The Knapsack Problem, Cryptography, and the Presidential Election

by

Brandon S. McMillen

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Mathematics

Program

YOUNGSTOWN STATE UNIVERSITY

May, 2012

The Knapsack Problem, Cryptography, and the Presidential Election

Brandon S. McMillen

Signature:

Brandon S. McMillen, Student   Date

Approvals:

Dr. Nathan Ritchey, Thesis Advisor   Date

Dr. Jozsi Jalics, Committee Member   Date

Dr. Jacek Fabrykowski, Committee Member   Date

Peter J. Kasvinsky, Dean of School of Graduate Studies & Research   Date

ABSTRACT

The 0-1 Knapsack Problem is an $\mathcal{NP}$-hard optimization problem that has been studied extensively since the 1950s, due to its real world significance. The basic problem is that a knapsack with a weight capacity c is to be filled with a subset of n items. Each item i, has a weight value w_i and a profit value p_i. The goal is to maximize total profit value without the having the total weight exceed the capacity.

In this thesis, the 0-1 Knapsack Problem is introduced and some of the research and applications of the problem are given. Pisinger's branch-and-bound algorithm that will converge to an optimal solution is presented. One of the earliest applications of the knapsack problem, the knapsack cryptosystems, is then discussed. The earliest knapsack cryptosystem, the Merkle-Hellman Cryptosystem, is described along with how Adi Shamir broke this cryptosystem. Generating functions are then used to provide a number of solutions to a knapsack problem. Using the generating function of the knapsack problem, the paper concludes with an application on the Electoral College.

Contents

1 0-1 Knapsack Problem

The family of knapsack problems requires that a subset of some given items be chosen such that a profit sum is maximized without exceeding the capacity of the particular knapsack. There are many different types of knapsack problems, but for this thesis we are going to discuss the *0-1 Knapsack Problem*. The *0-1 Knapsack Problem* is the problem of choosing a subset of n items, where each item has a profit p_i and a weight w_i, which will fit into a knapsack with capacity c. The goal is to maximize the profit sum without having the weight sum exceed the capacity c. This problem is formulated as the following maximization problem:

$$\begin{aligned} \text{maximize} \quad & z = \sum_{i=1}^{n} p_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \leq c \\ & x_i \in \{0,1\}, \quad i = 1, \ldots, n, \end{aligned} \tag{1.1}$$

where x_i equals 1 if item i is chosen to be in the knapsack, and 0 if it is not.

A particular case of the 0-1 knapsack problem arises when $p_i = w_i$. The objective of the problem is to find a subset of the weights that is closest to c without exceeding it. This problem is generally referred to as the *Subset-Sum Problem*. This problem is related to the diophantine equation

$$\begin{aligned} & \sum_{i=1}^{n} w_i x_i = \hat{c} \\ & \text{where } x_i \in \{0,1\}, \quad i = 1, \ldots, n, \end{aligned} \tag{1.2}$$

such that the optimal solution to the *Subset-Sum Problem* is the largest $\hat{c} \leq c$ for which (1.2) has a solution.

The Knapsack Problem has been studied extensively due to its simple structure. In some sense, knapsack problems are among the simplest integer programming problems, and also appear as subproblems in many complex programs and integer programming algorithms. In the 1950's, Bellman's [2] dynamic programming theory produced the first algorithm to exactly solve the knapsack problem. In 1957, Danzig [5] studied the knapsack problem and created a method to determine an upper bound on z that was used in various studies of the knapsack problem. The first branch and bound algorithm was derived in 1967 by Kolesar [11]. The first polynomial-time approximation algorithm was proposed by Johnson [8] in 1974. The main results from the eighties dealt with creating algorithms for large-sized problems. Throughout the years, the knapsack problem has been vastly studied, including quite recently in 2010 by Howgrave-Graham and Joux [7]. They developed an algorithm for knapsack problems that runs in the smallest time found.

The Knapsack Problems also have been vastly studied for the past fifty years due to their immediate applications in industry and financial management. Some of these applications include capital budgeting, selection of financial portfolios, cargo loading, the cutting stock problem, and the bin-packing problem. Also, the knapsack problem has been used to solve many combinatorial and optimization problems, such as a subproblem in many famous optimization problems including the traveling sales-person problem. They were also used in the first cryptosystems that we will discuss later on in this thesis. Knapsack problems can also be used to predict which presidential candidate will win an election, which we will show later in this thesis. The following example demonstrates a simple application of the Knapsack Problem.

Example 1.1 *Suppose you want to invest $c = \$20$ and you are considering 5 investment opportunities. Investment 1 costs \$8 with a return of \$10, investment 2 costs \$9 with a return of \$10, investment 3 costs \$3 with a return of \$5, investment 4 costs \$5 with a return of \$2, investment 5 costs \$7 with a return of \$9, and investment 6 costs \$3 with a return of \$1. How can you maximize your the return on your investment without spending more than c dollars?*

This problem can be modeled as a knapsack problem as follows:

$$\begin{aligned} \text{maximize} \quad & z = 10x_1 + 10x_2 + 5x_3 + 2x_4 + 9x_5 + x_6 \\ \text{subject to} \quad & 8x_1 + 9x_2 + 3x_3 + 5x_4 + 7x_5 + 3x_6 \leq 20 \\ & x_i \in \{0,1\}, \quad i = 1, \ldots, 6, \end{aligned} \tag{1.3}$$

where $x_i =$ investment i and equals 1 if investment i is chosen to be in the knapsack, and 0 if it is not.

The above example is so small that we are able to solve the problem by exhaustive search or brute force. However, on larger problems enumerating every solution is quite time consuming and not computationally feasible. In each problem, there are 2^n possible solutions. So, although the 0-1 Knapsack Problem has a very simple structure, it cannot, as of yet, be solved in a time bounded by a polynomial. The 0-1 Knapsack problem is known as an $\mathcal{NP}-$hard problems, meaning that it is very unlikely that researchers will ever devise polynomial algorithms to solve all instances of this problem.

The following proof of 0-1 Knapsack Problem being $\mathcal{NP}-$hard is adapted from Martello and Toth [13]. In order to show that a problem is $\mathcal{NP}-$hard, we show that for each problem P, its *recognition version*, $R(P)$, is $\mathcal{NP}-$ hard. We do this by showing that a basic $\mathcal{NP}-$hard problem can be polynomially transformed into $R(P)$. We will use the following basic $\mathcal{NP}-$hard problem, known as the Partition Problem, which is originally treated by Karp [10]. The Partition Problem is described as follows: Given n positive integers $w_1, \ldots, w_n$, is there a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i = \sum_{i \notin S} w_i$?

Theorem 1.1 *0-1 Knapsack Problem is $\mathcal{NP}-$hard.*

Proof Consider the recognition version of the Subset-Sum Problem, i.e.: given $n+2$ integers $w_1, \ldots, w_n, c$, and a, is there a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} w_i \leq c$ and $\sum_{i \in S} w_i \geq a$?

By setting $c = a = \sum_{i \in S} w_i/2$, any instance of Partition Problem can be polynomially transformed into an instance of the recognition version of the Subset-Sum Problem. Thus, the Subset-Sum Problem is $\mathcal{NP}-$hard. And since the Subset-Sum Problem is a particular case of the 0-1 Knapsack Problem when $p_i = w_i$ for $i \in \{1, \ldots, n\}$, the 0-1 Knapsack Problem is $\mathcal{NP}-$hard. $\square$

In the next section, we provide an algorithm to solve these problems. Note that numerous methods have been developed to solve instances of 0-1 Knapsack Problems. This algorithm is a branch-and-bound algorithm and is used because it helps us converge to an optimal solution more efficiently without having to always enumerate all possible solutions. Also, this algorithm is used, since it shows us certain attributes the 0-1 knapsack problem possesses.

1.1 Branch-and-Bound Algorithm

1.1.1 Introduction

Since Knapsack problems are $\mathcal{NP}-$ hard, we do not know any other exact solution techniques than completely checking each possible solution to see which is the optimal. However, we can reduce computational time by using a *branch-and-bound* algorithm. A *branch-and-bound* algorithm is a search method where one has a way of computing a lower bound on an instance of the problem and a way to divide the problem into feasible regions to create smaller subproblems. Then, the problem is analyzed in a systematic. A *branch-and-bound* algorithm is basically a complete enumeration, but we find nodes that cannot lead to an improved solution, thus reducing computational time.

Before proceeding with the description of the algorithm, let us define some terms first.

Definition 1.1 *The efficiency, e_i, of a knapsack item i is p_i/w_i, where p_i is the profit of item i and w_i is the weight of item i.*

Definition 1.2 *The profit sum $\bar{p}_i$ and the weight sum $\bar{w}_i$ of items up to i is defined as*

$$\bar{p}_i = \sum_{j=1}^{i} p_j, \quad i = 0, \ldots, n,$$

$$\bar{w}_i = \sum_{j=1}^{i} w_j, \quad i = 0, \ldots, n.$$

Definition 1.3 *When we order a knapsack from greatest efficiency to the smallest efficiency (i.e. $e_i > e_j$ when $i < j$) and begin to fill the knapsack in a greedy way (i.e. items $i = 0, \ldots, n$ are put into the knapsack as long as $w_i \leq c - \bar{w}_{i-1}$), the first* *item b that cannot fit into a the knapsack, where $\bar{w}_{b-1} \leq c < \bar{w}_b$, is called the break item.*

Definition 1.4 *The break solution $x' = \{x'_1, \ldots, x'_n\}$ is the solution which occurs when setting $x'_i = 1$ for $i = 0, ..., b-1$ and $x'_i = 0$ for $i = b, \ldots, n$.*

Also, for our algorithm, we will use the *Danzig bound*. This is an upper bound on the solution of the knapsack problem that Danzig showed by linear relaxation. The bound is

$$u = \left\lfloor \bar{p}_{b-1} + \frac{rp_b}{w_b} \right\rfloor . \tag{1.4}$$

Pisinger[15] performed a study that provided properties of solutions to *0-1 Knapsack Problems* that will help in our branch-and-bound algorithm. He discovered that in most solutions of the knapsack problem, only a few variables far from b differed from the break solution, i.e. most solutions that differed from the break solution contained items that were generally close to the break item b. He noted that

- The branching tree of a branch-and-bound algorithm should start with the break solution and gradually enumerate items from b outwards in a systematic way. This ensures that the algorithm will ensure fast convergence to the optimal solution.

- Small-weighted items are used for achieving a filled knapsack, so we should choose branching variables that will fill the knapsack.

These will be evident in the following algorithm.

1.1.2 Pisinger's Algorithm

In order to start Pisinger's Algorithm, we must first find the break item b. Knowing this break item, we are able to calculate the *Dantzig [5] bound* and the break solution, which will give us a pretty good lower bound to start our branch-and bound algorithm. Knowing this bound also allows us to reduce the size of the problem by fixing some variables x_i at their optimal value. To find the break item b, we order the *efficiencies* e_i of each item largest to smallest, i.e. $e_i \geq e_j$ where $i < j$. Then we find b by determining where $\bar{w}_{b-1} \leq c < \bar{w}_b$ by filling the knapsack using a greedy method. Thus, b would be the break item where $e_i \geq e_b$ for $i = 1, \ldots, b-1$ and $e_i \leq e_b$ for $i = b-1, \ldots, n$. When we have the break item b, we can now determine the break solution, $x' = (x'_1, \ldots, x'_n)$, which we will use as our lower bound and initial solution to the branch-and-bound algorithm.

To systematically enumerate items from b, we will use a recursive algorithm. Thus, we will either insert or remove one item of the knapsack. At each recursion s and t, where $s < b \leq t$, indicates that the variables x_i, where $i \in [s+1, t-1]$, are fixed to some value, while the other variables may vary arbitrarily as we begin to add and remove items. So to gradually expand from b, we first set $[s, t] = [b-1, b]$.

Then at any stage, we calculate the profit sum $\bar{p}$ and the weight sum $\bar{w}$ for our current value of the solution vector $x = (x_1, \ldots, x_n)$. Note that initially our solution will be the break solution x'. Then the succeeding iteration will try to make $\bar{w}$ as close to the capacity c as possible. That is, if $\bar{w} \leq c$, we will insert each item, one at a time, $i \geq t$, or if $\bar{w} > c$, we will remove an item $i \leq s$ from the knapsack. If we inserted an item i, set $t = i + 1$, or if we removed an item i, set $s = i - 1$. Doing this assures we only insert items after i or remove items before i.

We backtrack the algorithm if the upper bound, u, does not exceed z, where $z =$ current $\bar{p}$, i.e. if

$$u = \left\lfloor \bar{p} + \frac{(c - \bar{w}p_t)}{w_t} \right\rfloor \leq z, \quad \text{when} \quad \bar{w} \leq c. \tag{1.5}$$

$$u = \left\lfloor \bar{p} + \frac{(c - \bar{w}p_s)}{w_s} \right\rfloor \leq z, \quad \text{when} \quad \bar{w} > c, \tag{1.6}$$

That is if our upper bound u does not exceed z, we cannot improve z on the current branch and we must backtrack to try and improve our z. A problem occurs if no bound stops the branching, i.e. s may grow below 1 or t may grow above n. To prevent this from happening, we enter stop items, 0 and $n+1$, where $(p_0, w_0) = (1, 0)$ and $(p_{n+1}, w_{w+1}) = (0, 1)$. This will ensure that if $s = 0$ or $t = n + 1$, the upper bounds will be so bad it will force us to backtrack.

Throughout the algorithm, we do not need to keep track of our current solution vector x, we only keep track of the items i that differ from the break solution x', items where $x_i \neq x'_i$. We add these items to an *exception list* E each time an improved solution is found. That is, if we add an item i at a stage and (1.5) is true, we set $E = E \cup \{i\}$, which means our solution vector x is different from x' by including item i. Similarly, $E = E \cup \{i\}$ if we remove item i. Thus, at the end of the algorithm, we will have our list E that we use to add and remove variables from x' accordingly.

If we find an optimal solution x, no bounding will stop the iteration. The process will only stop if one of the following occurs:

$$E = \emptyset, \tag{1.7}$$

$$\text{or} \quad \bar{w} > c \quad \text{and} \quad i \in E \neq \emptyset : i \geq b, \tag{1.8}$$

$$\text{or} \quad \bar{w} \leq c \quad \text{and} \quad i \in E \neq \emptyset : i < b. \tag{1.9}$$

If (1.7) occurs, we cannot add and subtract anymore items from our break solution, thus we have reached the optimal solution. If (1.8) occurs, x is not a solution since our capacity has been exceeded. If (1.9) occurs, it implies that x is not the optimal solution, since it can be improved by setting $x_i = 1$ for remaining $i \in E$.

The following example will demonstrate the algorithm.

Example 1.2 *In this example, we refer back to Example 1.1. In order to find the break item b and the break solution x', we order the knapsack items according to their efficiencies. This is shown in the following table. Also, start and stop items are added as discussed previously.*

i	0	1	2	3	4	5	6	7
p_i	1	5	9	10	10	2	1	0
w_i	0	3	7	8	9	5	3	1

In this example, we find that $b = 4$ and $x' = (111000)$. Then as the algorithm states, we initially set $(\bar{p}, \bar{w}) = (24, 18)$, which is what the break solution produces, and set $(s, t) = (3, 4)$. Since our capacity is $c = 20$, we see that we need to add a knapsack item since $\bar{w} < c$. In Figure 1, we see that if we add knapsack item 5 and 6 that the bound u does not improve on our solution. However, when knapsack item 4 is added, $u = 26$. So, we calculate our new $(\bar{p}, \bar{w}) = (34, 27)$ with item 4 added to the solution. Then we check (1.5) and see that it is true, as shown in Figure 1. So, we set $E = E \cup \{4\}$ and continue to branch. Next, we would need to subtract an item since $\bar{w} > c$, as shown in Figure 1. So, continuing in this manner, we find that our optimal

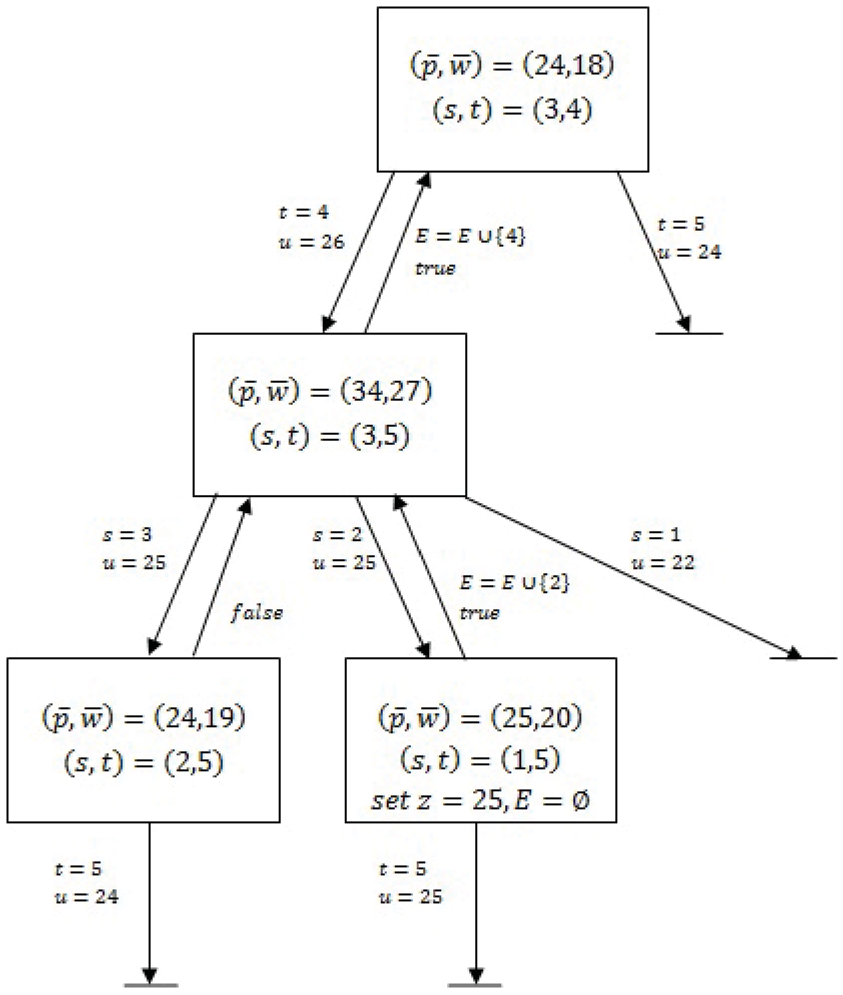

Figure 1: Branching tree for Example 1.2

solution is $z = 25$. In order to find our optimal solution vector x, we look at our set E and change the x' accordingly. Figure 1 shows us that $E = \{2, 4\}$. The 2 represents we removed knapsack item 2 and the 4 represents when we added knapsack item 4 as indicated in Figure 1 on whether we added item i or removed item i. Thus, our optimal solution vector $x = (101100)$. Note that the arrows that point up on Figure 1 refers to when we check if we found an improved solution or not. Also, notice that when initially adding item $t = 4$, then removing item $s = 3$, and finally adding item $t = 5$ to the knapsack, we see that our u does not exceed our current z, so we must backtrack the algorithm to eventually get to our optimal solution that we found.

Although this algorithm helps to converge to an optimal solution efficiently, we may often come across problems where a complete enumeration of the branching is needed. This is due to the fact the Knapsack problem is $\mathcal{NP}$-hard.

2 Cryptosystems

Included in the many applications of the knapsack problem is the ability to create cryptosystems. Some of the earliest public key cryptosystems were ones formed from the knapsack problem since it is $\mathcal{NP}$-hard and is believed to be computationally difficult to solve. With the creation of these cryptosystems, there also came people who wanted to break them. In this section, we discuss the earliest knapsack cryptosystem created by Ralph C. Merkle and Martin E. Hellman [14] in 1978. Then we discuss how, a few years later, Adi Shamir[16] broke the basic Merkle-Hellman knapsack cryptosystem which led to the fall of the knapsack cryptosystem.

2.1 Basic Merkle-Hellman Knapsack Cryptosystem

The basic idea of the Merkle-Hellman cryptosystem is to create a knapsack problem which appears to be hard to solve but in actuality is easy to solve. They refer to this special knapsack as a *trapdoor knapsack*.

2.1.1 Creating a Trapdoor Knapsack

First, a private super-increasing set of nonnegative integers $S = \{s_i : 1 \leq i \leq n\}$, where for all i

$$s_i > \sum_{j=1}^{i-1} s_j,$$

is created. Then for any knapsack problem of the form $\sum_{i=1}^{n} s_i x_i = e$, where $e \in \mathbb{Z}^+$ and $x_i \in \{0,1\}$, $x_n = 1$ if and only if $e - \sum_{j=i+1}^{n} s_j x_j \geq s_i$. Note that your n would have to be large enough so that this trend will not be evident. If n were small, then we could just enumerate all possible solutions and find the optimal one.

Example 2.1 *Let* $S = \{40, 116, 215, 458, 982, 2024\}$ *be a super-increasing set. Find the solution to*

$$40x_1 + 116x_2 + 215x_3 + 458x_4 + 982x_5 + 2024x_6 = 2279$$

where $x_i \in \{0,1\}$ *for* $1 \leq i \leq 6$. *From above, since*

$$2279 > 2024 \Rightarrow x_6 = 1$$

$$2279 - 2024 = 255 > 215 \Rightarrow x_3 = 1$$

$$255 - 215 = 40 \Rightarrow x_1 = 1$$

Therefore, the only solution to the knapsack problem is $\{1, 0, 0, 1, 0, 1\}$.

In order to produce a trapdoor knapsack problem, you must choose a *decryption pair* (m, w) which is privately known, and where

$$m > \sum_{i=1}^{n} s_i, \tag{2.1}$$

$$2^{dn} \leq m \leq 2^{dn+1}, \tag{2.2}$$

where $d \geq 1$ is called an *expansion factor*,

$$1 \leq w < m, \tag{2.3}$$

and

$$gcd(m, w) = 1. \tag{2.4}$$

To create the trapdoor knapsack, one calculates

$$a_i = ws_i \pmod{m}. \tag{2.5}$$

This creates a pseudo-randomly distributed set $A = \{a_i, 1 \leq i \leq n\}$ that is published publicly. Any person then could take this set A and a binary plaintext and create and send a ciphertext that only the creator of the cryptosystem would be able to break. The interpretation of d is that it is the measure of how much longer the ciphertext is than the plaintext.

The idea of this is that if a person wishes to encode a binary plaintext all they have to do is use a_i, weight values, from the set A and calculate it as a solution to a knapsack problem, i.e. calculate

$$\sum_{i=1}^{n} a_i x_i = e^*.$$

The sum of the knapsack is now the ciphertext, e^*, that is sent to be cracked. Since A has a psuedo-random order and n is large, this knapsack problem is quite difficult to solve, and it is claimed that only the person who knows $S, m,$ and w would be able to break the code.

Notice that when calculating e^*, we assume that there may be many other solutions that also add up to e^*. However, we chose the knapsack weights certain to ensure that e^* has at most one solution, so that the plaintext is uniquely recoverable.

In order to decrypt e^*, the creator of the cryptosystem first calculates w^{-1}. Then, $e \equiv w^{-1}e^* \pmod m$ is calculated. Finally, the easy knapsack problem $\sum_{i=1}^{n} s_i x_i = e$ where $x_i \in \{0, 1\}$ is solved. The solution to this knapsack is the plaintext that the encoder sent to be decrypted. The following example demonstrates this process.

Example 2.2

Alice chooses a private $m = 10015$, $w = 23$ as her decryption pair, and the set S in Example 2.1 ($d = 2$ in this case). She then uses (2.5), which creates the set $A = \{920, 2668, 4945, 519, 2556, 6492\}$. Alice publishes A.

Bob wants to encrypt a plaintext, $e = 110010$, and send it to Alice. So Bob calculates the ciphertext, $920(1) + 2668(1) + 4945(0) + 519(0) + 2556(1) + 6492(0) = 6144$. He then sends Alice the ciphertext $e^* = 6144$.

Alice wants to decrypt the message e^*. First, Alice calculates $w^{-1} = 6967$. Then calculates

$$6144 \times 6967 \pmod{10015},$$

which is 1138. Now, can Alice can use her super-increasing set S and 1138 to decrypt Bob's message.

$$1138 > 982 \Rightarrow x_5 = 1$$

$$1138 - 982 = 156 > 116 \Rightarrow x_2 = 1$$

$$156 - 116 = 40 \Rightarrow x_1 = 1.$$

Thus, Alice finds that the message is $e = 110010$, which is the message that Bob sent. Therefore, the encryption and decryption were successful.

For their cryptosystem to be secure, Merkle and Hellman suggested that size of the knapsack problem should contain at least $n = 100$ knapsack items, m to be chosen uniformly from $[2^{2n+1}+1, 2^{2n+2}-1]$, s_1 be chosen uniformly from $[1, 2^n]$, s_2 be chosen uniformly from $[2^n+1, (2)2^n]$, s_3 be chosen uniformly from $[(3)2^n+1, (4)2^n]$, and the s_i where $4 \leq i \leq n$ be chosen uniformly from $[(2^{i-1}-1)2^n+1, (2^{i-1})2^n]$. This method ensures that m is still greater than the sum of s_i's. Also, to make the set more secure, one could also use a permutation σ on the set A in order to protect the corresponding easy knapsack weights, σ would also be part of the trapdoor information.

Not too long after Merkle and Hellman created their knapsack cryptosystem, Adi Shamir [16] broke the cryptosystem which started a chain reaction for the failure of various knapsack cryptosystems.

2.2 Attack on the Basic Merkle-Hellman Cryptosystem

Without the trapdoor information, m, w^{-1}, and S , Merkel and Hellman thought that it would be difficult to crack this cryptosystem. And since solving knapsacks are hard, they believed that their system was secure. However in 1981, Adi Shamir [16] discovered a strong attack on the basic Merkle-Hellman cryptosystem. The object of this attack was to find a *decryption pair* (w^*, m^*), where $\frac{w^*}{m^*}$ is sufficiently close to $\frac{w^{-1}}{m}$. As we will see, this is possible since any basic knapsack cryptosystem has infinitely many decryption pairs.

The following algorithm highlights the attack that Shamir [16] provided. In his paper, Lagarias outlined the performance of Shamir's attack on the cryptosystem and considered all the steps of his attack. Let us refer to this algorithm as *Algorithm S*, as does Lagarias.

Step 1. Estimate the modulus m by

$$\tilde{m} = \max_{1 \leq i \leq n} a_i \tag{2.6}$$

and estimate the expansion factor d by

$$d^* = \frac{1}{n} \log_2(n^2 \tilde{m}). \tag{2.7}$$

Step 2. Set

$$g = \max\{d^* + 2, 5\}. \tag{2.8}$$

In this step, we want to find the a_i in the public set A that correspond to the g smallest super-increasing elements of the private set S. Since in most cases the person creating the cryptosystem will use a permutation σ on the set A, in order to hide the order of the elements, we must run the rest of the algorithm $\binom{n}{g}$ times until a solution is found.

Step 3. Solve the integer program

$$|k_i a_1 - k_1 a_i| \leq b, \qquad 2 \leq i \leq g \tag{2.9}$$

$$1 \leq k_1 \leq b - 1, \tag{2.10}$$

where

$$b = \lfloor 2^{-n+g} \tilde{m} \rfloor. \tag{2.11}$$

If a solution $(k_1^{(0)}, \ldots, k_g^{(0)})$ is found, we create and attempt to solve two new integer programs by replacing (2.10) by the constraints

$$1 \leq k_1 < k_1^{(0)} \tag{2.12}$$

and

$$k_1^{(0)} < k_1 \leq b - 1, \tag{2.13}$$

respectively. If solutions are found for these two new integer programs, we continue to subdivide the k_1 regions according to the values of each $k_1^{(i)}$. For example, if a solution $(k_1^{(1)}, \ldots, k_g^{(1)})$ is found for the new constraint (2.12), we would create two new integer programs with constraints $1 \leq k_1 < k_1^{(1)}$ and $k_1^{(1)} \leq k_1 < k_1^{(0)}$ respectively. We continue in this manner until either $n \log_2 n$ solutions are found, with at most $2n \log_2 n$ integer programs examined, or the process produces no further solutions.

Note that although the integer program in *Step 3* seems hard to solve, it can be solved in polynomial. According to Kannan [9], it takes $O(g^{9g} L \log L)$, where $L = g \log \tilde{m}$, which is polynomial in terms of g.

Step 4. For each solution $(k_1^{(i)}, \ldots, k_g^{(i)})$ found in Step 3, calculate the the following rational numbers

$$\theta_j^{(i)} = \frac{k_1^{(i)}}{a_i} + \frac{j}{n^7 2^n \tilde{m}} \quad ; \quad 1 \leq j \leq n^7. \tag{2.14}$$

Then for each $\theta_j^{(i)}$, put the rational in lowest terms and set $\theta_j^{(i)} = \dfrac{w_j^*}{m_j^*}$. Next, we must check to see if (w_j^*, m_j^*) is a decryption pair for the set A. We must see that the properties (2.3) and (2.4) hold for w_j^* and m_j^*. If these hold, we must calculate

$$s_i^* = a_i w_j^{*-1} \pmod{m_j^*} \quad for \quad 1 \leq i \leq n. \tag{2.15}$$

Then if $S^* = \{s_1^*, \ldots, s_n^*\}$ is a super-increasing set with 2.1 holding for m_j^*, then the algorithm succeeds with (w_j^*, m_j^*) a decryption pair.

Example 2.3

Eve wants to know what Bob is trying to send to Alice in Example 2.2. All she has access to is the set A that Alice publicly provided and the ciphertext $e^* = 6144$ that Bob created. Recall, in this example $n = 6$. She first estimates the modulus m by (2.6) and d^* by (2.7) which implies $\tilde{m} = 6492$ and $d^* \approx 2.9724$. Thus, by (2.8) and (2.11) , $g = 5$ and $b = 3246$ respectively. Now, she can start solving the integer program that was presented in (2.9) and (2.10). Her integer program would be as follows:

$$\begin{aligned} |2668k_2 - 920k_1| &\leq 3245 \\ |4945k_3 - 920k_1| &\leq 3245 \\ |519k_4 - 920k_1| &\leq 3245 \\ |2556k_5 - 920k_1| &\leq 3245 \end{aligned} \tag{2.16}$$

$$1 \leq k_1 < 3246 \tag{2.17}$$

Thus, an initial solution to the above program would be $k^{(0)} = (1, 0, 0, 0, 0, 0)$. Then replacing (2.17) with (2.10) we get a solution $k^{(1)} = (2, 0, 0, 0, 0, 0)$. Continuing in the manner of *Step 3*, we can create new integer programs and find new solutions. Fortunately for this example, we need only the $k^{(1)}$ solution.

Next, Eve moves on to *Step 4* to calculate (2.14), in order to find a decryption pair. She finds that when $i = 1$, and $j = 22687$ that

$$\frac{w^*_{30764}}{m^*_{30764}} = \frac{7270262513}{3343913902080},$$

which is in lowest terms, so w^*_{30764} and m^*_{30764} are relatively prime to each other. She then finds that $w^{*-1}_{30764} = 1308506319377$. Eve then uses (2.15) to find that $S^* = \{16809078040, 48746326316, 90348794465, 300257634423, 628250247612, 1281714112284\}$. Notice that S^* is a super-increasing set with

$$m^*_{30764} > \sum_{i=1}^{n} s^*_i.$$

Therefore, Eve found a decryption pair.

Now, Eve is able to decrypt e^* by calculating

$$e^* w^{*-1}_{30764} \pmod{m^*_{30764}},$$

which is 693805651968. Thus,

$$693805651968 > 628250247612 \Rightarrow x_5 = 1$$

$$693805651968 - 628250247612 = 65555404356 > 48746326316 \Rightarrow x_3 = 1$$

$$65555404356 - 48746326316 = 16809078040 \Rightarrow x_1 = 1.$$

Therefore, Eve found that the message Bob was trying to send to Alice was $e = 110010$.

2.2.1 Rationale for Breaking Merkle-Hellman Cryptosystem

We will now discuss the rationale for *Algorithm S*. Notice that the decryption congruence for $1 \leq i \leq n$

$$w^{-1}a_i \equiv s_i \pmod{m} \tag{2.18}$$

is equivalent to

$$w^{-1}a_i - mt_i = s_i \tag{2.19}$$

for some $t_i \in \mathbb{Z}^+$. This implies that

$$\frac{w^{-1}}{m} - \frac{t_i}{a_i} = \frac{s_i}{ma_i}, \tag{2.20}$$

by multiplying (2.19) by $\frac{1}{ma_i}$. Then by subtracting (2.20) and (2.20), when $i = 1$, and then multiplying this difference by $a_i a_1$,we get

$$t_i a_1 - t_1 a_i = \frac{1}{m}(s_1 a_i - s_i a_1). \tag{2.21}$$

Since we know that

$$s_1 + s_2 + \cdots + s_n < m$$

$$s_1 + s_2 + \cdots + s_{n-1}$$

$$\vdots$$

$$s_1 + s_2 < s_3$$

$$s_1 < s_2,$$

we are able to show that for any super-increasing set S,

$$0 \leq s_i \leq 2^{-n+i} m. \tag{2.22}$$

We can use this to determine the following bound:

Bound 2.1 *For* $1 \leq i \leq g$, $|t_i a_1 - t_1 a_i| \leq 2^{-n+g} \tilde{m}$.

Proof *Case 1:* Suppose that $s_1 a_i - s_i a_1 \geq 0$. Then

$$\begin{aligned} 0 &\leq |s_1 a_i - s_i a_1| \\ &\leq 2^{-n+1} \tilde{m} \\ &\leq 2^{-n+1} \tilde{m} m \\ &\leq 2^{-n+g} \tilde{m} m \end{aligned}$$

by (2.22) and the fact that $1 \leq g$. Hence,

$$\begin{aligned}|t_i a_1 - t_1 a_i| &= \frac{1}{m}|s_1 a_i - s_i a_1| \\ &\leq \frac{1}{m}|2^{-n+g}\tilde{m}m| \\ &= |2^{-n+g}\tilde{m}|.\end{aligned}$$

Case 2: Suppose that $s_1 a_i - s_i a_1 \leq 0$. This implies that $s_i a_1 - s_1 a_i \geq 0$. Thus,

$$\begin{aligned}0 &\leq |s_i a_1 - s_1 a_i| \\ &\leq 2^{-n+i}\tilde{m} \\ &\leq 2^{-n+i}\tilde{m}m \\ &\leq 2^{-n+g}\tilde{m}m\end{aligned}$$

by (2.22) and the fact that $i \leq g$. Hence,

$$|t_i a_1 - t_1 a_i| \leq 2^{-n+g}\tilde{m}.$$

$\square$

So in this case, the integer program in *Step 3* has at least one solution, $(k_1, \ldots, k_g) = (t_1, \ldots, t_g)$.

Now suppose that *Step 3* yields a solution $(k_1^{(i)}, \ldots, k_g^{(i)})$ with $k_1^{(i)} = t_1$. Also, suppose that

$$\frac{1}{n^2}m \leq a_1 \leq m. \tag{2.23}$$

Under these assumptions, we can produce the following bound.

Bound 2.2 *Step 4 of Algorithm S is bound to find a pair* $\frac{w_j^*}{m_j^*}$. *with*

$$\left|\frac{w_j^*}{m_j^*} - \frac{w^{-1}}{m}\right| \leq \frac{2n^2}{2^n m}.$$

Proof By (2.14) and (2.20),

$$\begin{aligned}\left|\frac{w_j^*}{m_j^*} - \frac{w^{-1}}{m}\right| &= \left|\left(\frac{t_1}{a_1} + \frac{j}{n^2 2^n \tilde{m}}\right) - \left(\frac{s_1}{ma_1} + \frac{t_1}{a_1}\right)\right| \\ &= \left|\frac{j}{n^7 2^n \tilde{m}} - \frac{s_1}{ma_1}\right|\end{aligned}$$

Case 1: Let $\frac{j}{n^7 2^n \tilde{m}}$ be as large as possible, i.e with $j = n^7$, and let $\frac{s_1}{ma_1}$ be as small as possible, i.e. with $s_1 = 0$. Then

$$\left|\frac{j}{n^7 2^n \tilde{m}} - \frac{s_1}{ma_1}\right| \leq \frac{n^7}{n^7 2^n \tilde{m}} = \frac{1}{2^n \tilde{m}} \leq \frac{2n^2}{2^n m}.$$

Case 2: Let $\frac{s_1}{ma_1}$ be as large as possible, i.e. with $s_1 = 2^{-n+1}m$ by (2.22), and let $\frac{j}{n^7 2^n \tilde{m}}$ be as small as possible. Then

$$\left|\frac{j}{n^7 2^n \tilde{m}} - \frac{s_1}{ma_1}\right| \leq \frac{2^{-n+1}m}{ma_1} = \frac{2}{2^n a_1} \leq \frac{2n^2}{2^n m} \quad \text{by (2.23)}$$

□

Now, let $(w^*, m^*) = (w_j^*, m_j^*)$ and define λ by

$$m^* = \lambda m. \tag{2.24}$$

Then from Bound 2.2, we know that

$$\frac{w^*}{m^*} = \frac{w^{-1}}{m} + \delta \text{ for some } \delta \in \mathbb{R}. \tag{2.25}$$

Then, by (2.24), we know that

$$\begin{aligned} w^* &= \frac{w^{-1}m^*}{m} + \delta m^* \\ &= w^{-1}\lambda + \delta m \lambda. \end{aligned}$$

Let $\epsilon = \delta m$. Then

$$w^* = \lambda(w + \epsilon). \tag{2.26}$$

Thus, by Bound 2.2,

$$|\epsilon| \leq \frac{2n^2}{2^n}. \tag{2.27}$$

Hence by (2.19), (2.24), and (2.26),

$$w^* a_i - m^* t_i = \lambda(s_i + \epsilon a_i),$$

where

$$|\epsilon a_i| \leq \frac{n^2}{2^n} m.$$

Then if $s_i^* = s_i + \epsilon a_i$ is a super-increasing set, then (w^*, m^*) will almost always be a decryption pair. Notice that the λ was dropped, due to the fact that it does not affect the super-increasing set.

Lagarias [12] then goes on to say that due to the way *Algorithm S* was created, the algorithm can only fail if:

1. The bound in (2.23) can fail to hold.
2. *Step 3* may fail to find a solution $(k_1^{(i)}, \ldots, k_g^{(i)})$ with $k_1^{(i)} = t_1$.
3. Step 4 may fail to find a super-increasing set.

He then proceeds to bound each of these failures, which tend to go away as $n \to \infty$.

As we showed before, knapsack problems can not in general be solved in polynomial time. However, when created in the Merkel-Hellman cryptosystem, they are quite easy to solve. Although this cryptosystem was a good attempt at creating a secure cryptosystem, *Algorithm S* can break the cryptosystem in polynomial time.

Theorem 2.1 *Algorithm S runs to completion time in $O(n^{g+10}L\log L)$ where $L = g\log\tilde{m}$.*

Proof In order to perform Step 1 of Algorithm S, it has a runtime of $O(n\log n)$ find $\tilde{m}$. According to Kannan [9], it takes $O(g^{9g}L\log L)$ to solve the integer program of Step 3. And it takes a runtime of $O(n^7)$ for Step 4. And , we need to perform $\binom{n}{g}$,which is $O(n^g)$, until a solution is found and solve at most $2n\log_2 n$ integer programs. Thus, combining these runtimes, we get a total runtime of $O(n^{g+9}g^{9g}(2\log_2 n)(\log n)(L\log L))$. And since $2g^{9g}$ is a constant and $O(\log_2 n\log n) \leq O(n)$,

$$O(n^{g+9}g^{9g}(2\log_2 n)(\log n)(L\log L)) \leq O(n^{g+10}(L\log L)).$$

$\square$

Many other attempts at creating a knapsack cryptosystems have been approached, but their security was in question. Merkle and Hellman [14] created an iterated version of the cryptosystem that was discussed in this thesis that appeared to be very secure.

However, shortly after the original cryptosystem was broken by Shamir [16], Ernest Brickell [3] broke the iterated knapsack method in 1988. Another attempt was at the knapsack cryptostem was created by Chor and Rivest [4] in 1985. However, this was also broken by Serge Vaudenay [18]. Many other attempts at creating a secure knapsack cryptosystem have been approached, but have not had the power to sustain attacks.

3 Generating Functions

In this section, we describe what is a generating function and how you can represent a knapsack problem as a generating function. Also, we use the generating function of a knapsack problem and apply it to the presidential race.

3.1 Introduction

Generating functions are used to solve many combinatorial problems, including selection and arrangement problems with repetition. The following gives the definition of a generating function.

Definition 3.1 *Suppose c_r is the number of ways to select r objects from n objects, then $g(x)$ is a generating function for c_r if $g(x)$ has the polynomial expansion*

$$g(x) = c_0 + c_1x + \cdots + c_rx^r + \cdots + c_nx^n.$$

The definition of a generating function allows us to understand the interpretation of the generating function. Suppose we have $g(x)$ as in Definition 3.1. Then in order to determine the number of ways to select r objects from n objects, all we have to do is find the coefficient of x^r (i.e. c_r) and that is the number of ways to select r objects from n objects. A famous example of a generating function is the binomial expansion,

$$(1+x)^n = 1 + \binom{n}{1}x + \cdots + \binom{n}{r}x^r + \cdots + \binom{n}{n}x^n.$$

Here, $g(x) = (1+x)^n$ is the generating function for $c_r = C(n,r)$.

The following example demonstrates a combinatorial problem than can be solved using a generating function.

Example 3.1 *Find the number of ways to select five balls from three green, three white, and four red balls.*

The desired generating function to solve this problem is

$$g(x) = (1+x+x^2+x^3)^2(1+x+x^2+x^3+x^4). \tag{3.1}$$

Here, the $(1+x+x^2+x^3)^2$ term represents the number of green and white balls available and the $(1+x+x^2+x^3+x^4)$ represents the number of red balls available then expanding (3.1) , we get that

$$g(x) = 1 + 3x + 13x^4 + 6x^2 + 10x^3 + 14x^5 + 13x^6 + 10x^7 + 6x^8 + 3x^9 + x^{10}.$$

Thus, there are 14 ways to choose five balls from the given balls.

We can also find the number of solutions to a knapsack problem by finding a generating function for the problem. Suppose that we have the following knapsack problem

$$\sum_{i=1}^{n} a_i x_i = c,$$

where $x_i \in \{0, 1\}$. A generating function for this problem is given by

$$k(v) = (1 + v^{a_1})(1 + v^{a_2}) \cdots (1 + v^{a_n}).$$

Then in order to find the number of solutions with capacity c, we would expand $k(v)$ and find the coefficient of the v^c term. For example, suppose we have the following knapsack problem

$$8x_1 + 3x_2 + 4x_3 + 6x_4 + 2x_5 + x_6 = 17.$$

Then the generating function for this knapsack problem would be

$$k(v) = (1 + v^8)(1 + v^3)(1 + v^4)(1 + v^6)(1 + v^2)(1 + v).$$

Then expanding $k(v)$ we get

$$\begin{aligned} k(v) &= 1 + v + v^2 + 2v^4 + 2v^5 + 3v^6 + 3v^7 + 3v^8 \\ &\quad + 4v^9 + 4v^{10} + 4v^{11} + 4v^{12} + 4v^{13} + 4v^{14} + 4v^{15} \\ &\quad + 3v^{16} + 3v^{17} + 3v^{18} + 2v^{19} + 2v^{20} + 2v^{21} \\ &\quad + v^{22} + v^{23} + v^{24}. \end{aligned}$$

Thus, by looking at the v^{17} term, we get there are three solutions to the presented knapsack problem.

In the next section, we will use the knapsack problem and its generating functions to help us predict a presidential race using electoral votes.

3.2 Predicting the Presidential Election

Throughout the history of the United States of America, there have been four times that the elected president did not receive a majority of the popular vote, in 1824, 1876, 1888 and 2000. In each of these elections, the Electoral College elected a president whom the majority of the people did not support. This can happen since that each citizen's vote does not count directly toward the presidential candidate, each vote goes toward an electoral candidate. Each state and Washington D.C. is allowed a different number of votes toward the presidency, equal to the amount of senators and representatives that they have. This way, smaller states are not lost in the vote and large states get the representation due to their population. There are a total of 538 Electoral College Votes. Table 1 shows the number of votes that each state has for the 2012 election. In each state other than Maine and Nebraska, the winner takes all Electoral College votes. A winner of the election is determined if the candidate receives a majority of the electoral votes, i.e. at least 270 of the votes. There is a chance of a tie, which will be demonstrated. If a tie does occur, then another election system process is set off in the House and Senate.

We set up a knapsack problem to help us determine the probability that the Electoral College vote produces a tie. Let x_j be equal to one if the Republican candidate is chosen and zero otherwise, for $j = 1, ..., 51$. And we must find the number of solutions to

$$55x_1 + 38x_2 + \cdots + 3x_{49} + 3x_{50} + 3x_{51} = 269,$$

which will determine the possible number of ties. From our section on generating functions, we can easily determine this number. The generating function for this

State	Vote	Proj.	State	Vote	Proj.	State	Vote	Proj.
CA	55	O	MO	10	U	UT	6	R
TX	38	R	MD	10	O	NE	5	O
NY	29	O	MN	10	O	NM	5	U
FL	29	U	WI	10	U	WV	5	R
PA	20	U	AL	9	R	HI	4	O
IL	20	O	CO	9	U	ID	4	R
OH	18	U	SC	9	R	ME	4	O
MI	16	U	LA	8	R	NH	4	U
GA	16	R	KY	8	R	RI	4	O
NC	15	U	CT	7	O	AK	3	R
NJ	14	O	OK	7	R	DE	3	O
VA	13	U	OR	7	O	DC	3	O
WA	12	O	IA	6	U	MT	3	R
MA	11	O	AR	6	R	ND	3	R
IN	11	R	KS	6	R	SD	3	R
AZ	11	R	MS	6	R	VT	3	U
TN	11	R	NV	6	U	WY	3	R

Table 1: Number of electoral votes for 2012 and the projected states for (O)bama, (R)epublican, and (U)ndecided [1].

problem is

$$k(v) = (1 + v^{55})(1 + v^{38}) \cdots (1 + v^3)(1 + v^3)(1 + v^3).$$

Then expanding the solution and finding the coefficient of the v^{269} term. We determine that there are $t = 16,976,480,564,070$ ways to have a tie. And since there are 2^{51} possible solutions to this knapsack problem,

$$P(tie) = \frac{t}{2^{51}} \approx 0.0075.$$

We can also predict the probability that President Obama or the Republican candidate will win the 2012 presidential election as of the time of this document. Table 1 shows which states the two candidates are predicted to win and which states are undecided, as of April 19, 2012. According to the polling data [1], President Obama currently predicted to have 198 electoral votes, so he needs at least 62 votes from the remaining 14 undecided states. We can determine President Obama's probability of winning the 2012 election by creating the following knapsack problem. Let x_j, where $1 \leq j \leq 14$, be one if Obama wins the undecided state j and zero otherwise. We must then determine the number of solutions to the equation

$$29x_1+20x_2+18x_3+16x_4+15x_5+13x_6+10x_7+10x_8+9x_9+6x_{10}+6x_{11}+5x_{12}+4x_{13}+3x_{14}$$

that is greater than or equal to 62. Therefore, we create the generating function

$$k(v) = (1+v^{29})(1+v^{20})\cdots(1+v^4)(1+v^3).$$

After expanding and summing all the coefficients to the v^k terms, where $k \geq 62$, we find that there are 12781 solutions to this problem. Thus assuming each candidate has the same probability of winning an undecided state, the probability of President Obama winning the 2012 presidential election is

$$P(\text{ObamaWinning}) = \frac{12781}{2^{14}} \approx 0.78.$$

Therefore, as of April 19, 2012, President Obama has a high probability of being reelected. Of course, these numbers may change as new information affects polling data and a person's preference.

As demonstrated, the knapsack problem has many real world applications, including being used to see the probability of a candidate winning a presidential election.

4 Conclusion

As you can see, the 0-1 Knapsack Problem is a surprising and powerful problem. In spite of its simple structure, it has a vast amount of applications. Recall, we provided just a few applications of this problem, although there are many. There are many algorithms to obtain an optimal solution to the problem; we provided a branch-and bound algorithm to perform this task. New ways to solve the knapsack problem and improve on existing ways will continue to be developed. Of course, until the $\mathcal{NP}-$hard issue is resolved, these will always be knapsack problems that are simply too large to solve. The 0-1 Knapsack Problem has been studied extensively throughout the past fifty years and will continue to be analyzed.

5 References

[1] *270toWin* 2012, accessed 19 April 2012, <http://www270towin.com> .

[2] R. Bellman, Some applications of the theory of dynamic programming - a review, *Operations Research* vol 2, 1954, pp. 275-288.

[3] E. F. Brickell, Breaking Iterated Knapsacks, *Advances in Cryptology-Proc. Crypto 84*, Springer-Verlag, Berlin, 1985, pp. 342-358.

[4] B. Chor, and R. Rivest, A knapsack type cryptosystem based on arithmetic in finite fields, IEEE Trans. Inform. Theory, vIT-34 i5, 1985, pp. 901-909.

[5] G.B. Dantzig, Discrete variable extremum problems, *Operations Research* vol. 5, 1957, pp. 266-277.

[6] M.R. Garey and D.S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.

[7] N. Howgrave-Graham and A. Joux, New generic algorithm for hard knapsacks, *EUROCRYPT'2010*, 2010, pp. 235-256.

[8] D.S. Johnson, Approximation algorithms for combinatorial problems, *Journal of Computer and System Sciences* vol. 9, 1974, pp. 256-278.

[9] R. Kannan, Improved Alogorithms for Integer Programming and Related Lattice Problems,*Proc. 15th Annual ACM Symposium on theory of Computing*, 1983, pp. 193-206.

[10] R.M. Karp, Reducibility among combinatorial problems, In R.E.Miller, J.W. Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, 1972, pp. 85-103.

[11] P.J. Kolesar, A branch and bound algorithm for the knapsack problem. *Management Science* vol. 13, 1967, pp. 723-735.

[12] J. C. Lagarias, Performance Analysis of Shamirs Attack on the Basic Merkle-Hellman Knapsack Cryptosystem, *Proc. 11th Intern. Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, vol. 172, Springer-Verlag, Berlin, 1984, pp. 312-323.

[13] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer implementataions*. Wiley, Chichester, England, 1990.

[14] R. C. Merkle and M. E. Hellman, Hiding Information and Signatures in Trapdoor Knapsacks, *IEEE Trans. Inform. Theory*, vol. 24, 1978, pp. 525-530.

[15] D. Pisinger, Algorithms for Knapsack Problems, Ph.D. Thesis, DIKU, University of Copenhagen Report 95/1, 1995.

[16] A. Shamir, A Polynomial Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem, *IEEE Trans. Inform. Theory*, vol. IT-30, 1984, pp. 699-70.

[17] A. Tucker, *Applied Combinatorics (5th ed.)*, John Wiley & Sons, 2007.

[18] S. Vaudenay, Cryptanalysis of the Chor-Rivest Cryptosystem, *Journal of Cryptology 14*, 2001, pp. 87-100.