.2357	85.9549	0.9259
Class 2	50	0.0273	2.1694	0.9280	0.0237	3.8163	0.9105	0.0356	5.9299	0.9138	0.0978	9.7614	0.9183

Table 5.4: Uncorrelated instances averaged

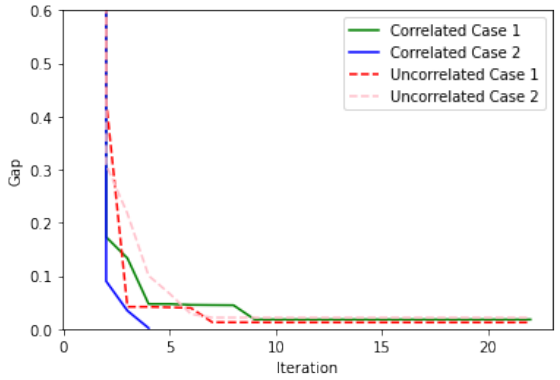

Figure 5.1: Speed of convergence for N = 100, B=0.6

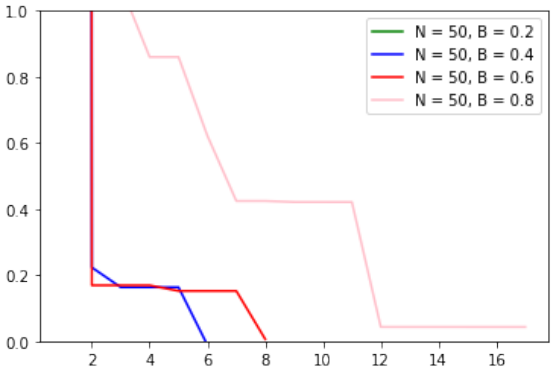

Figure 5.2: Speed of convergence for varying B

Figure 5.3: Speed of convergence

5.2.1 Interpretation of results

The algorithms were run on all the instances described above and the results can be seen in table 5.1 and table 5.2. It is important to note that when running the algorithm the variation was rather large, for example for two generated instances with the same number of items the algorithm could take a vastly different amount of iterations to arrive at the solution. Therefore the amount of time the algorithm takes to run two instances of the same size could differ significantly. In addition the performance of the laptop must be factored in. However, there are still some general patterns that can be extracted from table 5.1 and 5.2. First of all there seems to be a correlation between how low the gap was and how close the non-explosion probability was to the desired threshold for instances where there were a significant number of time-bomb items in the item pool (i.e not $B = 0.2$). This seems reasonable since in all instances the time-bomb items are the items with the largest profit, so it seems natural that the optimal solution would have as many time-bomb items as possible which ensures that the constraint is only just satisfied. In terms of accuracy the algorithm seems to perform reasonably for most instances, with one notable exception. In the case $B = 0.8$ for the uncorrelated instances the gaps are significantly higher than the others. This seems to be due to the fact that the proportion of time-bomb items is quite high, since we can also see that in general as the proportion of time-bomb items increases, the accuracy decreases. As a matter of fact a similar story holds for the time usage: As the proportion of time-bomb items increases, the time usage also increases.

In terms of time usage the algorithm performs quite poorly. For most instances 100 items seemed to be the limit that if crossed the laptop would simply crash, but for some instances with a high proportion of time-bomb items even that was unreachable.

This could be due to a couple of factors, but the most likely one is the programming language. Pyhton is known to be significantly slower than for example C so for any future work a switch in programming language is recommended.

In table 5.3 and table 5.4 one can see the average results for when the algorithm was run on 20 different cases of the same class and instance. The same general patterns discussed before can be seen here: as the proportion of time-bomb items increases the time usage increases and the accuracy slightly decreases. In addition, we again see that the accuracy for the uncorrelated instances with $B = 0.8$ is significantly worse than for the other cases. Furthermore, one can now clearly see that for both instances class 1 takes significantly longer to complete than class 2.

In figure 5.1 the speed of convergence for a single run can be seen for all instances and classes with $N = 100$ and $B = 0.6$. As can be seen it takes less than 10 iterations for all cases to converge to their respective optimal gaps. This was almost always the case for any N and B with the exception of the case $B = 0.8$ which could sometimes take significantly longer. In figure 5.2 the speed of convergence for strongly correlated instances with probabilities from class 1 are shown for different values of B. This figure seems to corroborate the data from the tables that the higher the proportion of time-bomb items the longer it takes for the algorithm to complete.

Chapter 6

Conclusions

In this paper we studied the 01-Timebomb-Knapsack problem, which is a stochastic version of the 0-1 Knapsack Problem in which, in addition to a profit and weight, each item has an additional parameter which represents the probability of exploding. We studied the related literature which introduced several upper and lower bounds for the 01-TB-KP and showed that these were also valid and usable in a more general setting. We then consider a variant of the 01-TB-KP in which we add an additional constraint bounds the probability that the knapsack explodes. We present an algorithm based on Lagrangian Relaxation that yields approximations of optimal solutions for this variant and evaluate it's performance on a set of benchmark instances. The computational experiments show that for a relatively low proportion of time-bomb items this algorithm performs reasonably well, but the accuracy deteriorates the higher the proportion of time-bomb items get. In addition, for now the algorithm is quite slow and can thus only be used instances with a small number of items.
Future work should focus on improving the accuracy and speed of the algorithm. For example, we could study the effect of using a different solver that solves the Lagrangian relaxation at each iteration in the algorithm. In this report the solver open-source "Couenne" was used, but better and faster solvers do exist such as "Baron" [7]. Furthermore, the computational analysis showed that there is a correlation between the optimality of the solution and how close the non-explosion probability was to the threshold. This implies that we might be able to split the solving process in two smaller problems to potentially make the algorithm faster:
- First we only consider the time-bomb items and maximize the profit without exceeding the probability constraint.
- Then we subtract the capacity used by the time-bomb items we chose from the total capacity and consider the deterministic knapsack problem with the remaining capacity and the left over deterministic items. Finally, in the future it would be better to implement this algorithm in a different programming language than python in order to improve performance.

Bibliography

[1] Pietro Belotti. couenne: a user's manual.

[2] Marshall L Fisher. An applications oriented guide to lagrangian relaxation.

[3] Marshall L Fisher. Ten most influential titles of "management science's" first fifty years. *Source: Management Science*, 50:1861–1871, 2004.

[4] C Deperrois Grisoni, D Galley, G Boussac, and P Van Hentenryck. Lagrangian relaxation can solve your optimization problem much, much faster.

[5] Mordechai I. Henig. Risk criteria in a stochastic knapsack problem. *Operations Research*, 38(5):820–825, 1990.

[6] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer Berlin Heidelberg, 2004.

[7] Jan Kronqvist, David E Bernal, Andreas Lundell, and Ignacio E Grossmann. A review and comparison of solvers for convex minlp. 2018.

[8] Diego Lisbona and Timothy Snee. A review of hazards associated with primary lithium and lithium-ion batteries. *Process Safety and Environmental Protection*, 89(6):434–442, 2011. Special Issue: Centenary of the Health and Safety Issue.

[9] Yasemin Merzifonluoğlu, Joseph Geunes, and H. Edwin Romeijn. The static stochastic knapsack problem with normally distributed item sizes. *Mathematical Programming*, 134(2):459 – 489, 2012. Cited by: 21.

[10] Michele Monaci, Ciara Pike-Burke, and Alberto Santini. Exact algorithms for the 0–1 time-bomb knapsack problem. 2021.

[11] David Pisinger. Where are the hard knapsack problems? *Computers and Operations Research*, 32:2271–2284, 2005.

Appendix A

Relevant code

A.1 Instance generation

*

```
#Strongly correlated instances with probabilities from class 1

B = 0.8 #Proportion of items that are timebomb items.
n = 50 #mber of items.

T = math.floor(B*n) #number of timebomb items.
R = 1000 #Range for the weights.
#Generating data: Strongly correlated instances: Weights wj are distributed in [1;
    R] and p j = wj + R/10
w_j = np.random.randint(1,R,n)
w_j = -np.sort(-w_j)
p_j = w_j + R/10

#Defining the explostion probabilities according
#to case 1 in the paper.

tbitems = p_j[0:T-1]
detitems = p_j[T:n]
pmax = max(tbitems)
pmin = min(tbitems)
pp = max(detitems)
q_j = 0.1*((p_j-pp)/(pmax-pp))
q_j[T+1:n] = 0
c = np.sum(w_j)/2  #Capacity (For now set to total weight/2)
L=1 #lambda
x0 = np.zeros(n)#initial solution
beta = 0.9 #Treshold for explosion probability
```

```
#Strongly correlated instances with probabilities from Class 2
B = 0.2 #Proportion of items that are timebomb items.
n = 50 #mber of items.

T = math.floor(B*n) #number of timebomb items.
R = 1000 #Range for the weights.
#Generating data: Strongly correlated instances: Weights wj are distributed in [1;
    R] and p j = wj + R/10
w_j = np.random.randint(1,R,n)
w_j = -np.sort(-w_j)
p_j = w_j + R/10

tbitems = p_j[0:T-1]
detitems = p_j[T:n]
q_j = np.random.beta(1,10,n)
q_j[T+1:n] = 0
```

```
c = np.sum(w_j)/2
L = 1
x0 = np.zeros(n)
beta = 0.9 #Treshold for explosion probability
```

```
#Class 1 uncorrelated
B = 0.6 #Proportion of items that are timebomb items.
n =75 #mber of items.

T = math.floor(B*n) #number of timebomb items.
R = 1000 #Range for the weights.

w_j = np.random.randint(1,R,n)

p_j = np.random.randint(1,R,n)
p_j = -np.sort(-w_j)



tbitems = p_j[0:T-1]
detitems = p_j[T:n]
pmax = max(tbitems)
pmin = min(tbitems)
pp = max(detitems)
q_j = 0.1*((p_j-pp)/(pmax-pp))
q_j[T+1:n] = 0
c = np.sum(w_j)/2  #Capacity (For now set to total weight/2)
L=1
x0 = np.zeros(n)
beta = 0.9 #Treshold for explosion probability
```

```
#Class 2 uncorrelated
B = 0.6 #Proportion of items that are timebomb items.
n = 100 #mber of items.

T = math.floor(B*n) #number of timebomb items.
R = 1000 #Range for the weights.
#Generating data: Strongly correlated instances: Weights wj are distributed in [1;
    R] and p j = wj + R/10
w_j = np.random.randint(1,R,n)
p_j = np.random.randint(1,R,n)
p_j = -np.sort(-w_j)


tbitems = p_j[0:T-1]
detitems = p_j[T:n]
q_j = np.random.beta(1,10,n)
q_j[T+1:n] = 0
c = np.sum(w_j)/2
L = 1
x0 = np.zeros(n)
beta = 0.9 #Treshold for explosion probability
```

A.2 The model

```
#The model in pyomo
opt = SolverFactory('Couenne')
model = pyo.ConcreteModel()
model.x = pyo.Var(range(n), within=pyo.Binary)
model.P = pyo.Param(initialize = L,mutable = True)
alphas = 1-np.multiply(q_j,model.x)
model.value = pyo.Objective(
```

```
expr = (sum(p_j[i]*model.x[i] for i in range(n)) + model.P*(pyo.prod(alphas[i] for
    i in range(n))-beta)) ,
sense = pyo.maximize)

model.weight = pyo.Constraint(
expr = sum(w_j[i]*model.x[i] for i in range(n))<=c)
```

A.3 The algorithm

```
#The first part to find the starting lambda
start1 = time.time()
N = 100
gamma = 2
x0 = np.zeros(n)
x0[n-1]=1
x1 = np.ones(n)
Zlb = np.dot(p_j,x0)
Zlb0 = Zlb
Zub = np.dot(p_j,x1)+1
current_best_sol = x0
current_best_obj_val = Zlb
iterr = 0
err = 0.01
gap = 10
for i in range(N):
    print(i)
    if gap < abs(err):
        break
    result = opt.solve(model)
    a = model.x.extract_values()
    new_lis = list(a.values())
    x_vals = np.array(new_lis)

    #print(x_vals)
    obj_val = model.value()
    #print(obj_val)
    alp = 1-np.multiply(q_j,x_vals)
    #print(alp)
    gk = prod(alp) - beta
    #print(gk)
    Zub_old = Zub
    Zub = min(Zub_old, obj_val)
    if gk>=0:

        Zlb = max(Zlb, obj_val-(L*gk) )
        if (obj_val-(L*gk))>= current_best_obj_val:
            current_best_sol = x_vals
            current_best_obj_val = obj_val-(L*gk)
        model.P = L
        break
    if Zub == Zub_old:
            iterr +=1
    if iterr == 4:
        iterr = 0
        gamma = gamma/2
    gap = (Zub-Zlb)/Zlb
    #print(gap)
    #gaps44.append(gap)
    #Compute step size here
    step = (gamma *(obj_val-Zlb))/np.linalg.norm(gk)
    #print(current_best_sol)



    L = max(0,L - step*gk)
```

```
    #print(L)
    model.P = L
#print(np.dot(w_j,x_vals), current_best_sol)
end1 = time.time()
totaltime1 = end1 - start1
```

```
#The dichotomy approach
start2 = time.time()
N = 100

L_min_feas = np.inf
L_max_unfeas = 0
err = 0.01
gap = 10
for i in range(N):
    print(i)
    if gap < abs(err):
        break
    if abs(L_min_feas-L_max_unfeas)<1:
        break
    result = opt.solve(model)
    a = model.x.extract_values()
    new_lis = list(a.values())
    x_vals = np.array(new_lis)
    print(x_vals)
    obj_val = model.value()
    #print(obj_val)
    alp = 1-np.multiply(q_j,x_vals)
    #print(alp)
    gk = prod(alp) - beta
    print(gk)
    Zub_old = Zub

    if gk>=0:
        Zub = min(Zub_old, obj_val)
        L_min_feas = L

        Zlb = max(Zlb, obj_val-(L*gk) )
        if (obj_val-(L*gk))>= current_best_obj_val:
            current_best_sol = x_vals
            current_best_obj_val = obj_val-(L*gk)
    if gk<0:
        L_max_unfeas = L
    if Zub == Zub_old:
            iterr +=1
    if iterr == 4:
        iterr = 0
        gamma = gamma/2
    gap = (Zub-Zlb)/Zlb
    print(gap)
    #gaps44.append(gap)
    #Compute step size here
    step = (gamma *(obj_val-Zlb))/np.linalg.norm(gk)
    print(current_best_sol)


    print(L_min_feas,L_max_unfeas)
    L = 0.5*(L_max_unfeas+L_min_feas)
    print(L)
    model.P = L
#print(np.dot(w_j,x_vals), current_best_sol)

end2 = time.time()
totaltime2= end2 - start2
```